Prolog for

Pattern-Oriented Software Diagnostics

Logic Programming for Memory Dump, Trace, and Log Analysis

```
frame(103, 0, ntdll, 'NtWaitForAlertByThreadId', 0x14).
frame(103, 1, ntdll, 'RtlpWaitOnCriticalSection', 0x1cf).
frame(103, 2, ntdll, 'RtlpEnterCriticalSectionContended',
      0x1cd).
frame(103, 3, ntdll, 'RtlEnterCriticalSection', 0x40).
frame(103, 4, dbclient, 'Connection::Send', 0x58).
frame(103, 5, appserver, 'Session::Commit', 0x9e).
frame(103, 6, appserver, 'Worker::Run', 0x120).
frame(103, 7, ucrtbase, 'thread_start<...>', 0x42).
frame(103, 8, kernel32, 'BaseThreadInitThunk', 0x14).
frame(103, 9, ntdll, 'RtlUserThreadStart', 0x21).
waits_for(103, critical_section, 0x7ff6_1d13_a140).
waits_for(104, critical_section, 0x7ff6_1d13_a0c0).

?- deadlock(critical_section, C).
C = [103, 104].
```

Concept by **Dmitry Vostokov**

PROLOG FOR PATTERN-ORIENTED SOFTWARE DIAGNOSTICS

LOGIC PROGRAMMING FOR MEMORY DUMP, TRACE, AND LOG ANALYSIS

Dmitry Vostokov

Software Diagnostics Institute

Prolog for Pattern-Oriented Software Diagnostics
Logic Programming for Memory Dump, Trace, and Log Analysis

Concept by Dmitry Vostokov
Developed with AI assistance (GPT-6 Astra and Fable 5.1)

Published by OpenTask, Republic of Ireland
Copyright © 2026 by Dmitry Vostokov
Copyright © 2026 by OpenTask
Copyright © 2026 by Software Diagnostics Institute
ISBN-13: 978-1-919135-06-9

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the publisher's prior written permission.

Product and company names mentioned in this book may be trademarks of their owners.

OpenTask books are available through booksellers and distributors worldwide. For further information or comments, send requests to press@opentask.com.

Version 20
October 2026

Table of Contents

Table of Contents **3**

Preface **23**

About the Concept Author **25**

Part I. Prolog from the Ground Up **26**

Chapter 1. Logic Programming and Diagnostics **27**

1.1 Diagnosis as Inference over Artifacts *27*

1.2 Facts, Rules, and Queries *28*

1.3 The Declarative and Procedural Readings *30*

1.4 Installing SWI-Prolog *31*

1.5 The Toplevel *32*

1.6 Errors and Warnings at Load Time *34*

1.7 Diagnostic Use *36*

Chapter 2. Terms **40**

2.1 Atoms *40*

2.2 Numbers *41*

2.3 Variables *43*

2.4 Compound Terms *44*

2.5 Operators and Canonical Form *45*

2.6 Lists *46*

2.7 Strings, Code Lists, and Char Lists *47*

2.8 Type Tests *48*

2.9 The Standard Order of Terms *49*

2.10 Artifacts as Terms *50*

2.11 Diagnostic Use *51*

Chapter 3. Unification **55**

3.1 Identity and Unifiability *55*

3.2 How Unification Works *57*

3.3 Unification as Pattern Matching *58*

3.4 The Occurs Check and Cyclic Terms *59*

3.5 Deciding Without Binding *61*

3.6 The Variables of a Term, and Copies *62*

3.7 Diagnostic Use *63*

Chapter 4. Clauses, Queries, and Search **67**

4.1 Horn Clauses *67*

4.2 Goal Reduction *68*

4.3 The Search Tree *70*

4.4 Conjunction, Disjunction, True, and Fail *71*

4.5 The Order of Clauses and Goals *73*

4.6 The Tracer *76*

4.7 Diagnostic Use *77*

Chapter 5. Lists and Recursion **81**

5.1 Lists as Terms *81*

5.2 Recursion over Lists *82*

5.3 The List Library *85*

5.4 Accumulators and Tail Recursion *88*

5.5 Difference Lists *89*

5.6 Diagnostic Use *90*

Chapter 6. Arithmetic **96**

6.1 Evaluation *96*

6.2 Comparison *97*

6.3 Integers and Floats *98*

6.4 Bits *100*

6.5 Generating Integers *101*

6.6 Formatted Output *102*

6.7 Diagnostic Use *105*

Chapter 7. Control **111**

7.1 Determinism and Choice Points *111*

7.2 The Cut *112*

7.3 If-Then-Else *115*

7.4 Negation as Failure *117*

7.5 Loops *119*

7.6 Exceptions *120*

7.7 Diagnostic Use *123*

Chapter 8. Atoms, Strings, and Term Inspection **132**

8.1 Text in Four Forms *132*

8.2 Concatenation, Substrings, and Splitting *134*

8.3 Terms as Text *137*

8.4 Term Inspection *139*

8.5 Diagnostic Use *142*

Chapter 9. Collecting Solutions and Aggregation **147**

9.1 Collecting All Solutions *147*

9.2 Bags, Sets, and Free Variables *149*

9.3 Aggregation *150*

9.4 Solution Sequences *152*

9.5 Pairs and Sorting *154*

9.6 Diagnostic Use *156*

Chapter 10. Higher-Order Predicates and Libraries **164**

10.1 Goals as Arguments *164*

10.2 Mapping Lists *167*

10.3 Filtering and Partitioning *168*

10.4 Folding *169*

10.5 Lambda Expressions *171*

10.6 Association Lists *173*

10.7 Dicts *175*

10.8 Graphs *178*

10.9 Diagnostic Use *179*

Chapter 11. The Dynamic Database, Global State, and Tabling **184**

11.1 Static and Dynamic Predicates *184*

11.2 Adding and Removing Clauses *185*

11.3 The Logical Update View *186*

11.4 Inspecting the Program *187*

11.5 Global Variables *189*

11.6 Tabling *190*

11.7 Diagnostic Use *195*

Chapter 12. Definite Clause Grammars **199**

12.1 Grammar Rules *199*

12.2 The Translation *200*

12.3 The Basics Library *203*

12.4 Sequences *205*

12.5 A Frame Line *206*

12.6 A Log Line *209*

12.7 Grammars over Files *210*

12.8 Diagnostic Use *213*

Chapter 13. Programs, Modules, Files, and Tooling **218**

13.1 Modules *218*

13.2 Programs and Scripts *223*

13.3 Reading and Writing Files *225*

13.4 CSV, JSON, and Regular Expressions *227*

13.5 Debugging, Assertions, and Unit Tests *230*

13.6 Timing, Indexing, and Inspection *233*

13.7 Diagnostic Use *235*

Chapter 14. Constraints and Coroutining **238**

14.1 Coroutining *238*

14.2 Constraints over Integers *240*

14.3 Global Constraints and Reification *242*

14.4 Reconstructing an Event Order *244*

14.5 Deducing a Counter *248*

14.6 Thread Timelines *251*

14.7 Diagnostic Use *252*

Part II. Diagnostic Artifacts as Terms **257**

Chapter 15. Memory Dumps as Knowledge Bases **258**

15.1 What a Dump Contains and What a Debugger Derives *258*

15.2 The Schema *259*

15.3 From Debugger Output to Facts *264*

15.4 Normalization *271*

15.5 Unknowns and Provenance *274*

15.6 Kernel and Complete Memory Dumps *284*

15.7 The Shared Dump *285*

Chapter 16. Software Traces and Logs as Knowledge Bases **289**

16.1 Messages and Traces *289*

16.2 The msg/6 Schema *290*

16.3 Loading a Text Log *291*

16.4 The Shared Trace *294*

16.5 Loading and Saving CSV *296*

16.6 Derived Relations *297*

16.7 Indexing and Scale *300*

16.8 Several Traces in One Knowledge Base *301*

Chapter 17. The Pattern as a Predicate **303**

17.1 Description and Recognizer *303*

17.2 The Three Forms *304*

17.3 Evidence *306*

17.4 Thresholds as Facts *309*

17.5 Composition *312*

17.6 The Catalog as Data *316*

17.7 Reports and Severity *318*

17.8 Absence and the Closed World *320*

17.9 Determinism *323*

17.10 Testing Recognizers *324*

17.11 Conventions for Parts III and IV *326*

Part III. Memory Dump Analysis Patterns **327**

Chapter 18. Stack Traces and Threads **328**

18.1 Stack Trace *329*

18.2 Stack Trace Collection (Unmanaged Space) *330*

18.3 Stack Trace Collection (Predicate) *331*

18.4 Main Thread *332*

18.5 Special Thread *333*

18.6 Passive Thread *334*

18.7 Spiking Thread *335*

18.8 Blocked Thread (Software) *336*

18.9 Active Thread *337*

18.10 Thread Age *338*

18.11 Stack Trace Signature *338*

18.12 Least Common Frame *339*

18.13 Stack Trace Set *341*

18.14 Frame Regularity *342*

18.15 Caller-n-Callee *343*

18.16 Procedure Call Chain *344*

18.17 Incorrect Stack Trace *345*

18.18 Truncated Stack Trace *346*

18.19 Coincidental Symbolic Information *347*

18.20 Glued Stack Trace *349*

18.21 Execution Residue (User Space) *350*

Chapter 19. Exceptions and Faults **352**

19.1 Exception Stack Trace *358*

19.2 Fault Context *359*

19.3 NULL Pointer (Data) *360*

19.4 NULL Pointer (Code) *361*

19.5 Invalid Pointer (General) *362*

19.6 Wild Pointer *363*

19.7 Wild Code *364*

19.8 Local Buffer Overflow (User Space) *365*

19.9 Stack Overflow (User Mode) *366*

19.10 Divide by Zero (User Mode) *371*

19.11 Nested Exceptions (User Space) *372*

19.12 Multiple Exceptions (User Space) *373*

19.13 Hidden Exception (User Space) *374*

19.14 Handled Exception (User Space) *375*

19.15 C++ Exception *376*

19.16 Managed Code Exception *376*

19.17 Stored Exception *377*

19.18 Exception Module *378*

19.19 Problem Module *379*

19.20 Crash Signature *380*

Chapter 20. Synchronization Patterns 382

20.1 Wait Chain (Critical Sections) *384*

20.2 Wait Chain (Mutex Objects) *386*

20.3 Wait Chain (Thread Objects) *387*

20.4 Deadlock (Critical Sections) *388*

20.5 Deadlock (Mixed Objects) *394*

20.6 Blocked Queue *396*

20.7 High Contention (Critical Sections) *397*

20.8 Critical Section Corruption *398*

20.9 Swarm of Shared Locks *400*

20.10 Dispatch Level Spin *400*

20.11 Waiting Thread Time *401*

20.12 Thread Starvation (Normal Priority) *402*

20.13 Blocking Module *403*

Chapter 21. Memory, Heaps, Handles, and Leaks 405

21.1 Insufficient Memory (Committed Memory) *406*

21.2 Handle Leak *409*

21.3 Insufficient Memory (Handle Leak) *410*

21.4 Memory Leak (Process Heap) *411*

21.5 Memory Fluctuation *413*

21.6 Dynamic Memory Corruption (Process Heap) *414*

21.7 Double Free (Process Heap) *416*

21.8 Invalid Handle (General) *419*

21.9 Paged Out Data *420*

21.10 Shared Buffer Overwrite *421*

21.11 Value References *421*

21.12 Regular Data *423*

21.13 Data Correlation (Function Parameters) *424*

Chapter 22. Modules, Versions, and Environment 426

22.1 Module Variety *427*

22.2 Not My Version *429*

22.3 Unknown Component *430*

22.4 No Component Symbols *431*

22.5 Missing Component (General) *432*

22.6 Hooked Functions (User Space) *433*

22.7 Hooked Modules *434*

22.8 Hidden Module *435*

22.9 Changed Environment *436*

22.10 Virtualized Process (WOW64) *437*

22.11 Deviant Module *438*

22.12 Same Vendor *439*

22.13 Top Module *440*

22.14 Origin Module *441*

22.15 Module Hint *442*

22.16 Historical Information *443*

22.17 Last Error Collection *443*

22.18 Zombie Processes, Special Process, Frozen Process, and Busy System *445*

22.19 Lateral Damage *446*

22.20 Ubiquitous Component *447*

Chapter 23. Dump Quality, Composition, and Reports **449**

23.1 Manual Dump (Process) *451*

23.2 Truncated Dump *452*

23.3 Inconsistent Dump *453*

23.4 Early Crash Dump *454*

23.5 False Positive Dump *455*

23.6 Corrupt Dump *457*

23.7 No Current Thread *458*

23.8 Deadlock (Critical Sections), Blocked Thread (Software), and Wait Chain (Critical Sections) *458*

23.9 Step Dumps *460*

23.10 The Analysis Report *462*

Part IV. Trace and Log Analysis Patterns **481**

Chapter 24. Trace Structure **482**

24.1 Basic Facts *483*

24.2 Trace Dimension *484*

24.3 Message Set *485*

24.4 Vocabulary Index *487*

24.5 Trace Partition *489*

24.6 Message Space *490*

24.7 Head of a Message Set *491*

24.8 Tail of a Message Set *492*

24.9 Trace Frames *493*

24.10 Focus of Tracing *494*

24.11 Trace Mask *496*

24.12 Trace Extension *496*

24.13 Truncated Trace *498*

24.14 Sparse Trace *499*

24.15 Visibility Limit *500*

24.16 Empty Trace *500*

24.17 Singleton Event *501*

24.18 Message Directory *503*

24.19 Trace Summary *505*

24.20 Trace Viewpoints *506*

24.21 Working Set *508*

24.22 Paratext *509*

Chapter 25. Time 510

25.1 Time Delta *511*

25.2 Discontinuity *513*

25.3 Correlated Discontinuity *514*

25.4 Statement Density and Current *515*

25.5 Density Distribution *518*

25.6 Relative Density *520*

25.7 Trace Acceleration *521*

25.8 Delay Dynamics *522*

25.9 Periodic Error *523*

25.10 Periodic Message Block *525*

25.11 Timeout *527*

25.12 Significant Interval *528*

25.13 Punctuated Activity *529*

25.14 Milestones *531*

25.15 Last Activity *532*

25.16 No Activity *533*

25.17 Blackout *535*

Chapter 26. Activities and Threads of Activity 537

26.1 Thread of Activity *537*

26.2 Adjoint Thread of Activity *538*

26.3 Activity Region *540*

26.4 Activity Theatre *541*

26.5 Activity Overlap *542*

26.6 Fiber of Activity *543*

26.7 Fiber Bundle *543*

26.8 Sheaf of Activities *544*

26.9 Coupled Activities *545*

26.10 Glued Activity *547*

26.11 Resume Activity *548*

26.12 Background and Foreground Components *549*

26.13 Guest Component *552*

26.14 Watch Thread *553*

26.15 Error Thread *554*

26.16 Error Distribution *555*

26.17 Message Interleave *556*

26.18 Dominant Event Sequence *557*

26.19 Data Flow *559*

26.20 Use Case Trail *561*

Chapter 27. Messages **564**

27.1 Error Message *565*

27.2 Significant Event *566*

27.3 Message Invariant *567*

27.4 Counter Value *568*

27.5 Linked Messages *570*

27.6 Shared Point *571*

27.7 Opposition Messages *572*

27.8 Characteristic Message Block *574*

27.9 Missing Message *575*

27.10 Ordinals in Messages *576*

27.11 Anchor Messages *577*

27.12 Pivot Message *579*

27.13 Message Context *580*

27.14 Indirect Message *581*

27.15 Identification Messages *582*

27.16 Implementation Discourse *583*

27.17 Hidden Error *584*

27.18 Silent Messages *584*

27.19 Exception Stack Trace *585*

27.20 Message Change *587*

27.21 False Positive Error *589*

27.22 Diegetic Messages *590*

27.23 Marked Message *591*

27.24 Palimpsest Messages *592*

27.25 Translated Message *593*

27.26 Recovered Messages *595*

27.27 Message Annotations *596*

27.28 Message Cover *597*

27.29 Defamiliarizing Effect *599*

Chapter 28. Sequences, Motifs, and Correlation 600

28.1 Event Sequence Order *603*

28.2 Event Sequence Phase *604*

28.3 Macrofunction *606*

28.4 Motif *607*

28.5 Dominant Event Sequence *609*

28.6 Inter-Correlation and Intra-Correlation *610*

28.7 Split Trace and Master Trace *611*

28.8 Quotient Trace *613*

28.9 Impossible Trace *614*

28.10 Data Reversal *615*

28.11 Dialogue *616*

28.12 Minimal Trace *617*

28.13 Visitor Trace *618*

28.14 Semantic Mapping *618*

28.15 Karnaugh Map *620*

28.16 State Dump *621*

28.17 Small DA+TA *622*

Part V. Engineering Diagnostic Knowledge 625

Chapter 29. Meta-Interpreters and Explanation 626

29.1 Clauses as Data *626*

29.2 The Vanilla Meta-Interpreter *630*

29.3 Proof Trees *632*

29.4 Printing Proofs as Explanations *635*

29.5 Evidence from Proofs *637*

29.6 Depth-Bounded Interpretation *640*

29.7 Counting Inferences *642*

29.8 A Tracing Meta-Interpreter *643*

29.9 Certainty Factors *645*

29.10 Meta-Interpreters and Analysis Pattern Networks *647*

Chapter 30. Scale and Performance **648**

30.1 Measuring *648*

30.2 Indexing *650*

30.3 A Synthetic Trace *651*

30.4 Discontinuity at Scale *653*

30.5 Facts, Association Lists, and Dicts *654*

30.6 Distinct Values and library(nb_set) *655*

30.7 Tabling as Memoization *656*

30.8 Collecting Less *657*

30.9 Streaming from Files *658*

30.10 Stacks and Garbage Collection *660*

30.11 Fact Files and Quick-Load Files *661*

30.12 Profiling *662*

Chapter 31. Integration: Scripts, Embedding, and Debugger Extensions **663**

31.1 Command-Line Tools *663*

31.2 JSON *666*

31.3 CSV *667*

31.4 An HTTP Service *668*

31.5 Embedding the Engine in C *669*

31.6 A Debugger Extension: A Design *671*

31.7 PrologWinDbg: The Extension as Built *673*

31.8 The Machine Query Interface *685*

31.9 Prolog as a Verifier for LLM-Based Agents *686*

Appendix A. SWI-Prolog Quick Reference **699**

A.1 Toplevel and Programs *699*

A.2 Terms and Types *699*

A.3 Unification and Comparison *700*

A.4 Control *700*

A.5 Arithmetic *701*

A.6 Atoms, Strings, and Text *701*

A.7 Lists *701*

A.8 Solutions and Aggregation *702*

A.9 Data Structures *702*

A.10 Database and State *702*

A.11 Grammars *703*

A.12 Files and Interchange *703*

A.13 Debugging, Testing, and Measurement *703*

Appendix B. The Shared Artifacts **704**

B.1 dump1.pl *704*

B.2 trace1.pl *708*

B.3 patterns_dump.pl *711*

Appendix C. Pattern Cross-Reference 733

C.1 Memory Analysis Patterns 733

C.2 Trace and Log Analysis Patterns 739

Appendix D. Glossary 742

Bibliography 747

Index 750

Preface

Software diagnostics, as practiced in this book, is the discipline of reasoning from artifacts, such as memory dumps, software traces, and logs, to statements about the software that produced them. Its vocabulary is a pattern language: a catalog of named, recurring structures of abnormal behavior, such as **Spiking Thread** (page 335), Wait Chain, Deadlock, **Discontinuity** (page 513), and **Significant Event** (page 566), each a description of what to look for, where, and what it may mean. The catalogs grew from practice, and they were written for people. This book asks what happens when their descriptions are written for a machine as well, in a language in which a description of structure is also a procedure for finding it.

Prolog is that language. A memory dump rendered into facts is a knowledge base; a pattern written as a rule is a query that enumerates its instances; and the search that Prolog performs when it answers a query is the search the analyst performs when reading a stack trace or a trace of messages, made explicit, repeatable, and open to inspection. The fit is close enough that the book's two subjects can be taught together: Prolog from the ground up, with every construct motivated by a diagnostic use, and the pattern catalogs from the ground up, with every pattern that admits a formal recognizer given one.

The book has five parts. Part I teaches Prolog, from terms and unification through search, lists, arithmetic, control, text, aggregation, higher-order predicates, the dynamic database and tabling, grammars, modules and tooling, to constraints. Each chapter ends with a diagnostic use of what it introduced, on a shared synthetic memory dump and a shared synthetic trace that recur throughout the book. Part II fixes the representation of diagnostic artifacts as terms and the discipline of writing a pattern as a predicate. Parts III and IV treat the memory analysis patterns and the trace and log analysis patterns, one section per pattern, with the recognizer, its transcript, and the limits of the formalization. Part V treats meta-interpreters and explanation, scale and performance, and integration with scripts, other languages, and debuggers. The appendices provide a reference to the built-in predicates used, the listings of the shared artifacts, a cross-reference from pattern names to predicates and chapters, and a glossary.

The executable Prolog examples and their transcripts were checked with SWI-Prolog version 9.0.4, and the output shown is the output it printed. The programs, the transcripts, the synthetic artifacts they run on, and the harness that replays every transcript against SWI-Prolog are published in the repository github.com/DumpAnalysis/PrologForSoftwareDiagnostics, so that the reader need not type them in and can rerun the check. Illustrative integration fragments, the HTTP service and the C embedding examples of Chapter 31, are labeled separately; the C examples were not compiled or executed, and the debugger session of Section 31.7 is reproduced from the validation log of the extension's repository, not from a run by the author of this edition. Where the output depends on the machine, such as timings, the varying part is elided. The reader who types the programs into the same system should see the same answers; the reader who uses a later version may see differences in the wording of error messages and in the printing of answers, which the text points out where they are known. The code is set with a fixed colouring: comments in italic brown, quoted atoms and

strings in green, variables in blue, numbers in purple, the control constructs `:-`, `-->`, `->`, `;`, `\+`, `!` and `|` in bold orange, and the toplevel prompt `?-` in bold. The name of a built-in or library predicate is in bold where it is called, `true` and `fail` included, and plain where it occurs in a data term, as `error` does in the catcher of a `catch/3`: the call positions were found by reading every clause and query with SWI-Prolog's own reader and following the goal arguments that the meta-predicate declarations of the called predicates name, so the emphasis marks calls, not names. Symbolic predicates such as `=` and `<` are never emphasized, and neither is a predicate the book defines itself, whatever it calls. In a transcript, an answer binding keeps the colours of its variables, numbers and quoted text, since it is a term, but carries no call emphasis, since it is data; a program's printed output is plain. The synthetic artifacts are invented; they are modelled on what a debugger shows for a Windows process and on what an application log contains, but no value in them was taken from a real system.

The book contains no exercises and no proofs. It is a reference as much as a course, and its index is meant to be used. Pattern names are written as in the catalogs, with capital initials and their variant in parentheses, and the predicates that recognize them are named after them. Where the book gives a pattern an operational definition narrower than the catalog's description, it says so, because the gap between a description and a recognizer is the subject of the book as much as the recognizers themselves.

About the Concept Author



Dmitry Vostokov is an internationally recognized expert, speaker, educator, scientist, inventor, and author. He founded the pattern-oriented software diagnostics, forensics, and prognostics discipline (Systematic Software Diagnostics) and the Software Diagnostics and Observability Institute (DA+TA: DumpAnalysis.org + TraceAnalysis.org). Vostokov has also authored over 50 books on software diagnostics, anomaly detection and analysis, software and memory forensics, root cause analysis and problem solving, memory dump analysis, debugging, software trace and log analysis, reverse engineering, and malware analysis. He has over 30 years of experience in software architecture, design, development, and maintenance in various industries, including leadership, technical, and people management roles. Dmitry founded OpenTask Iterative and Incremental Publishing (OpenTask.com) and Software Diagnostics Technology and Services (formerly Memory Dump Analysis Services), PatternDiagnostics.com. In his spare time, he explores Software Narratology, Software Pathology, Software Morphology, Quantum Software Diagnostics, and Pattern-Oriented AI. His interests include theoretical software diagnostics and its mathematical and computer science foundations; applying formal logic, semiotics, artificial intelligence, machine learning, and data mining to diagnostics and anomaly detection; software diagnostics engineering and diagnostics-driven development; and diagnostics workflow and interaction. Recent areas of interest also include functional programming, cloud-native computing, monitoring, observability, visualization, security, automation, applications of category theory to software diagnostics, development and big data, and artificial intelligence diagnostics.

Part I. Prolog from the Ground Up

Part I teaches Prolog as a working language, in the dialect of SWI-Prolog, to a reader who has programmed in other languages and has not programmed in this one. The order is the order of dependence: terms before unification, unification before clauses and search, search before the lists and arithmetic that most programs are made of, and those before control, text, aggregation, higher-order predicates, the dynamic database and tabling, grammars, modules, and constraints. Each chapter closes with a diagnostic use of what it introduced, applied to the shared memory dump `dump1.pl` or the shared trace `trace1.pl` that Appendix B lists in full, so that by the end of the part the reader has met most of the predicates of Parts III and IV in a first form.

Chapter 1. Logic Programming and Diagnostics

A memory dump or a software trace is data. Everything an analyst says about it, that a thread is blocked, that two threads block each other, that a gap between two messages is a discontinuity, is an inference from that data, made with a vocabulary of named patterns. This chapter presents Prolog as a notation for writing down such data as facts, such inferences as rules, and such statements as queries, and for having the inferences checked mechanically. It covers the three kinds of statement informally, the two ways of reading a Prolog program, the installation of SWI-Prolog on Windows, macOS, and Linux, the interactive toplevel, and the loading of program files, including what happens when a file contains a mistake.

The chapter uses facts, rules, conjunction, variables, and the two output predicates `write/1` and `nl/0`, and nothing else. Later chapters add the remaining constructs one layer at a time; Chapters 2 and 3 make precise what this chapter describes informally. The diagnostic use that closes the chapter is deliberately small: six facts about the threads of a process, queried by hand.

1.1 Diagnosis as Inference over Artifacts

Software diagnostics is the discipline of finding out what happened in a software system from the artifacts the system leaves behind: a **memory dump** (process, kernel, or complete), a **software trace** or **log**, and the output of the tools that read them. **Pattern-oriented software diagnostics** supplies a vocabulary for this activity. A **diagnostic analysis pattern** is a named, recurring configuration in an artifact together with its diagnostic meaning: **Spiking Thread** (page 335), **Wait Chain (Critical Sections)** (page 384), **Time Delta** (page 511), **Discontinuity** (page 513). The catalogs of memory analysis patterns and of trace and log analysis patterns name hundreds of such configurations. A pattern is identified in an artifact by an analyst; a concrete occurrence of a pattern is a **pattern instance**.

Three layers are involved, and keeping them apart is the first discipline of the subject. The first layer is the artifact itself: bytes in a dump file, lines in a log file. The second is what a tool derives from it. A debugger reconstructs a **stack trace** for each **thread** from saved registers and stack memory, lists the loaded **modules** from the loader's data structures, and reports which **critical sections** are owned by which threads. These derivations are usually right and sometimes wrong; a stack trace that passes through a module without a symbol file is a hypothesis, not an observation. The third layer is what the analyst concludes: this thread is blocked by that one; those two block each other; this is a deadlock. Each conclusion is an inference from second-layer statements, and each follows a definition, the definition of a pattern.

Prolog addresses the second and third layers. Second-layer statements become **facts**: thread 103 owns critical section A; thread 103 waits for critical section B. Pattern definitions become **rules**: a thread is blocked by another thread if it waits for a critical section that the other thread owns. Questions become **queries**: which threads are blocked, and by whom. Prolog answers a query by finding every way in which the facts and rules together support it. The analyst still decides what the

artifact means. What Prolog contributes is that the definitions are written down once, exactly, and applied to every fact without fatigue or omission.

Two cautions apply throughout the book. First, a Prolog program answers questions about the facts it has been given, not about the artifact. If the extraction from debugger output to facts dropped a thread, no rule can bring it back; Chapters 15 and 16 are about doing the extraction carefully. Second, the answer `false` to a query means that the query cannot be derived from the facts and rules, not that the corresponding statement about the system is false. A dump that does not record the owner of a lock does not prove that the lock was free.

1.2 Facts, Rules, and Queries

A Prolog **program** is a text file containing **clauses**. The simplest clause is a fact. The following file states who owns two critical sections, who waits for them, and what it means for one thread to be blocked by another. The file name ends in `.pl`, the conventional extension for Prolog source files.

```
% owner(CS, TID): critical section CS is
% owned by thread TID.
owner(cs_a, 103).
owner(cs_b, 104).

% waits_for(TID, CS): thread TID waits to
% enter critical section CS.
waits_for(101, cs_a).
waits_for(103, cs_b).
waits_for(104, cs_a).

% blocked_by(Waiter, Owner): Waiter waits
% for a critical section that Owner owns.
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, CS),
    owner(CS, Owner).
```

Lines beginning with `%` are comments and run to the end of the line; `/*` and `*/` delimit comments that span lines. Every clause ends with a period. A comment stating the meaning of each argument precedes the clauses of every predicate in this book.

The first five clauses are facts. `owner(cs_a, 103)` states that the relation `owner` holds between `cs_a` and `103`. The data items in it are **terms**: `cs_a` is an **atom**, a symbolic constant written in lowercase (or in single quotes when it would not otherwise parse, as in `'Buffer::Copy'`), and `103` is a **number**. The whole of `owner(cs_a, 103)` is also a term, a compound one; Chapter 2 treats terms systematically. The name `owner` together with the number of **arguments**, two, identifies the **predicate**; it is written `owner/2`, and the number is its **arity**. The set of clauses for a predicate is its **procedure**. When the clauses represent an artifact, as the facts here represent part of a dump, the set of loaded clauses is called a **knowledge base**; when the clauses are code, it is called a program.

A query is a statement submitted to Prolog for confirmation. At the interactive **toplevel**, described in Section 1.5, queries are typed after the prompt `?-` and end with a period. The transcripts in this book show the query on the first line and, below it, what SWI-Prolog prints.

```
?- owner(cs_a, 103).
true.

?- owner(cs_a, 104).
false.
```

SWI-Prolog prints `true.` when the query follows from the program and `false.` when it does not. The second answer deserves the caution of Section 1.1: `false.` reports that no clause supports `owner(cs_a, 104)`, nothing more.

A query becomes a question when it contains a **variable**. A variable is a name beginning with an uppercase letter or an underscore. It stands for a term that is not yet known, and Prolog answers the query by finding terms for which the query holds. Such an association of a variable with a term is a **binding**.

```
?- owner(cs_a, T).
T = 103.

?- owner(CS, TID).
CS = cs_a,
TID = 103 ;
CS = cs_b,
TID = 104.
```

The second query has two answers. SWI-Prolog prints the first and stops, because it has recorded a **choice point**, a note that untried alternatives remain, in this case the second `owner/2` fact. The user presses `;` to ask for the next answer or Enter to stop. In the transcripts of this book the `;` after an answer stands for the user pressing `;`. After the last answer SWI-Prolog prints a period, which indicates that no choice point remains. When a choice point remains but yields no further answer, pressing `;` produces `false.`; an example appears in Section 1.7. Chapter 4 explains choice points and the order in which answers appear.

The last clause of the file is a rule. A rule has a **head**, the part before `:-`, and a **body**, the part after it, and states that the head holds whenever the body holds. The symbol `:-` is read “if”. The body here is a **conjunction** of two **goals** separated by a comma, which is read “and”; the goals of a body are also called **subgoals**. The variable `CS` occurs in both goals and so links them: the critical section that `Waiter` waits for must be the same one that `Owner` owns. A variable’s scope is the clause in which it appears; `CS` in this rule has nothing to do with `CS` in a query, and two clauses may use the same variable name without any connection between them.

```
?- blocked_by(101, T).
T = 103.
```

```
?- blocked_by(T, 103).
T = 101 ;
T = 104.

?- blocked_by(A, B), blocked_by(B, A).
A = 103,
B = 104 ;
A = 104,
B = 103.
```

The last query is itself a conjunction. It asks for two threads each of which is blocked by the other, and finds the pair 103 and 104 twice, once in each order, because the condition is symmetric. This is the configuration the debugger command `!locks` lets the analyst see by hand and that Chapter 20 formalizes over the full dump schema as **Deadlock (Critical Sections)** (page 388). Here a definition of three lines and a query of one line replace a manual cross-check between two lists.

1.3 The Declarative and Procedural Readings

A clause can be read in two ways, and both readings are needed.

In the **declarative reading**, a clause is a logical statement about the relations it mentions. `blocked_by(Waiter, Owner) :- waits_for(Waiter, CS), owner(CS, Owner)` states: for all `Waiter`, `Owner`, and `CS`, if `Waiter` waits for `CS` and `Owner` owns `CS`, then `Waiter` is blocked by `Owner`. A fact is a statement with no condition. The program is the set of these statements, and the answers to a query are the instances of the query that follow logically from the program. Under this reading the order of clauses in the file and the order of goals in a body do not matter. This is the reading an analyst checks against a pattern's definition in the catalog, asking whether the rule says what the pattern means.

In the **procedural reading**, a clause is a recipe. To solve the goal `blocked_by(W, O)`, first solve `waits_for(W, CS)`, then solve `owner(CS, O)`. Goals in a body are solved from left to right; clauses for a predicate are tried from top to bottom; when a goal fails, Prolog goes back to the most recent choice point and tries the next alternative, a process called **backtracking**. Under this reading the orders matter: they determine which answer is found first, how much work is done, and whether the search terminates at all. The procedural reading is also what gives meaning to `write/1`, which prints a term when the goal containing it is solved, and to `nl/0`, which prints a newline.

```
?- blocked_by(T, 103), write(T), nl.
101
T = 101 ;
104
T = 104.
```

The output of `write/1` appears before the answer because the printing happens while the query is being solved, and the answer is printed afterwards by the toplevel. On backtracking into `blocked_by/2` for the second answer, `write/1` is executed again. `write/1` prints atoms without

quotes; `write('Thread 101')` prints `Thread 101`. Chapter 2 introduces `write_canonical/1`, which prints terms in a form that can be read back.

For the programs in Part I the two readings agree: every answer the procedure finds is a logical consequence of the program, and every logical consequence is found, provided the search terminates. Chapter 4 shows how the search is organized and where termination can fail; Chapter 7 introduces constructs, the cut and negation, under which the two readings part ways.

1.4 Installing SWI-Prolog

The implementation used in this book is **SWI-Prolog**, edition 9.0.4. Every transcript was produced with it. SWI-Prolog is free software, distributed from `https://www.swi-prolog.org`, whose download page offers a stable and a development release for each platform. Use the stable release unless you need a feature that only the development release has. Where this book says "Prolog prints" the behavior is common to Prolog systems; where it says "SWI-Prolog prints" the format is specific to SWI-Prolog and may differ in other systems or other editions.

1.4.1 Windows

Download the 64-bit installer from the download page and run it. The installer places the system under `C:\Program Files\swipl` by default. It installs two executables in the `bin` subdirectory: `swipl-win.exe`, which opens a console window of its own, and `swipl.exe`, which runs inside an existing command prompt or PowerShell window. The installer offers to associate the `.pl` extension with `swipl-win.exe`, so that double-clicking a program file starts SWI-Prolog with that file loaded, and offers to add the `bin` directory to the `PATH` environment variable, so that `swipl` can be started from any command prompt. Accept the second offer; the first conflicts with Perl, which uses the same extension, and should be accepted only if Perl is not in use.

1.4.2 macOS

The download page offers a disk image containing `SWI-Prolog.app`. Drag the application to the `Applications` folder. The application opens a console window; the command-line executable inside the bundle is `/Applications/SWI-Prolog.app/Contents/MacOS/swipl`, and a shell alias or a symbolic link on the `PATH` makes it available as `swipl`. Alternatively, Homebrew installs the command-line version with `brew install swi-prolog`, and MacPorts with `sudo port install swi-prolog`. Either package manager places `swipl` on the `PATH`.

1.4.3 Linux

Most distributions package SWI-Prolog. On Debian and Ubuntu the package is `swi-prolog` (`sudo apt install swi-prolog`); the packaged edition depends on the distribution release and may be older than 9.0.4. The SWI-Prolog project maintains a personal package archive for Ubuntu, `ppa:swi-`

`prolog/stable`, with current releases. On Fedora the package is named `pl`. A snap package, `swi-prolog`, is available for distributions that support snaps, and a Docker image, `swipl`, is available on Docker Hub. Building from source with CMake is documented on the project site and takes a few minutes on a current machine.

On every platform, the installation can be checked from a shell:

```
$ swipl --version
SWI-Prolog version 9.0.4 for x86_64-linux
```

The platform part of the output differs on Windows and macOS.

1.4.4 Editing Files

Prolog source files are plain text, encoded in UTF-8, and can be written with any editor. Editors with a Prolog mode (Emacs, Vim, Visual Studio Code with a Prolog extension) indent clauses and match parentheses, which helps with the nested terms of Chapter 2. SWI-Prolog itself provides `edit/1`: `edit(blocked_by)` opens the file that defines `blocked_by/2` at the right line, in the editor named by the `EDITOR` environment variable or, in the Windows console, in the built-in editor. The book does not depend on any particular editor.

1.5 The Toplevel

Starting `swipl` without arguments prints a banner (abridged here) and the prompt:

```
$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)
...
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
```

The loop that reads a query, solves it, and prints the answers is the toplevel. A program file is loaded with `consult/1`, whose argument is the file name as an atom; the extension `.pl` may be omitted, and a file name that is not a valid unquoted atom must be quoted, as in `consult('C:/analysis/locks')`. Forward slashes are accepted in file names on every platform. The shorthand `[locks]`, a list containing the file name, does the same thing and is what the manual calls **consulting** the file.

```
?- consult(locks).
true.

?- [locks].
true.
```

Loading an already loaded file again replaces its clauses with the new contents of the file. This is the normal development cycle: edit the file, reload it, repeat the query. `make/0` reloads every loaded file that has changed on disk since it was loaded, and so does the second step for all files at once.

```
?- make.
true.
```

In SWI-Prolog 9.0.4, loading a file in the toplevel prints nothing on success; earlier editions printed a message with the file name and the loading time, and Section 1.6 shows what is printed when the file contains a mistake.

`listing/1` prints the clauses currently loaded for a predicate, given its name or its name and arity, in a standard layout; in this edition the variable names are those of the source file. It is the quickest way to confirm that a file was loaded and that a clause says what it was meant to say.

```
?- listing(blocked_by).
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, CS),
    owner(CS, Owner).

true.

?- listing(owner/2).
owner(cs_a, 103).
owner(cs_b, 104).

true.
```

The built-in documentation is reached with `help/1` and `apropos/1`. `help(write)` prints the manual entry for every predicate named `write`; `help(write/1)` restricts it to the one with arity 1. `apropos(listing)` lists the predicates and manual sections whose name or summary mentions the word. Both predicates need the documentation that ships with the installation; on a minimal Linux package it may be in a separate package.

The toplevel is left with `halt/0`, entered as a query: `halt.` It ends the process with exit status 0. On Unix-like systems the end-of-file character (Control-D) at the prompt has the same effect.

Two command-line forms are used throughout the book. `swipl file.pl` loads the file and then enters the toplevel:

```
$ swipl locks.pl
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)
...
?- blocked_by(101, T).
T = 103.
```

For batch use, `swipl -g Goal -t halt file.pl` loads the file, runs `Goal`, and then runs `halt`, so that the process ends instead of entering the toplevel. The goal must be quoted for the shell. The

output is whatever the goal prints; the toplevel's `true.` and bindings are not printed because the toplevel never runs.

```
$ swipl -g "blocked_by(101, T), write(T), nl" -t halt locks.pl
103
```

The **exit status** of the process tells a caller how the goal ended. If the goal succeeds, `halt` runs and the status is 0. If the goal fails, SWI-Prolog prints a message naming the goal and the word `false`, and the status is 1. If the goal raises an error that nothing catches, SWI-Prolog prints the error and the status is 2. The two nonzero statuses are distinct in this edition (9.0.4), so a script can tell a goal that found nothing from a goal that could not run; Chapter 13 covers scripts, arguments, and exit codes in detail. In a Unix shell `echo $?` prints the status of the previous command; `cmd.exe` keeps it in `%ERRORLEVEL%` and PowerShell in `$LASTEXITCODE`. The second command below calls `holder/2`, a predicate the file does not define, and the error line is wrapped here because it is wider than the page:

```
$ swipl -g "blocked_by(102, T), write(T), nl" -t halt locks.pl
ERROR: -g blocked_by(102, T), write(T), nl: false
$ echo $?
1
$ swipl -g "holder(cs_a, T), write(T), nl" -t halt locks.pl
ERROR: -g holder(cs_a, T), write(T), nl: '<meta-call>'/1:
    Unknown procedure: holder/2
$ echo $?
2
```

The message for the failing goal says only that the conjunction had no solution; it does not say which goal failed, since failure is not an error. The message for the undefined predicate is the one Section 1.6 explains, prefixed by the goal that was running.

1.6 Errors and Warnings at Load Time

Three kinds of mistake are common when writing clauses, and SWI-Prolog reports each differently.

A **syntax error** is reported with the file name, the line, and the column at which the parser gave up, and the offending clause is skipped; the rest of the file is still loaded. The following file lacks a comma in its second clause.

```
owner(cs_a, 103).
owner(cs_b 104).
```

Loading it prints:

```
ERROR: /home/dv/bad.pl:2:11: Syntax error: Operator expected
```

"Operator expected" is the parser's way of saying that it found two terms, `cs_b` and `104`, with nothing between them. The same message appears when a query is mistyped at the toplevel; a

query that begins with an uppercase letter, for instance, is parsed as a variable followed by a term, and is rejected in the same words.

A goal for a predicate that does not exist is a runtime error, reported when the query is run rather than when the file is loaded:

```
?- holder(cs_a, T).
ERROR: Unknown procedure: holder/2 <...>
```

The full message ends with "(DWIM could not correct goal)": DWIM is the component that looks for a similarly named predicate, and when it finds one SWI-Prolog asks whether it should run the corrected goal instead. An unknown procedure is an error rather than a failure because it almost always indicates a typing mistake; Chapter 11 shows how to declare predicates that legitimately have no clauses.

The third kind is not an error but a **warning**, and the most frequent one concerns **singleton variables**. A variable that occurs exactly once in a clause cannot link anything to anything, so it is usually a misspelling of a variable that occurs elsewhere in the clause. The following file is loaded without error, but the clause does not do what was intended.

```
:- [locks].

% blocked_report(TID): print who blocks TID.
blocked_report(TID) :-
    blocked_by(TID, Owner),
    write(TID),
    write(' is blocked by '),
    write(Ownr),
    nl.
```

The first line is a **directive**: a goal, introduced by `:-` with no head, that is run when the file is loaded. Here it consults `locks.pl`, so that loading `report.pl` loads both files. Loading prints:

```
Warning: /home/dv/report.pl:4:
Warning:    Singleton variables: [Owner,Ownr]
```

The line number is that of the clause's head. The two names are reported together because each occurs once. Running the predicate shows the consequence: `Ownr` is never bound, and `write/1` prints an unbound variable as _ followed by a number that varies from run to run.

```
?- blocked_report(101).
101 is blocked by _<...>
true.
```

When a variable is meant to occur once, because the clause does not need the value, the warning is silenced by naming it `_`, the **anonymous variable**, or by giving it a name that begins with an underscore, such as `_Owner`, which documents what the argument is without using it. Chapter 2

returns to both forms. Here the fix is to spell the name consistently and reload with `make/0`. The corrected file:

```
:- [locks].

% blocked_report(TID): print who blocks TID.
blocked_report(TID) :-
    blocked_by(TID, Owner),
    write(TID),
    write(' is blocked by '),
    write(Owner),
    nl.

?- blocked_report(101).
101 is blocked by 103
true.

?- blocked_report(T).
101 is blocked by 103
T = 101 ;
103 is blocked by 104
T = 103 ;
104 is blocked by 103
T = 104.
```

A second common warning reads “Clauses of `thread_state/2` are not together in the source-file”. SWI-Prolog expects the clauses of a predicate to be contiguous, because a clause of some other predicate between them usually means that a clause was put in the wrong place. Files generated from debugger output often interleave predicates deliberately, as the shared data file `dump1.pl` of this book does for `frame/5`; the directive `:- discontiguous frame/5.` at the top of such a file declares the intent and suppresses the warning. Warnings do not stop the file from loading, but a file that loads with warnings should be treated as a file that has not been checked.

1.7 Diagnostic Use

The following knowledge base records, for six threads of a process, the state in which the debugger found each thread when the dump was written. Three states are distinguished: `running`, the thread was executing when the dump was taken; `waiting`, the thread was in a wait; `suspended`, the thread’s execution had been suspended by another thread. The shared data file used from Chapter 2 onwards refines `waiting` with a reason; here the state is a single atom.

```
% thread_state(TID, State): the state of thread
% TID at the time of the dump.  State is one of
% running, waiting, suspended.
thread_state(100, waiting).
thread_state(101, waiting).
thread_state(102, running).
thread_state(103, waiting).
thread_state(107, running).
thread_state(108, suspended).
```

Querying facts directly is the first thing to do with a knowledge base. With both arguments given, a query is a check; with one argument a variable, it is a lookup; with both variables, it enumerates the relation.

```
?- thread_state(102, S).
S = running.

?- thread_state(T, running).
T = 102 ;
T = 107.

?- thread_state(T, suspended).
T = 108.
```

Three patterns from the memory analysis catalog classify threads by this kind of information. **Active Thread** (page 337) names a thread that was executing code at the time of the dump rather than waiting; in the catalog it is contrasted with **Passive Thread** (page 334), a thread that waits for something and whose wait is, by itself, unremarkable: a message loop, a thread-pool idle wait, a timer. **Suspended Thread** names a thread whose execution was stopped by a suspend request from another thread, which is a different condition from waiting and is often a sign of deliberate intervention by the program or by a debugger. The three can be written as rules over the facts. The rules are predicates named after the patterns, which is the naming convention of this book for pattern recognizers.

```
active_thread(TID) :-
    thread_state(TID, running).

passive_thread(TID) :-
    thread_state(TID, waiting).

suspended_thread(TID) :-
    thread_state(TID, suspended).

?- active_thread(T).
T = 102 ;
T = 107.

?- passive_thread(T).
T = 100 ;
T = 101 ;
T = 103.

?- suspended_thread(108).
true.
```

Each rule restates one fact pattern under a pattern name, and that is already useful: `active_thread(T)` reads as the question the analyst asks, and a change in how the state is recorded, when the shared schema introduces `waiting(Reason)` in Chapter 2, is absorbed by changing the rule rather than every query. The rules also make visible what the facts do not support. `active_thread/1` says that a thread was executing, not that it was consuming an unusual share of processor time; **Spiking Thread** (page 335), which says that, needs the CPU times that Chapter 6

adds to the knowledge base. `passive_thread/1` says that a thread was waiting, not what it was waiting for; whether the wait is unremarkable or part of a wait chain is decided by the stack trace and the synchronization objects, which Chapters 4 and 20 bring in.

A predicate may have several clauses, each an alternative. Threads that are not executing are those that are waiting or suspended:

```
not_executing(TID) :-
    thread_state(TID, waiting).
not_executing(TID) :-
    thread_state(TID, suspended).

?- not_executing(T).
T = 100 ;
T = 101 ;
T = 103 ;
T = 108.

?- not_executing(103).
true ;
false.
```

The second transcript shows the behavior promised in Section 1.2. Thread 103 is found by the first clause, and SWI-Prolog prints `true` but not the period, because the second clause has not been tried and might also succeed. On `;` the second clause is tried, `thread_state(103, suspended)` has no supporting fact, and `false.` is printed. The query has succeeded exactly once; the trailing `false.` reports only that the remaining alternative led nowhere.

Rules may share variables between goals in the same way as `blocked_by/2`. Two threads are in the same state if some state is the state of both:

```
same_state(TID1, TID2) :-
    thread_state(TID1, State),
    thread_state(TID2, State).

?- same_state(102, T).
T = 102 ;
T = 107.

?- same_state(T, 108).
T = 108.
```

The answer `T = 102` to the first query is correct under the declarative reading, since thread 102 is in the same state as itself, and unwanted in practice. Excluding it requires saying that `TID1` and `TID2` are different, which needs the inequality of Chapter 3; the constructs of this chapter can only state what holds, never what does not.

Finally, a rule may describe a thread in words, using `write/1` and `nl/0` in its body:

```
describe(TID) :-
    thread_state(TID, State),
    write(TID),
    write(' is '),
    write(State),
    nl.

?- describe(108).
108 is suspended
true.

?- describe(109).
false.

?- active_thread(T), describe(T).
102 is running
T = 102 ;
107 is running
T = 107.
```

`describe(109)` fails without printing anything, because its first goal fails and the output goals are never reached; the knowledge base holds no fact about thread 109, and under the caution of Section 1.1 that is all the failure means. The last query combines a pattern predicate with the report, and prints a line per active thread as the toplevel is asked for each answer in turn. Printing a report for all threads without pressing `;` for each needs a construct from Chapter 7 (a failure-driven loop) or Chapter 9 (collecting all answers); formatting it in columns needs `format/2` from Chapter 6.

The shared file `dump1.pl`, introduced in Chapter 2 and used through the rest of the book, is this knowledge base at full size: a process, ten threads, their stack traces, modules, synchronization objects, heaps, and handles, all as facts, with pattern definitions as rules over them.

Chapter 2. Terms

Everything in Prolog is a term: the data in a knowledge base, the clauses of a program, the queries typed at the toplevel, and the answers printed back. This chapter describes the kinds of term, their written syntax, how SWI-Prolog prints them, the built-in predicates that test what kind of term a given term is, and the standard order in which terms are compared. It also records where SWI-Prolog departs from the ISO standard: strings, the representation of lists, the empty list, unbounded integers, and rational numbers.

The diagnostic motivation is the choice of representation. A stack frame, a loaded module, a memory address, a log message each have to become a term before any rule can speak about them, and the choice between an atom, a number, a string, and a compound term determines what the rules can later do cheaply and what they cannot do at all. The schema of the shared data files is presented from that point of view, and the chapter ends by inspecting the frame terms of `dump1.pl`. The equality predicates are left to Chapter 3; terms are shown here by printing them and tested with type tests and `compare/3`.

2.1 Atoms

An **atom** is a symbolic constant. It has a name and nothing else: no value, no fields, no type beyond being an atom. Three spellings produce atoms. A name beginning with a lowercase letter and continuing with letters, digits, and underscores is an atom: `ntdll`, `kernel32`, `cs_a`, `user_request`. A sequence of symbol characters is an atom: `+`, `:-`, `->`, `\==`, `=..`. Any text at all in single quotes is an atom: `'Buffer::Copy'`, `'NtWaitForMultipleObjects'`, `'Unknown'`, `'C:\\App\\app.cfg'`. The three special atoms `[]`, `{}`, and `!`, and the solo characters `;` and `,` (the last always quoted, `','`), complete the list.

```
?- atom(ntdll).
true.

?- atom('Buffer::Copy').
true.

?- atom(NtWaitForMultipleObjects).
false.
```

`atom/1` succeeds when its argument is an atom. The third query fails because a name beginning with an uppercase letter is a variable (Section 2.3); the debugger's function names must therefore be quoted when they become atoms. Inside a **quoted atom** a backslash begins an escape sequence: `\\` is a backslash, `\n` a newline, `\'` a quote; a quote may also be doubled, `'it''s'`.

`write/1` prints an atom's name and nothing else, so quoted and unquoted atoms look alike in its output. `write_canonical/1` prints every term in a form that can be read back as the same term, which means that it quotes an atom whenever the atom would not otherwise parse.

```
?- write('Buffer::Copy'), nl.
Buffer::Copy
true.

?- write_canonical('Buffer::Copy'), nl.
'Buffer::Copy'
true.

?- write_canonical('123'), nl.
'123'
true.
```

The last query shows that `'123'` is an atom whose name consists of digits; it is not the number 123. The toplevel prints answers in the same quoted form, which is why a binding to a function name appears as `F = 'Buffer::Copy'` in the transcripts of this book.

Atoms are **interned**: SWI-Prolog keeps one copy of each distinct name in an atom table, so two occurrences of the same atom are the same object, comparing two atoms for identity costs one pointer comparison however long their names, and a knowledge base with ten thousand frames in `ntdll` stores the module name once. SWI-Prolog reclaims atoms that are no longer referenced, but the table is still the wrong place for free text that is never compared as a whole; Section 2.7 gives message text a different representation. The rule of thumb for this book: a value that names something and is compared for identity is an atom; a value that is read, searched, or split is a string.

In SWI-Prolog, since version 7, the empty list `[]` is a reserved constant that is not an atom and is distinct from the atom `'[]'`; `atomic/1` (Section 2.8) accepts it, `atom/1` rejects it. ISO Prolog, and SWI-Prolog started with the `--traditional` option, treat `[]` as an atom. The empty curly braces `{}` are an ordinary atom in both.

```
?- atom([]).
false.

?- atom('[]').
true.

?- atom({}).
true.

?- atomic([]).
true.
```

2.2 Numbers

A **number** is an **integer** or a **float**; SWI-Prolog adds **rational numbers**. Integers are written in decimal, or with a radix prefix: `0x1a` is hexadecimal, `0o17` octal, `0b1010` binary. The form `0'c` denotes the **character code** of the character `c`, so `0'A` is 65 and `0'\n` is 10. SWI-Prolog also accepts an underscore between groups of digits, `0x7ffb_b2a0_0000` or `1_000_000`, which this book uses for addresses; **digit groups** are not part of the ISO syntax. Whatever the written form, the integer is the same term, and the toplevel and `write/1` print integers in decimal.

```
?- integer(0x1a).
true.

?- write(0x1a), nl.
26
true.

?- write(0x7ffb_b2a0_0000), nl.
140719010349056
true.

?- write(0'A), nl.
65
true.
```

The decimal output is a recurring nuisance in diagnostics, where addresses and exception codes are read in hexadecimal: the toplevel shows the exception code 3221225477 where the debugger showed `c0000005`. Chapter 6 introduces `format/2`, whose `~16r` directive prints an integer in hexadecimal; until then the prose names the hexadecimal values.

SWI-Prolog integers are **unbounded**: an integer literal of any length is accepted, and arithmetic (Chapter 6) never overflows. The flag `bounded` reports this; ISO leaves the choice to the implementation. For address arithmetic this removes a class of errors: a 64-bit address plus an offset is computed exactly, with no wraparound.

```
?- integer(18446744073709551616).
true.

?- current_prolog_flag(bounded, F).
F = false.
```

Floats are written with a decimal point and at least one digit on each side, optionally with an exponent: `1.0`, `0.212`, `1.0e3`, `1.5e-3`. They are IEEE double precision, and SWI-Prolog prints them with enough digits to read them back exactly and always with a decimal point or an exponent. The integer `1` and the float `1.0` are different terms; `float/1` accepts only the latter, `integer/1` only the former, and `number/1` both.

```
?- float(1.0).
true.

?- float(1).
false.

?- write(1.0e3), nl.
1000.0
true.
```

The timestamps of the shared trace are floats in seconds; the CPU times of the shared dump are integers in milliseconds. Both choices are deliberate: timestamps are compared and subtracted, where a float is adequate, while CPU times are summed, where an integer is exact.

SWI-Prolog 9 reads `1r3` as the rational number one third when the flag `rational_syntax` has its default value `compatibility`, and `rational/1` accepts it. Rationals are not part of ISO Prolog and play no further role in this book.

A minus sign immediately followed by digits is part of the number: `-5` is a negative integer. With a space between them, `- 5` is a compound term with the functor `-/1` and the argument `5` (Section 2.4); `write/1` keeps the space to preserve the distinction.

```
?- number(-5).
true.

?- number(- 5).
false.

?- write_canonical(- 5), nl.
-(5)
true.
```

2.3 Variables

A **variable** is a name beginning with an uppercase letter or an underscore: `TID`, `Module`, `X`, `_Offset`, `_`. A variable stands for a term that is not yet known. It is **unbound** until unification (Chapter 3) binds it to a term, after which every occurrence of the variable in its clause stands for that term. `var/1` succeeds when its argument is an unbound variable and `nonvar/1` when it is not; neither binds anything.

```
?- var(X).
true.

?- nonvar(X).
false.

?- var(ntdll).
false.
```

The scope of a variable name is the clause or query in which it occurs. `TID` in one clause and `TID` in the next are different variables. Within one clause, every occurrence of the same name is the same variable; this is how a variable in a head is connected to the goals of the body, and how two goals of a body are connected to each other.

The **anonymous variable** `_` is the exception: each occurrence of `_` is a distinct variable, so `frame(_, 0, ntdll, _, _)` has three unrelated anonymous arguments. It is used for an argument whose value is not needed. A name beginning with an underscore followed by letters, such as `_Offset`, is an ordinary variable, shared between its occurrences, but it signals that a single occurrence is intended; the singleton warning of Chapter 1 is not issued for it, and a warning is issued instead if it occurs twice. This book writes `_` for an irrelevant argument and `_Name` when the argument's role is worth stating.

```
% module_of(Frame, Module): Module is the module
% argument of the frame term Frame.
module_of(frame(_TID, _N, Module, _F, _Off),
          Module).

?- module_of(frame(109, 0, dbclient,
       'Buffer::Copy', 26), M).
M = dbclient.

?- module_of(thread(100, 4120), M).
false.
```

This fact selects one argument out of a five-argument term by shape alone. How the two frame terms are matched is the subject of Chapter 3; here the four underscore-named variables document which positions are ignored.

`write/1` prints an unbound variable as an underscore followed by a number that varies from run to run, as the singleton example of Chapter 1 showed. `write_canonical/1` instead names the variables of a term `A`, `B`, `C`, … in order of appearance and prints a variable that occurs only once as `_`; this is SWI-Prolog behavior, not ISO.

```
?- write_canonical(f(X, Y, X)), nl.
f(A,_,A)
true.
```

2.4 Compound Terms

A **compound term** consists of a name, which is an atom, and one or more **arguments**, which are terms. It is written as the name followed by the arguments in parentheses, separated by commas, with no space between the name and the opening parenthesis: `frame(109, 0, dbclient, 'Buffer::Copy', 26)`, `waiting(user_request)`, `point(1, 2)`. The name together with the number of arguments is the **functor**, written `frame/5`, `waiting/1`, `point/2`, and the number of arguments is the **arity**. The functor of the term as a whole, as opposed to the functors of terms nested inside it, is its **principal functor**. Terms with the same name and different arities, `thread_state/2` and `thread_state/3`, are unrelated.

Because arguments are terms, compound terms nest to any depth: `thread_state(109, waiting(exception))` has the principal functor `thread_state/2` and a second argument that is itself a compound term. The shared dump schema uses this nesting in exactly one place, the state of a thread, where `waiting/1` carries the reason for the wait; Chapter 1 flattened it to the atom `waiting`.

```
?- compound(waiting(user_request)).
true.

?- compound(ntdll).
false.
```

```
?- write_canonical(waiting(user_request)), nl.
waiting(user_request)
true.
```

`compound/1` succeeds for compound terms only. A term that contains no variables anywhere is **ground**; `ground/1` tests this, and every fact in a knowledge base derived from a dump is ground, since a dump records values, not unknowns.

```
?- ground(frame(109, 0, dbclient,
       'Buffer::Copy', 26)).
true.

?- ground(frame(_, 0, ntdll, _, _)).
false.
```

Clauses are compound terms too. The rule `blocked_by(W, O) :- waits_for(W, CS), owner(CS, O)` is a term with principal functor `:-/2`, whose first argument is the head and whose second argument is a term with principal functor `','/2`. A fact is a term whose principal functor is the predicate's; a query is a term. This uniformity is what allows a program to be printed by `listing/1` and, from Chapter 29 on, interpreted by another Prolog program.

2.5 Operators and Canonical Form

The clause above is not written as `:-(blocked_by(W, O), ','(...))`, because `:-` and `,` are **operators**: atoms that may be written between, before, or after their arguments. Operator notation is syntax only. The term written `1 + 2 * 3` is the same term as `+(1, *(2, 3))`, and nothing about it is arithmetic until Chapter 6 asks for it to be evaluated. The form with the name in front of parenthesized arguments is the **canonical form** of a term, and `write_canonical/1` prints it.

```
?- write_canonical(1 + 2 * 3), nl.
+(1,*(2,3))
true.

?- write_canonical((1 + 2) * 3), nl.
*(+(1,2),3)
true.

?- write_canonical((a :- b, c)), nl.
:-(a,','(b,c))
true.
```

Each operator has a **priority** from 1 to 1200 and a type. The priority settles which operator is the principal functor when several appear without parentheses: `*` at 400 binds tighter than `+` at 500, so `1 + 2 * 3` has `+` at the top. The type settles how a sequence of the same operator groups: `yfx` to the left, so `1 - 2 - 3` is `(1 - 2) - 3`; `xfy` to the right, so `a, b, c` is `a, (b, c)`; `xfx` not at all, so `a = b = c` is a syntax error. Parentheses override both, and a term whose principal functor has a priority above 999 must be parenthesized as an argument, which is why `(a :- b, c)` above carries parentheses. The standard operators used in this book:

Priority	Type	Operators
1200	xfx	:-, -->
1200	fx	:- (directive)
1100	xfy	;
1050	xfy	->
1000	xfy	,
900	fy	\+
700	xfx	=, \=, ==, \==, is, <, >, @<
500	yfx	+, -
400	yfx	*, /, //, mod
200	xfx	**
200	xfy	^
200	fy	- (sign)

`write/1` and the toplevel print with operator notation, so a term built in canonical form is printed back with operators. `display/1`, a traditional predicate outside ISO, prints with quotes and without operators like `write_canonical/1`, and also expands list notation (Section 2.6).

```
?- write(+(1, *(2, 3))), nl.
1+2*3
true.

?- write(*(+(1, 2), 3)), nl.
(1+2)*3
true.
```

Programs may declare their own operators with `op/3`; this book does not.

2.6 Lists

A **list** is written as a comma-separated sequence of terms in square brackets, `[ntdll, kernelbase, kernel32]`, and the empty list as `[]`. The notation is a convenience for a chain of compound terms: a non-empty list is a term with two arguments, the first element and the list of the remaining elements, ending in `[]`. In SWI-Prolog, since version 7, the functor of this term is `'[|]'/2`[[‘[|]’/2]]; in ISO Prolog and in traditional systems it is `'.'/2`. `display/1` shows the chain.

```
?- write_canonical([1, 2, 3]), nl.
[1,2,3]
true.

?- display([1, 2, 3]), nl.
'[|]'(1,'[|]'(2,'[|]'(3,[])))
true.

?- write_canonical('[|]'(1, [])), nl.
[1]
true.
```

The vertical bar separates the leading elements from the rest: `[H | T]` is a list whose first element is `H` and whose remaining elements form the list `T`, and `[a, b | T]` has two leading elements. A list whose tail is an unbound variable is a **partial list**. A list is a compound term, so `compound([1])` succeeds, while `[]` is the reserved constant of Section 2.1, so `compound([])` fails. `is_list/1` succeeds only for a proper list: a chain of `'[|]'/2` cells that ends in `[]`.

```
?- compound([1]).
true.

?- compound([]).
false.

?- is_list([1, 2, 3]).
true.

?- is_list([1 | T]).
false.

?- write_canonical([a, b | T]), nl.
[a,b|_]
true.
```

In SWI-Prolog 7 and later the functor `'.'/2` is reserved for the dict notation of Chapter 10, so `'.'(1, [])` does not construct a list and raises an error when evaluated; code from traditional systems that builds lists with `'.'/2` must use the bracket notation, which is portable. Chapter 5 is about computing with lists; here they are only a term shape, and one the shared schema does not use, since every relation of the shared files is a set of facts.

2.7 Strings, Code Lists, and Char Lists

Text has four representations in SWI-Prolog. An atom, `'Query end'`, is an interned name. A **string** object, `"Query end"`, is a sequence of characters that is not interned, compared by content, and compact; it is a term kind of its own, tested by `string/1`, and is specific to SWI-Prolog. A **code list**, `[0'Q, 0'u, ...]` or equivalently `[81, 117, ...]`, is an ordinary list of character codes. A **char list**, `['Q', u, e, r, y]`, is a list of single-character atoms. ISO Prolog has the last three; a text in double quotes denotes a code list in ISO and in SWI-Prolog's `--traditional` mode.

```
?- string("Query end").
true.

?- atom("Query end").
false.

?- write("Query end"), nl.
Query end
true.

?- write_canonical("Query end"), nl.
"Query end"
true.
```

```
?- write_canonical(`abc`), nl.
[97,98,99]
true.
```

What double quotes denote is governed by the flag `double_quotes`, whose values are `string`, `codes`, `chars`, and `atom`; the default in SWI-Prolog 7 and later is `string`. Backquotes denote a code list under the default of the companion flag `back_quotes`. Both flags, when set in a source file with `set_prolog_flag/2`, affect that file only.

```
?- current_prolog_flag(double_quotes, F).
F = string.

?- is_list("abc").
false.
```

The last query shows the trap for code written against ISO or older textbooks: `"abc"` is not a list in SWI-Prolog, and list predicates fail on it or raise errors. Chapter 8 introduces the predicates that convert between the four representations and that search, split, and compare strings, and Chapter 12 uses code lists where grammars require them.

For the shared trace the text of a message is a string: `msg(14, 1.048, 4120, 101, dbclient, "Query end: 1 row, 44 ms")`. A message text is rarely compared as a whole, is often searched for a substring or split at delimiters, and in a large trace runs to millions of distinct values; all three argue against atoms and for strings. The component name `dbclient`, which is compared as a whole in every query and takes a handful of values, is an atom for the mirror-image reasons.

2.8 Type Tests

The predicates introduced so far that test the kind of a term are collected here with the remaining ones. Each takes one argument, succeeds or fails according to the kind of term the argument is at the moment of the call, binds nothing, and never raises an error. They are **type tests**.

Predicate	Succeeds for
`var/1`	an unbound variable
`nonvar/1`	anything else
`atom/1`	an atom (`[]` excluded)
`number/1`	an integer, float, or rational
`integer/1`	an integer
`float/1`	a float
`rational/1`	a rational, integers included
`string/1`	a string
`atomic/1`	an atom, number, string, or `[]`
`compound/1`	a compound term
`callable/1`	an atom or a compound term
`is_list/1`	a proper list

Predicate	Succeeds for
`ground/1`	a term without variables

`atomic/1` accepts every term that is neither a variable nor compound. `callable/1` accepts the terms that could be goals: atoms and compound terms, but not `[]`, numbers, or strings. `rational/1` includes integers because every integer is a rational.

```
?- atomic("abc").
true.

?- atomic(X).
false.

?- callable(foo(1)).
true.

?- callable([]).
false.
```

Type tests describe the current state of a term, not its eventual value. `atom(X)` with `X` unbound fails rather than waiting to see what `X` becomes, so a test must come after the goal that binds the argument it tests. Chapter 7 shows how `must_be/2` turns a failed type expectation into an error with a message, the right response when a fact violates the schema.

2.9 The Standard Order of Terms

Any two terms can be compared, whether or not they are of the same kind, in the **standard order of terms**. `compare/3` does so: `compare(Order, A, B)` binds `Order` to one of the atoms `<`, `=`, and `>`. The predicates `@</2`, `@>/2`, `@=</2`, and `@>=/2` test the order directly. In SWI-Prolog the order is:

1. Variables, ordered by address, before everything else.
2. Numbers, compared by value; when an integer and a float are equal in value, the float comes first.
3. Strings, compared by character code, position by position.
4. Atoms, likewise by character code.
5. Compound terms, first by arity, then by name, then by arguments from left to right.

ISO Prolog specifies the same order without strings, which it does not have. In this edition the reserved constant `[]` sorts after all strings and before all atoms.

```
?- compare(O, 1, a).
O = (<).

?- compare(O, a, "a").
O = (>).
```

```
?- compare(O, 1.0, 1).
O = (<).

?- compare(O, f(a, b), g(a)).
O = (>).

?- compare(O, f(a), g(a)).
O = (<).
```

The toplevel prints the answer `(<)` in parentheses because `<` is an operator and would otherwise be ambiguous when read back. Arity before name means that `f(a, b)` comes after `g(a)` despite `f` preceding `g`; the comparison of arguments from left to right means that two frame terms of the same thread are ordered by their frame number, and two frames of different threads by thread identifier first.

```
?- compare(O, frame(101, 2, ntdll, f, 0),
       frame(101, 10, ntdll, f, 0)).
O = (<).

?- compare(O, frame(102, 0, ntdll, f, 0),
       frame(101, 9, ntdll, f, 0)).
O = (>).
```

This is the order in which a stack trace is read, and the sorting predicates of Chapter 5 produce it without further instructions when frames are kept as `frame/5` terms with integer arguments. The qualification matters: numbers are ordered by value, atoms by character code, so frame numbers stored as atoms would order `'10'` before `'9'`.

```
?- compare(O, 10, 9).
O = (>).

?- compare(O, '10', '9').
O = (<).
```

Comparison of variables by address is well defined at the moment of comparison but not stable, since binding a variable changes where it falls in the order; sorting terms that contain unbound variables is rarely meaningful.

2.10 Artifacts as Terms

The shared files encode two artifacts, a process memory dump and a software trace, as facts. Each predicate’s arguments were chosen by the rules this chapter has set out. The following clauses are illustrative copies of facts from those files.

```
module(dbclient, 0x7ffb_8d10_0000, 0x6e000).
frame(109, 0, dbclient, 'Buffer::Copy', 0x1a).
thread_state(109, waiting(exception)).
exception_record(109, 0xc0000005,
                 0x7ffb_8d10_31aa).
msg(69, 14.002, 4120, 109, dbclient,
    "Error: buffer copy failed, code=0x3e6").
```

A module name is a lowercase atom, `dbclient`, so that it needs no quotes in a query; the debugger's `dbclient.dll` is normalized to it. A base address, a size, an offset, and an exception code are integers, written in hexadecimal with digit groups, so that they can be added and compared in Chapter 6 and printed in hexadecimal when needed; as atoms they could be neither added nor ranged over. A function name is an atom spelled exactly as the debugger showed it, quotes and all, because the analyst will search for `'Buffer::Copy'` with the spelling found in the stack trace, and because quoting preserves the C++ scope operator and the template brackets of names such as `'thread_start<...>'`; an unresolved symbol is the atom `'Unknown'`, which is what the debugger prints. A thread state is an atom or, when it carries a reason, a compound term, so that a rule can match `waiting(_)` to mean any wait or `waiting(exception)` to mean one wait in particular. Message text is a string.

Two things are deliberately absent. Nothing is a variable: a knowledge base built from a dump records what was observed, and an unknown value is recorded as an explicit atom or by the absence of a fact; Chapter 15 discusses the difference. Nothing is a list: each frame is its own fact rather than an element of a per-thread list, so that a query reaches any frame by thread and number directly and the knowledge base grows by appending facts. Lists are built from facts when a computation needs them, as Chapter 5 does for stack traces.

2.11 Diagnostic Use

The shared dump is loaded by consulting `dump1.pl`, which the following program block does with the list shorthand of Chapter 1 as a directive.

```
:- [dump1].
```

The `frame/5` relation records one fact per frame of every thread's stack trace: `frame(TID, N, Module, Function, Offset)`, with `N` counting from 0 at the top of the stack, the frame executing or waiting when the dump was written. The top frame of the faulting thread, 109, and the one frame of thread 101 that the debugger could not resolve:

```
?- frame(109, 0, M, F, O).
M = dbclient,
F = 'Buffer::Copy',
O = 26.

?- frame(101, 4, M, F, O).
M = cachelib,
F = 'Unknown',
O = 10768.
```

The module is an unquoted atom; the function is a quoted atom, printed with its quotes by the toplevel; the offsets, written `0x1a` and `0x2a10` in the file, appear as the decimal integers 26 and 10768. `write_canonical/1` prints a frame term from the knowledge base in the form it had in the file except for this.

```
?- frame(101, 0, M, F, O),
   write_canonical(frame(101, 0, M, F, O)), nl.
frame(101,0,ntdll,'NtWaitForAlertByThreadId',20)
M = ntdll,
F = 'NtWaitForAlertByThreadId',
O = 20.
```

A knowledge base produced by a parser from debugger output can violate its own schema: a frame number read as an atom, an offset left as text, a function name read as a string. The type tests make the schema checkable. The following helper succeeds for a frame whose arguments have the prescribed kinds.

```
% well_formed_frame(TID, N): frame N of thread TID
% has the argument types the schema prescribes.
well_formed_frame(TID, N) :-
    frame(TID, N, Module, Function, Offset),
    integer(TID),
    integer(N),
    atom(Module),
    atom(Function),
    integer(Offset).

?- well_formed_frame(109, 0).
true.

?- well_formed_frame(101, 4).
true.

?- frame(T, N, _, F, _), string(F).
false.
```

The last query asks whether any frame has a string where an atom is required; none does. The complementary question, which frames are not well formed, requires negation and is deferred to Chapter 7.

The atom `'Unknown'` is the representation of a frame whose function the debugger could not name, and querying for it is a one-line matter:

```
?- frame(T, N, M, 'Unknown', _).
T = 101,
N = 4,
M = cachelib.
```

Thread 101 has such a frame, in `cachelib`. The reason is recorded in `module_info/3`, whose `symbols` key records what symbol information the debugger had for each module: `pdb` for a full symbol file, `export` for the export table only, `none` for nothing.

```
?- module_info(cachelib, symbols, S).
S = none ;
false.

?- module_info(M, symbols, none).
M = cachelib ;
M = thirdparty_hook.
```

The `false.` after the first answer is the behavior described in Section 1.2: a later fact about `cachelib`, the one with the key `company`, was still a candidate when the `symbols` fact was found, so a choice point remained; on `;` the candidate was tried and did not match. Chapter 13 explains how SWI-Prolog selects candidate clauses.

A module for which the debugger had no symbols is an instance of the **No Component Symbols** (page 431) pattern of the memory analysis catalog: its frames show offsets from the module base rather than function names, and the analyst must treat each of them as a gap in the stack trace. The predicate restates the fact under the pattern name:

```
no_component_symbols(Module) :-
    module_info(Module, symbols, none).


?- no_component_symbols(M).
M = cachelib ;
M = thirdparty_hook.

?- no_component_symbols(ntdll).
false.
```

A module with export symbols only, `dbclient` in this dump, is a borderline case: the debugger names functions by the nearest preceding exported symbol, which may be wrong by an arbitrary distance. Whether to count it as an instance is a judgment Chapter 22 takes up; a rule that includes `export` would be a second clause of the predicate above.

The companion key `company` records the vendor read from the module's version resource, with the atom `'Unknown'` when there was none. A module with no vendor, no symbols, and an unfamiliar name is an instance of **Unknown Component** (page 430): a module about which the dump itself says nothing, and which the analyst must identify by other means before deciding whether it matters.

```
unknown_component(Module) :-
    module_info(Module, company, 'Unknown').


?- unknown_component(M).
M = thirdparty_hook.

?- module_info(thirdparty_hook, K, V).
K = version,
V = '1.0.0.1' ;
K = symbols,
V = none ;
K = company,
V = 'Unknown'.
```

The last query lists everything the knowledge base holds about the module: a version, no symbols, no vendor. Both predicates in this section are a single fact pattern under a catalog name, and both depend on representation decisions made before any rule was written: that `'Unknown'` is one atom, spelled one way, wherever an unresolved name is recorded, and that the symbol kinds are three fixed atoms rather than free text. Chapter 3 shows what the matching behind these queries does, and Chapter 15 shows how facts are produced so that the decisions hold throughout a knowledge base.

Chapter 3. Unification

Every query in the first two chapters was answered by the same operation. When `frame(109, 0, M, F, O)` met the fact `frame(109, 0, dbclient, 'Buffer::Copy', 26)`, the two terms were compared position by position, the variables of the query were bound to the corresponding parts of the fact, and the bindings were reported as the answer. That operation is **unification**, and it is the only mechanism Prolog has for passing values, matching data against patterns, and constructing terms. This chapter defines it exactly, distinguishes it from the weaker notion of two terms being identical, and introduces the built-in predicates that unify, that compare without unifying, and that inspect and copy terms with variables in them: `=/2`, `\=/2`, `==/2`, `\==/2`, `unify_with_occurs_check/2`, `?=/2`, `subsumes_term/2`, `term_variables/2`, and `copy_term/2`.

The diagnostic motivation is the **frame template**: a frame term with variables in some positions and constants in others, such as `frame(_, 0, ntdll, F, _)`, which reads “a top frame in `ntdll`, whatever the thread, whatever the offset, and tell me the function”. Querying a knowledge base with such a template is unification applied to every frame fact, and the templates an analyst uses by eye when reading `~*k` output, the top frame in a wait function, a frame in the critical-section entry path, two adjacent frames with the same function, become one-line queries. The chapter ends by writing them against `dump1.pl`.

3.1 Identity and Unifiability

Two terms are **identical** when they are the same atom, the same number of the same kind (the integer `1` and the float `1.0` are not identical), the same string, the same variable, or compound terms with the same name, the same arity, and pairwise identical arguments. `==/2` tests identity and `\==/2` its negation. Neither binds a variable.

```
?- a == a.
true.

?- 1 == 1.0.
false.

?- 'Buffer::Copy' == 'Buffer::Copy'.
true.

?- "abc" == abc.
false.

?- X == Y.
false.

?- X == X.
true.
```

Two terms are **unifiable** when there is some way of binding their variables that makes them identical. `=/2` tests unifiability and, when it succeeds, performs the bindings; `\=/2` succeeds when

the two terms are not unifiable and binds nothing. Identical terms are unifiable with no bindings at all, so `a = a` succeeds exactly as `a == a` does. The two notions part ways as soon as a variable is involved.

```
?- a = b.
false.

?- 1 = 1.0.
false.

?- f(X, b) = f(a, Y).
X = a,
Y = b.

?- X = Y.
X = Y.

?- X = Y, X == Y.
X = Y.
```

The third query is the typical case: the two terms are not identical, but binding `X` to `a` and `Y` to `b` makes them identical, and the toplevel reports exactly those bindings. The fourth binds two unbound variables to each other; from then on they are **aliased**, a single variable with two names, which the toplevel reports as `X = Y`. The last query shows the consequence: after `X = Y`, the two are identical.

The negated predicates differ in the same way, and the difference is a frequent source of error. `\==` asks whether two terms are different now; `\=` asks whether they can never be made the same.

```
?- f(X) \== f(a).
true.

?- f(X) \= f(a).
false.

?- X \== a.
true.

?- X \= a.
false.
```

`f(X) \== f(a)` succeeds because an unbound variable is not identical to `a`. `f(X) \= f(a)` fails because `X` could be bound to `a`. Which predicate is right depends on what the question is. To ask whether a thread identifier found in one fact is a different thread from one found in another, with both already bound to integers, either predicate gives the same answer, and `\==` is the conventional choice because it states a test on the terms as they are. To ask whether an unbound variable is different from something, neither predicate can answer: `\==` says yes because the variable is still a variable, `\=` says no because it might yet become the thing. The sound way to constrain a variable to differ from a term it has not yet been bound to is `dif/2`, which Chapter 14 introduces; until then every inequality in this book is written after the goals that bind its arguments.

The SWI-Prolog manual writes the argument **modes** of these predicates as `?Term1 = ?Term2` and `@Term1 == @Term2`. In this notation, used throughout the manual and this book, `+` marks an argument that must be instantiated when the predicate is called, `-` one that is unbound at the call and bound by it, `?` one that may be either, and `@` one that the predicate inspects without binding or further instantiating it. `=/2` accepts anything in either position and may bind both; `==/2` binds nothing.

3.2 How Unification Works

Unification of two terms proceeds by cases.

1. Two atoms, two numbers, or two strings unify when they are identical; otherwise they do not.
2. An unbound variable unifies with any term, and is bound to it. Two unbound variables unify by becoming aliased.
3. Two compound terms unify when they have the same name and the same arity and their arguments unify pairwise, from left to right, with a binding made for an earlier pair holding for the later pairs.
4. In every other case, an atom with a number, an atom with a compound term, compound terms with different functors, the two terms do not unify.

The outcome of a successful unification is a **substitution**: a set of **bindings** of variables to terms. Of all the substitutions that would make two terms identical, unification produces the **most general unifier**, the one that binds no more than necessary; every other unifier of the two terms is obtained from it by binding further variables. `f(X, Y) = f(Y, a)` has the unifier `X = a, Y = a`, and the toplevel reports it as an aliasing followed by one binding.

```
?- f(X, Y) = f(Y, a).
X = Y, Y = a.

?- f(X, X) = f(a, b).
false.

?- f(X) = f(X, Y).
false.

?- foo(X) = bar(X).
false.

?- f(a, g(X)) = f(Y, g(b)).
X = b,
Y = a.
```

The second query fails by case 3: the first pair of arguments binds `X` to `a`, and the second pair then asks `a` to unify with `b`. Whether unification succeeds, and what the most general unifier is, does not depend on the order in which pairs are processed; the left-to-right order only determines which pair is found to disagree first. The third query fails on arity, the fourth on name, and the fifth shows unification descending into nested compound terms.

Unification builds terms as readily as it takes them apart, and it never evaluates anything. The query `X = 1 + 2` binds `X` to the compound term `+(1, 2)`; nothing adds the numbers. Arithmetic is a separate matter, treated in Chapter 6, and the distinction between a term that denotes an expression and the value of the expression is one that Prolog never blurs.

```
?- X = 1 + 2.
X = 1+2.

?- X = f(Y), Y = 1.
X = f(1),
Y = 1.
```

The second query shows that a binding made later reaches into terms built earlier: `X` was bound to `f(Y)` while `Y` was unbound, and binding `Y` afterwards makes `X` stand for `f(1)`. A variable is not a slot that is copied into; it is shared by every term it occurs in.

The same operation runs every time a goal is solved against a clause. Before the goal `module_of(frame(109, 0, dbclient, 'Buffer::Copy', 26), M)` of Chapter 2 is compared with the head `module_of(frame(_TID, _N, Module, _F, _Off), Module)`, the clause is **renamed**: every variable in it is replaced by a fresh one that occurs nowhere else, so that solving the same clause twice, or against itself, cannot confuse the variables of the two uses. The goal and the renamed head are then unified. If they do not unify, the clause is not applicable. If they do, the bindings hold for the body of the clause, which is solved next, and for the rest of the query. Here the first arguments unify by case 3, binding the fresh variables to 109, 0, `dbclient`, `'Buffer::Copy'`, and 26, and the second arguments then unify `M` with the fresh variable already bound to `dbclient`. Chapter 4 places this step in the full account of how a query is solved.

3.3 Unification as Pattern Matching

A term with variables in it is a pattern, and **pattern matching** by unifying it with a term without variables either fails, because the shapes disagree, or succeeds and extracts the parts of the second term that correspond to the variables. The frame template of the introduction does this to a frame term:

```
?- frame(T, 0, ntdll, F, _) =
   frame(100, 0, ntdll, 'NtDelayExecution', 20).
T = 100,
F = 'NtDelayExecution'.

?- frame(T, 0, ntdll, F, _) =
   frame(100, 0, kernelbase, 'SleepEx', 158).
false.
```

The template fixes two positions, the frame number and the module, leaves two to be reported, the thread and the function, and ignores one, the offset, with an anonymous variable. The second frame has the wrong module, and the attempt fails at that position. Nothing else is involved: no comparison function, no accessor, no declaration of which arguments are input and which are output. The same

template, unified with a frame term containing variables, would fill in whatever it could and leave the rest; the direction of information flow is decided at run time by which side has the variables.

The same mechanism constructs terms incrementally. A term with unbound variables in it is a **partial structure**, and binding its variables later completes it.

```
?- X = frame(_, 0, ntdll, _, _).
X = frame(_, 0, ntdll, _, _).

?- X = frame(T, N, M, F, O), T = 101, N = 0.
X = frame(101, 0, M, F, O),
T = 101,
N = 0.
```

The toplevel prints the unbound positions of the first answer as `_`, since each was an anonymous variable with no other occurrence to relate it to, and prints the second with the query's own variable names. Lists are compound terms (Section 2.6), so the same applies to them; `[H | T]` unified with a list binds `H` to the first element and `T` to the rest, and this is the whole basis of list processing in Chapter 5.

```
?- [H | T] = [ntdll, kernelbase, kernel32].
H = ntdll,
T = [kernelbase, kernel32].

?- [a, b | T] = [a, b].
T = [].
```

Pattern matching by unification has one property that distinguishes it from pattern matching in most programming languages: a variable that occurs twice in a pattern demands the same term in both places. `f(X, X)` matches `f(a, a)` and not `f(a, b)`. The frame template `frame(T, 0, M, F, _), frame(T, 1, M, F, _)` in Section 3.7 uses this to say "two frames of the same thread, adjacent, in the same module, with the same function", with no explicit equality test at all.

3.4 The Occurs Check and Cyclic Terms

Case 2 of Section 3.2 has a gap. Unifying `X` with `f(X)` asks for a term that is identical to `f` applied to itself. No finite term satisfies this, and the ISO standard leaves the result of such a unification undefined. Detecting the case, the **occurs check**, costs time proportional to the size of the term on every variable binding, and the case does not arise in correct programs that manipulate finite data.

SWI-Prolog, like most systems, omits the check by default and lets the binding happen. The result is a **cyclic term**, a finite graph that unfolds to the infinite term `f(f(f(...)))`. SWI-Prolog supports such terms consistently: they can be unified, compared, copied, and printed, and the toplevel prints `X = f(X)` to show one. `cyclic_term/1` and `acyclic_term/1` test for the property.

```
?- X = f(X).
X = f(X).

?- X = f(X), cyclic_term(X).
X = f(X).

?- acyclic_term(f(a)).
true.
```

Three provisions exist for programs that must not create cyclic terms. `unify_with_occurs_check/2` is the ISO predicate that unifies with the check and fails where plain unification would build a cycle. The flag `occurs_check`, set with `set_prolog_flag/2`, changes the behavior of every unification in the system, head unification included: its default `false` omits the check, `true` performs it and fails, and `error` performs it and raises an exception that names the offending variable and term. The first is the choice for a program that needs the check in one place; the flag is the choice during development, when a cycle is almost certainly a bug and the cost is irrelevant.

```
?- unify_with_occurs_check(X, f(X)).
false.

?- unify_with_occurs_check(f(X, b), f(a, Y)).
X = a,
Y = b.

?- set_prolog_flag(occurs_check, true).
true.

?- X = f(X).
false.

?- set_prolog_flag(occurs_check, error).
true.

?- X = f(X).
ERROR: Cannot unify <...> infinite tree
...

?- set_prolog_flag(occurs_check, false).
true.
```

The full text of the error is “Cannot unify _G with f(_G): would create an infinite tree”, with the internal name of the variable in place of `_G`, followed by the goal that raised it. The last query restores the default, which the rest of this book assumes.

Cyclic terms do not occur in a knowledge base built from a dump or a trace, whose facts are ground. They arise in rules, typically when a predicate that should construct a new term from an old one is written so that the new term contains the old one and the two are then unified. A deadlock in a wait-for graph is a cycle in the data, two threads each waiting for the other, but it is represented by a set of ground facts and found by a rule in Chapter 20; it is not a cyclic term, and nothing in this book needs one.

3.5 Deciding Without Binding

Two predicates answer questions about unifiability without performing any bindings.

`?=/2` succeeds when the question “are these two terms equal” can be decided now: either they are identical, or they can never unify. It fails when the answer depends on bindings that have not been made yet. It is a guard for code that must not commit to a comparison prematurely.

```
?- ?=(a, a).
true.

?- ?=(f(X), g(Y)).
true.

?- ?=(f(X), f(Y)).
false.

?- ?=(X, a).
false.
```

`subsumes_term/2` tests one-sided matching. `subsumes_term(General, Specific)` succeeds when `Specific` is an instance of `General`: there is a substitution for the variables of `General` alone that makes the two identical, and `Specific` is left untouched. It binds nothing. The predicate was added to the ISO standard in its second corrigendum and is the exact statement of “this frame matches that template”, as opposed to “this frame and that template can be made to agree”, which is what `=/2` says; the two coincide when the frame is ground, and differ when the candidate has variables of its own that `=/2` would happily bind.

```
?- subsumes_term(f(_), f(a)).
true.

?- subsumes_term(f(a), f(_)).
false.

?- subsumes_term(f(X, X), f(Y, Z)).
false.

?- subsumes_term(f(X, Y), f(Z, Z)).
true.

?- subsumes_term(frame(_, 0, ntdll, _, _),
   frame(100, 0, ntdll, 'NtDelayExecution', 20)).
true.
```

The second query fails because `f(a)` can only be matched to `f(_)` by binding the variable on the specific side, which subsumption forbids. The third fails because `f(X, X)` requires its two arguments to be the same term, and `Y` and `Z` are distinct variables; the fourth succeeds because binding `X` and `Y` both to `Z` is a substitution on the general side only. The last is the frame template of Section 3.3 as a one-sided test.

3.6 The Variables of a Term, and Copies

`term_variables/2` collects the distinct variables of a term into a list, in the order in which they are first encountered reading the term from left to right and depth first. A ground term yields the empty list, so `term_variables(T, [])` is another way of writing `ground(T)`.

```
?- term_variables(f(X, g(Y, X), Z), Vs).
Vs = [X, Y, Z].

?- term_variables(frame(T, 0, ntdll, F, _), Vs).
Vs = [T, F, _].

?- term_variables(f(a, b), Vs).
Vs = [].
```

For a template, the list is the list of positions the template leaves open, and its length is a crude measure of how general the template is. Later chapters need the predicate wherever a program must know which variables a goal has, as a meta-interpreter does (Chapter 29).

`copy_term/2` makes a copy of a term in which every variable has been replaced by a fresh variable, consistently: two occurrences of the same variable in the original become two occurrences of the same fresh variable in the copy. The copy and the original share no variables, so binding a variable of one leaves the other unchanged.

```
?- copy_term(f(X, Y, X), C).
C = f(_A, _, _A).

?- copy_term(f(X, Y, X), C), X = a.
X = a,
C = f(_A, _, _A).

?- copy_term(f(a, b), C).
C = f(a, b).
```

The toplevel names a fresh variable that occurs more than once in an answer `_A`, `_B`, …, and a fresh variable that occurs once `_`; the first answer therefore shows that the two occurrences of `X` became two occurrences of one new variable while `Y` became an unrelated one. The second answer shows that binding `X` afterwards did not touch the copy. A copy of a ground term is the term itself, since there is nothing to rename.

The renaming of a clause before head unification, described in Section 3.2, is `copy_term/2` applied to the clause, and the two most common explicit uses of the predicate follow from the same need. A template held in a variable and unified with a frame is consumed by the unification: its variables are now bound, and it can no longer match a different frame. Matching the same template against several frames in one conjunction therefore requires a copy for each. In Chapter 9, `findall/3` copies every solution it collects for the same reason, which is why bindings made inside it do not survive it. Bindings undone by backtracking need no copy: when the toplevel is asked for the next answer, the bindings of the previous one are retracted, and the template is as it was.

3.7 Diagnostic Use

The shared dump `dump1.pl` is consulted as in Chapter 2, and three frame templates are recorded as facts of `template/2`, so that they can be named, listed, and matched by a single predicate.

```
:- [dump1].

% template(Name, Frame): the frame template
% called Name.
template(top_in_ntdll,
         frame(_, 0, ntdll, _, _)).
template(cs_wait,
         frame(_, _, ntdll,
               'RtlEnterCriticalSection', _)).
template(unresolved,
         frame(_, _, _, 'Unknown', _)).
```

A template is used most directly by typing it as a query. The first template asks for every thread whose top frame is in `ntdll`, and for the function it is in; since the answers are produced by unifying the query with one `frame/5` fact after another, each answer is one frame.

```
?- frame(T, 0, ntdll, F, _).
T = 100,
F = 'NtWaitForMultipleObjects' ;
T = 101,
F = 'NtWaitForAlertByThreadId' ;
T = 103,
F = 'NtWaitForAlertByThreadId' ;
T = 104,
F = 'NtWaitForAlertByThreadId' ;
T = 105,
F = 'NtWaitForWorkViaWorkerFactory' ;
T = 106,
F = 'NtWaitForWorkViaWorkerFactory' ;
T = 108,
F = 'NtDelayExecution'.
```

Seven of the ten threads were in a system call in `ntdll` when the dump was written. The function names say what kind of wait each was: a multiple-object wait in the main thread's message loop, the alert-by-thread-id wait that the critical-section implementation uses in three worker threads, the worker-factory wait of two idle thread-pool threads, and a delay in the housekeeping thread. These are the threads an analyst skims past as **Passive Thread** (page 334) instances when reading `~*k` output, and the template is that skim written down. It is not quite the same as the state the debugger reports, and the difference is instructive: the conjunction with `thread_state/2` shows that six of the seven were waiting and one, thread 108, was suspended, which a template on frames alone cannot see.

```
?- frame(T, 0, ntdll, _, _),
   thread_state(T, suspended).
T = 108.
```

```
?- frame(T, 0, ntdll, _, _),
   thread_state(T, running).
false.
```

The complementary template, a top frame outside `ntdll`, needs an inequality, and Section 3.1 settles which one and where it goes. `M` is bound by the time `M \== ntdll` is reached, so the test compares two atoms and means what it says; placed before the `frame/5` goal it would compare an unbound variable with an atom, succeed vacuously, and filter nothing.

```
?- frame(T, 0, M, F, _), M \== ntdll.
T = 102,
M = reportengine,
F = 'Table::Find' ;
T = 107,
M = reportengine,
F = 'Node::Visit' ;
T = 109,
M = dbclient,
F = 'Buffer::Copy'.
```

The three remaining threads are the ones that were doing something when the dump was written: two running in the report engine, one faulting in the database client. Chapter 1 called the first two **Active Thread** (page 337) instances on the strength of a state fact; here the same threads are selected from their stack traces, and thread 109 joins them because an exception, not a wait, stopped it.

The second template looks below the top frame. A thread blocked on a critical section shows `RtlEnterCriticalSection` a few frames down, above the function that tried to enter it; the frame number is left open because the depth varies with the implementation’s internal calls.

```
?- frame(T, N, ntdll,
   'RtlEnterCriticalSection', _).
T = 101,
N = 3 ;
T = 103,
N = 3 ;
T = 104,
N = 3.
```

A thread whose stack trace shows it waiting to acquire a synchronization object is an instance of **Blocked Thread (Software)** (page 336): it was stopped by a wait that software requested, as opposed to the hardware waits of the catalog’s companion variant, and the question that follows is what it waits for and who holds it. The template becomes the recognizer `blocked_thread/1` by giving it a name:

```
% blocked_thread(TID): TID is blocked entering a
% critical section.
blocked_thread(TID) :-
    frame(TID, _, ntdll,
          'RtlEnterCriticalSection', _).
```

```
?- blocked_thread(T).
T = 101 ;
T = 103 ;
T = 104.

?- blocked_thread(102).
false.

?- blocked_thread(T),
   waits_for(T, critical_section, A).
T = 101,
A = 140695026507968 ;
T = 103,
A = 140695026508096 ;
T = 104,
A = 140695026507968.
```

The last query joins the recognizer with the `waits_for/3` facts that the debugger's `!locks` output contributes, and the shared variable `T` does the joining. Threads 101 and 104 wait for the same critical section, 103 for another; the addresses are printed in decimal, as Chapter 2 warned, and are `0x7ff6_1d13_a0c0` and `0x7ff6_1d13_a140` in the file. Following them to their owners, and the owners to what they wait for, is the **Wait Chain (Critical Sections)** (page 384) of Chapter 20, and the cycle among 103 and 104 is the deadlock of Chapter 1 seen from the stack traces rather than the lock list. The recognizer itself is limited in a way worth stating: it recognizes a wait by one function name in one module, which is how this dump's debugger rendered critical-section waits, and a thread blocked in a different synchronization primitive, or a dump whose symbols named the function differently, would not match it. Chapter 20 widens the test.

The repeated-variable idiom of Section 3.3 finds two adjacent frames of one thread with the same module and function:

```
?- frame(T, 0, M, F, _), frame(T, 1, M, F, _).
T = 107,
M = reportengine,
F = 'Node::Visit' ;
false.
```

Thread 107 has `Node::Visit` in frames 0 and 1, and in fact in frames 0 through 7. A stack trace in which the same frame recurs is an instance of **Frame Regularity** (page 342), a repeating structure in a stack trace that points to recursion or to a loop of calls, and a long run of it is the first thing to check when **Stack Overflow (User Mode)** (page 366) is suspected. The query above sees only the top two frames because frame numbers are constants in it; writing "frame `N` and frame `N + 1`" for any `N` needs arithmetic, and Chapter 18 does so. The `false.` after the answer is a choice point left in the first `frame/5` goal: other threads' top frames remained to be tried and none of them was followed by a matching frame 1.

Finally, the templates recorded as facts are matched by one predicate, `frame_matches/3`, which fetches a template by name, fetches a frame, and tests whether the frame is an instance of the template with `subsumes_term/2`. The frame facts are ground, so `=/2` would give the same answers;

`subsumes_term/2` states the intended direction and would continue to do so if the knowledge base held partially known frames.

```
% frame_matches(Name, TID, N): frame N of thread
% TID is an instance of the template called Name.
frame_matches(Name, TID, N) :-
    template(Name, Template),
    frame(TID, N, M, F, O),
    Frame = frame(TID, N, M, F, O),
    subsumes_term(Template, Frame).

?- frame_matches(cs_wait, TID, N).
TID = 101,
N = 3 ;
TID = 103,
N = 3 ;
TID = 104,
N = 3 ;
false.

?- frame_matches(unresolved, TID, N).
TID = 101,
N = 4 ;
false.

?- frame_matches(Name, 108, 0).
Name = top_in_ntdll ;
false.
```

The predicate works in every mode: given a template name, it enumerates the matching frames; given a frame, it enumerates the templates it matches; given both, it checks. The third query asks which named shapes the top frame of the suspended thread has. Each use of `template/2` renames the template's variables, as Section 3.2 explained, so successive matches do not interfere, and because `subsumes_term/2` binds nothing, the template is unchanged by a successful match. Chapter 17 takes this arrangement, patterns as data matched by a generic predicate, further.

Chapter 4. Clauses, Queries, and Search

The first three chapters used Prolog without saying how it answers a query. The answers appeared in a definite order, the toplevel sometimes printed `;` and sometimes a period, and a trailing `false.` turned up after the last answer of some queries and not others. This chapter explains all of it. A program is a set of clauses of a restricted logical form; a query is answered by reducing it, one goal at a time, against those clauses; the reductions that are possible form a tree; and Prolog walks the tree in a fixed order, depth first and left to right, keeping a record of the branches it has not yet tried. Everything the toplevel prints is a report on that walk.

The chapter adds four constructs to the three chapters' repertoire: disjunction `;/2`, the goals `true/0`, `fail/0`, and `false/0`, and the ordering of clauses and goals as a tool rather than an accident. It also introduces the tracer, which shows the walk as it happens, in prose and one abridged transcript. The diagnostic use is the question an analyst asks after `!locks`: which threads wait on which critical sections, and who owns them. Written as a conjunction over the `waits_for/3` and `critical_section/4` facts of `dump1.pl`, the question is answered by the search this chapter describes, and the attempt to follow the answer one step further, from the owner to what the owner waits for, meets the limit of the search in the form that matters most in diagnostics: a cycle in the data.

4.1 Horn Clauses

Every clause in a Prolog program has one of two forms. A **rule** is `Head :- Body`, where the head is a single goal and the body is a conjunction of one or more goals. A **fact** is a head alone. A query is a body alone. In logic these are three cases of one form, the **Horn clause**: a disjunction of literals of which at most one is positive. The positive literal, if any, is the head; the negative literals are the body goals. A clause with a head is a **definite clause**; a clause without one is a **goal clause**. Prolog programs consist of definite clauses, and queries are goal clauses.

A fact is a rule whose body is the goal `true/0`, which always succeeds. `owner(cs_a, 103).` and `owner(cs_a, 103) :- true.` are the same clause, and Chapter 11 shows that SWI-Prolog stores and reports them identically. `listing/1` prints facts in the shorter form. In a query, `true` is occasionally useful as a goal that does nothing:

```
?- true.
true.

?- true, true.
true.
```

The variables of a clause are **universally quantified** over the whole clause: `blocked_by(W, O) :- waits_for(W, CS), owner(CS, O)` says that for every `W`, `O`, and `CS`, if the body holds then the head holds. The variables of a query are read the other way: `?- blocked_by(101, O)` asks whether there exists an `O` such that `blocked_by(101, O)` holds, and an answer is a witness. Both readings follow

from the clause form, since a goal clause is the denial of a conjunction and so asks for a counterexample to the denial.

A **program** is a finite set of definite clauses. The declarative reading of Chapter 1 is now exact: a query is **a logical consequence** of the program when it holds in every interpretation in which all the clauses hold, and the answers Prolog prints are the substitutions for the query's variables under which this is so. Declaratively the program is a set, and the order of clauses within it means nothing. Procedurally SWI-Prolog keeps the clauses of each predicate in the order in which they appear in the source file, tries them in that order, and solves the goals of a body in the order in which they are written. Section 4.5 is about the consequences.

The clause form is restrictive. A head may not be a disjunction, so "either 103 or 104 owns `cs_a`" cannot be stated as a fact, and a body may not contain a negation, so "no thread owns `cs_a`" cannot be stated with the constructs of this chapter; Chapter 7 adds negation as failure. The restrictions are what make the search procedure of the next sections possible.

4.2 Goal Reduction

A query is answered by **goal reduction**. The state of the computation is the **resolvent**: the list of goals that remain to be solved, initially the goals of the query. One step of reduction does the following.

1. If the resolvent is empty, the computation has succeeded. The bindings made so far, restricted to the variables of the original query, are the answer.
2. Otherwise, select the leftmost goal of the resolvent. This is Prolog's **left-to-right selection**.
3. Find a clause of the selected goal's predicate whose head unifies with the goal. The clause is renamed first, as Section 3.2 described, so that its variables are fresh.
4. If no clause's head unifies, the computation along this path has failed.
5. Otherwise, replace the selected goal in the resolvent by the body of the clause, apply the unifier to the whole resolvent, and continue from step 1. A fact's body is `true`, which is removed at once.

The sequence of resolvents from the query to the empty resolvent is a **derivation**. This procedure is **SLD resolution**: resolution with a Selection function, applied Linearly, to Definite clauses. SLD resolution is sound, in that every answer it produces is a logical consequence of the program, and complete for the clause form of Section 4.1, in that every logical consequence is reachable by some derivation. Whether Prolog finds a given derivation depends on how it searches, which is the subject of Section 4.3.

The following knowledge base, three facts about the top frames of three threads and three facts about which module each function lives in, is small enough to reduce by hand.

```
% top(TID, Function): the top frame of thread
% TID is in Function.
top(102, 'Table::Find').
top(107, 'Node::Visit').
top(109, 'Buffer::Copy').

% in_module(Function, Module): Function belongs
% to Module.
in_module('Table::Find', reportengine).
in_module('Node::Visit', reportengine).
in_module('Buffer::Copy', dbclient).

% runs_in(TID, Module): the top frame of TID is
% in a function of Module.
runs_in(TID, Module) :-
    top(TID, F),
    in_module(F, Module).
```

The query `runs_in(T, reportengine)` has the following derivation, written with the renamed clause and the unifier at each step. Renamed variables carry a subscript.

```
?- runs_in(T, reportengine).
   clause  runs_in(TID1, Module1) :-
               top(TID1, F1), in_module(F1, Module1).
   unifier TID1 = T, Module1 = reportengine
?- top(T, F1), in_module(F1, reportengine).
   clause  top(102, 'Table::Find').
   unifier T = 102, F1 = 'Table::Find'
?- in_module('Table::Find', reportengine).
   clause  in_module('Table::Find', reportengine).
   unifier (empty)
?-                                  success: T = 102
```

The first step replaces the query by the body of the only `runs_in/2` clause, with `Module1` already bound to `reportengine` because the head unified with the query. The second step selects `top(T, F1)`, the leftmost goal, and the first `top/2` fact whose head unifies with it is the first fact in the file. Its body is `true`, so the resolvent shrinks to the one remaining goal, with `F1` now bound. The third step solves that goal against a fact, and the resolvent is empty. The answer is the binding of the query variable, `T = 102`; the bindings of `F1`, `TID1`, and `Module1` are internal to the derivation and not reported.

```
?- runs_in(T, reportengine).
T = 102 ;
T = 107 ;
false.
```

The toplevel agrees with the hand derivation, and then finds a second answer and reports that there is no third. Step 3 of the procedure said “find a clause”, not “find the clause”, and at the second step of the derivation three clauses were candidates. The derivation above followed the first. The others are followed on backtracking, and the structure they form is the subject of the next section.

4.3 The Search Tree

The derivations of a query form the **search tree** of the query. Its root is the query. Each node is a resolvent. The children of a node are the resolvents obtained by reducing its leftmost goal with each clause whose head unifies with it, in clause order from left to right. A node with the empty resolvent is a **success leaf**; a node whose leftmost goal unifies with no clause head is a **failure leaf**. The tree for `runs_in(T, reportengine)`:

```
?- runs_in(T, reportengine)
    |
    | runs_in/2, clause 1
    |
?- top(T, F), in_module(F, reportengine)
    |                   |                   |
    | top/2 #1          | top/2 #2          | top/2 #3
    | T = 102           | T = 107           | T = 109
    |                   |                   |
?- in_module(       ?- in_module(      ?- in_module(
   'Table::Find',       'Node::Visit',     'Buffer::Copy',
   reportengine)        reportengine)      reportengine)
    |                   |                   |
    | in_module #1      | in_module #2      | no clause
    |                   |                   |
   []  success         []  success         failure
```

The tree has two success leaves and one failure leaf. The answers to the query are the success leaves, read off in the order in which the search reaches them. The tree is a property of the program and the query; how it is explored is a property of Prolog.

Prolog explores the search tree by **depth-first search**: from a node it descends into the first child, and from there into that child's first child, and so on, until it reaches a leaf. Whenever a node has more than one child, Prolog records a **choice point** before descending into the first: a note of the node and of which children remain. On reaching a failure leaf, and on being asked for another answer after a success leaf, Prolog **backtracks**: it returns to the most recent choice point with untried children, undoes every binding made since that choice point was recorded, and descends into the next child. When a choice point has no untried children left it is discarded and backtracking continues to the one before. When no choice point remains, the search is over.

The toplevel transcript is a report on this walk. SWI-Prolog prints an answer when the search reaches a success leaf, and prints `;` and waits only when a choice point remains, because only then can there be a further answer. If no choice point remains it prints a period, and the user is not asked. Pressing `;` resumes the search by backtracking. If the resumed search reaches another success leaf, the next answer is printed; if it exhausts the tree without one, `false.` is printed. In the transcript above, `T = 107` was followed by `;` because the choice point at the second node still held the third `top/2` fact, and the resumed search descended into the failure leaf and found nothing else.

The alternatives recorded at a choice point are **clauses that might apply**, not clauses that do apply. SWI-Prolog decides which clauses are candidates for a goal by indexing on the goal's arguments; when the goal's bound arguments rule out all the clauses but one, no choice point is recorded, and the answer is printed with a period. This is why the second query below ends without a trailing `false.` although the first argument is unbound: SWI-Prolog indexes the second argument when the first is a variable, finds two candidates for `reportengine`, and after the second has no third to try.

```
?- in_module(F, dbclient).
F = 'Buffer::Copy'.

?- in_module(F, reportengine).
F = 'Table::Find' ;
F = 'Node::Visit'.

?- top(T, F).
T = 102,
F = 'Table::Find' ;
T = 107,
F = 'Node::Visit' ;
T = 109,
F = 'Buffer::Copy'.
```

A trailing `false.` therefore says nothing about the program's logic: it records whether SWI-Prolog could tell, at the time of the last answer, that nothing remained. The set of answers is the same either way. Chapter 13 describes the indexing.

Depth-first search is **complete** for finite trees: every success leaf is eventually reached. It is incomplete for infinite trees. If the first child of some node is the root of an infinite subtree with no success leaf, depth-first search descends into it and never comes back, and a success leaf under a later sibling is never reached although the corresponding answer is a logical consequence of the program. Section 4.5 shows how infinite trees arise and how clause and goal order keep the search away from them.

4.4 Conjunction, Disjunction, True, and Fail

The body of a clause, and a query, is built from goals with two connectives. **Conjunction**, written with the comma, is the operator `','/2`: `(A, B)` succeeds when `A` succeeds and then `B` succeeds under the bindings `A` made. It is right-associative, so `A, B, C` is `(A, (B, C))`, and the search procedure's "leftmost goal" is `A`. **Disjunction**, written with the semicolon, is the operator `;/2`: `(A ; B)` succeeds when `A` succeeds or when `B` succeeds. In the search tree a disjunction is a node with two children, `A` followed by the rest of the body and `B` followed by the rest of the body, explored in that order; a choice point is recorded, and `B` is tried when backtracking reaches it, whether or not `A` ever succeeded.

```
?- (X = a ; X = b).
X = a ;
X = b.
```

```
?- X = 1 ; X = 2 ; X = 3.
X = 1 ;
X = 2 ;
X = 3.

?- (true ; true).
true ;
true.
```

The third query shows that a disjunction yields an answer for each alternative that succeeds, even when the two answers are indistinguishable. The operator priorities of Chapter 2 govern how the connectives group: `;` has priority 1100 and `,` 1000, so `A, B ; C` is `(A, B) ; C` and parentheses are required to write `A, (B ; C)`.

```
?- X = 1, X = 2 ; X = 3.
X = 3.

?- X = 1, (X = 2 ; X = 1).
X = 1.
```

In the first query the left alternative `X = 1, X = 2` fails, because `X` cannot be both, and the right alternative gives the only answer. In the second, the disjunction is reached with `X` already bound to 1, its left alternative fails, and its right alternative succeeds trivially. The layout convention of this book for a disjunction in a clause body puts each alternative on its own line, preceded by `(` or `;` and indented to show the extent:

```
% stopped(TID): the top frame of TID is in the
% report engine or in the database client.
stopped(TID) :-
    (   runs_in(TID, reportengine)
    ;   runs_in(TID, dbclient)
    ).
```

A disjunction in a body is equivalent to two clauses with the same head and one alternative each, and the two forms produce the same answers in the same order. This book prefers the two-clause form when the alternatives are whole goals, since each clause then reads as a statement on its own, and the disjunction form when the alternatives share goals that would otherwise be written twice. A disjunction whose left alternative contains `->` is a different construct, if-then-else, treated in Chapter 7.

```
?- stopped(T).
T = 102 ;
T = 107 ;
T = 109.
```

`fail/0` is a goal that never succeeds; `false/0` is the same goal under a name that reads better in a declarative context, added to the ISO standard in its second corrigendum. Reaching either in a body is reaching a failure leaf, and the search backtracks at once. A query consisting of `fail` alone has an empty search tree:

```
?- fail.
false.

?- false.
false.

?- (fail ; true).
true.
```

The use of `fail` is to force backtracking. A conjunction that ends in `fail` can never succeed, so every goal before it is backtracked into until the choice points are exhausted, and any output produced on the way is printed as a side effect. This prints each thread identifier in the knowledge base without the toplevel being asked for each answer in turn:

```
?- top(T, _), write(T), nl, fail.
102
107
109
false.
```

The query fails, as it must, and the `false.` at the end is its only answer; the three lines above it were printed during the search. This idiom is the **failure-driven loop**. Chapter 7 treats it with the other control constructs, and Chapter 9 gives the alternative of collecting all answers into a list. Here it shows what `fail` does: the three `top/2` facts were visited in order because they were the three children of the one choice point in the tree.

4.5 The Order of Clauses and Goals

The declarative reading of a program ignores order. The procedural reading is nothing but order, and three things depend on it: which answer comes first, how much of the tree is searched, and whether the search terminates.

Clause order determines the order of a predicate's children at every node of the tree, and hence the order of answers. The first `runs_in/2` answer was `T = 102` because the `top/2` fact for 102 comes first in the file. Reversing the facts would reverse the answers and change nothing else. Where a predicate's answers are to be consumed in a particular order, that order must be produced by the clauses or imposed afterwards by sorting (Chapter 5), never assumed.

Goal order determines the shape of the tree. The two goals of `runs_in/2` can be written the other way round:

```
% runs_in_reordered(TID, Module): as runs_in/2,
% with the module looked up first.
runs_in_reordered(TID, Module) :-
    in_module(F, Module),
    top(TID, F).
```

```
?- runs_in_reordered(T, reportengine).
T = 102 ;
T = 107.

?- runs_in_reordered(T, M).
T = 102,
M = reportengine ;
T = 107,
M = reportengine ;
T = 109,
M = dbclient.
```

The answers are the same, as the declarative reading promises. The trees differ. With the module known, the reordered clause selects `in_module(F, reportengine)` first, and its two children, the two functions of the report engine, each lead to a `top/2` goal that succeeds. There is no failure leaf, and indexing leaves no choice point after the second answer, so the transcript ends with a period. In the original order, `top(T, F)` was selected with both arguments unbound, every `top/2` fact was a child, and one child led to a failure leaf. On three facts the difference is a node; on ten thousand frames and ten modules it is the difference between visiting every frame and visiting the frames of one module. The guideline follows from the tree: put first the goal that, given what is bound at that point, has the fewest clauses that apply, and put tests, such as the `\==` of Chapter 3, immediately after the goal that binds what they test.

Termination is the third dependency and the one that cannot be fixed afterwards. A search terminates when its tree is finite or when depth-first exploration reaches every success leaf and the user stops asking. A tree is infinite when some resolvent reduces, after one or more steps, to a resolvent that contains a renamed copy of itself. Recursion makes this possible. The following knowledge base records static calls between the functions of the report engine of `dump1.pl`, and `reaches/2` is the **transitive closure** of the relation: `F` reaches `G` when `F` calls `G` or calls something that reaches `G`.

```
% calls(Caller, Callee): Caller contains a call to
% Callee.
calls('Report::Build', 'Table::Insert').
calls('Table::Insert', 'Table::Find').
calls('Report::Build', 'Tree::Walk').
calls('Tree::Walk', 'Node::Visit').
calls('Node::Visit', 'Node::Visit').

% reaches(F, G): there is a chain of one or more
% calls from F to G.
reaches(F, G) :-
    calls(F, G).
reaches(F, H) :-
    calls(F, G),
    reaches(G, H).
```

The first clause is the **base case**, a reduction that does not reintroduce `reaches/2`; the second is the **recursive case**. Written in this order, and with the recursive goal last, the predicate behaves well on the acyclic part of the data:

```
?- reaches('Table::Insert', G).
G = 'Table::Find' ;
false.

?- calls(F, F).
F = 'Node::Visit'.
```

The second query shows the problem: `Node::Visit` calls itself. It is a recursive function, which is what the eight repeated frames of thread 107 in `dump1.pl` show dynamically, and in the call graph it is a cycle of length one. The search tree of any query that reaches it is infinite, because `reaches('Node::Visit', H)` reduces by the second clause to `calls('Node::Visit', G), reaches(G, H)`, which reduces to `reaches('Node::Visit', H)` again.

```
?- reaches('Report::Build', G).
G = 'Table::Insert' ;
G = 'Tree::Walk' ;
G = 'Table::Find' ;
G = 'Node::Visit' ;
G = 'Node::Visit' ;
G = 'Node::Visit' ;
...
```

The query has infinitely many answers, and the search produces them for as long as the user presses `;`. Each is a logical consequence of the program; a chain of any number of self-calls is a chain. Here the infinite branch contains success leaves, so the user sees something. The query `reaches('Report::Build', 'Table::Find')` is worse: it succeeds once, and when asked for another answer the search enters the cycle, where no leaf satisfies the goal, and never returns. This is an **infinite loop**, a search that runs forever and prints nothing; it is not shown in a transcript because it does not end.

Swapping the two goals of the recursive clause, or putting the recursive clause first, makes things worse in a different way. With `reaches(F, H) :- reaches(F, G), calls(G, H)` as the first clause, the leftmost goal of the first child of the root is again a `reaches/2` goal with the same first argument and a fresh second argument, and the search descends through an unbounded sequence of such goals without consulting `calls/2` at all. This is **left recursion**, and since each descent records a choice point for the untried second clause, SWI-Prolog's local stack grows until the limit set by the flag `stack_limit`, by default one gigabyte in this edition, is reached, and the query ends with a resource error after some seconds.

```
% Left recursion: do not write this.
reaches_left(F, H) :-
    reaches_left(F, G),
    calls(G, H).
reaches_left(F, G) :-
    calls(F, G).
```

```
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 48.4Mb, trail: 0Kb
ERROR:   Stack depth: 6,340,556, last-call: 0%, ...
ERROR:   Probable infinite recursion (cycle):
ERROR:     [6,340,556] user:reaches_left(...)
ERROR:     [6,340,555] user:reaches_left(...)
```

SWI-Prolog reports the depth reached, notices that consecutive frames are the same goal, and names the likely cause. Left recursion fails on every input, cyclic or not, because the infinite branch is the first branch explored. Right recursion with the base case first fails only on cyclic data, and only after the finite answers have been produced. The rule for recursive predicates follows: the base case first, the recursive goal last, and some argument that gets smaller with each recursive call, so that the base case is eventually reached. The third condition is what cyclic data violates. Chapter 5 supplies the standard remedy, a list of nodes already visited, and Chapter 11 supplies tabling, under which `reaches/2` as written terminates on any finite graph.

4.6 The Tracer

The search can be watched. SWI-Prolog's **tracer** stops at each of four events in the life of a goal, prints the goal, and waits for a keystroke. The four events are the **ports** of the **procedure box model**, in which each goal is a box entered and left on either side. **Call**: the goal is selected and its first clause is about to be tried. **Exit**: the goal has succeeded, with its bindings. **Redo**: backtracking has returned to the goal, and its next alternative is about to be tried. **Fail**: the goal has no (more) alternatives and fails. In terms of Section 4.3, Call is the descent into a node, Exit is reaching a success leaf below it, Redo is backtracking to a choice point recorded at or below it, and Fail is the exhaustion of that choice point. SWI-Prolog adds further ports, among them Exception, for goals that raise an error (Chapter 7).

`trace/0` switches the tracer on for the rest of the current query and, if entered as a query on its own, for the queries that follow; `notrace/0` switches it off, and `nodebug/0` also leaves debug mode. At each port the tracer prints the port name, the depth of the goal in parentheses, and the goal, then a `?` and waits. Enter, printed back as `creep`, proceeds to the next port. `s` (skip) runs the goal to its Exit or Fail port without showing what happens inside; `l` (leap) runs until the next spy point; `a` aborts the query; `h` lists the commands. The following is the tracer's output for the query of Section 4.2, abridged by removing the blank lines and with the depth counter as SWI-Prolog printed it, which starts above 1 because the toplevel's own goals occupy the first levels.

```
?- trace, runs_in(T, reportengine).
   Call: (11) runs_in(_1142, reportengine) ? creep
   Call: (12) top(_1142, _2360) ? creep
   Exit: (12) top(102, 'Table::Find') ? creep
   Call: (12) in_module('Table::Find', reportengine) ? creep
   Exit: (12) in_module('Table::Find', reportengine) ? creep
   Exit: (11) runs_in(102, reportengine) ? creep
T = 102 ;
   Redo: (12) top(_1142, _2360) ? creep
   Exit: (12) top(107, 'Node::Visit') ? creep
   Call: (12) in_module('Node::Visit', reportengine) ? creep
   Exit: (12) in_module('Node::Visit', reportengine) ? creep
   Exit: (11) runs_in(107, reportengine) ? creep
T = 107 ;
   Redo: (12) top(_1142, _2360) ? creep
   Exit: (12) top(109, 'Buffer::Copy') ? creep
   Call: (12) in_module('Buffer::Copy', reportengine) ? creep
   Fail: (12) in_module('Buffer::Copy', reportengine) ? creep
   Fail: (11) runs_in(_1142, reportengine) ? creep
false.
```

The transcript is the search tree of Section 4.3 read top to bottom and left to right. The unbound variables are printed with SWI-Prolog's internal names, `_1142` and `_2360`, which differ from run to run. One detail repays attention: after `T = 102` the Redo port is that of `top/2`, not of `in_module/2`. The goal `in_module('Table::Find', reportengine)` left no choice point, because its first argument was bound and indexed to a single clause, so backtracking went straight past it to the choice point of `top/2`. The tracer shows the choice points that exist, not the ones a reader of the program might expect.

A **spy point**, set with `spy/1` on a predicate indicator and removed with `nospy/1` or `nospyall/0`, makes the tracer stop at the ports of that predicate only, so that a long computation can be run with `l` until the predicate of interest is reached. Setting a spy point switches on **debug mode**, in which SWI-Prolog keeps the information the tracer needs at some cost in speed; `nodebug/0` leaves it. None of these commands appear in the transcripts of this book. For the pattern predicates of Parts III and IV the tracer is the way to see which fact a recognizer matched and where a recognizer that should have matched gave up.

4.7 Diagnostic Use

The `!locks` output of a debugger lists, for each critical section, its address, its owning thread, and its lock and recursion counts; the stack traces show which threads are waiting to enter one, and the debugger or the analyst works out which. In `dump1.pl` the first list is the `critical_section/4` relation and the second is `waits_for/3`, and the working out is a conjunction.

```
:- [dump1].
```

```
% blocked_by(Waiter, Owner): Waiter waits for a
% critical section that Owner owns.
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, critical_section, Addr),
    critical_section(Addr, Owner, _, _),
    Owner \== 0.
```

The rule is the `blocked_by/2` of Chapter 1 on the full schema, with one addition. The schema records an unowned critical section with 0 as its owner, and a thread recorded as waiting for an unowned section would otherwise be reported as blocked by thread 0, which does not exist. The test comes after the goal that binds `Owner`, for the reason given in Section 3.7. The search tree of `blocked_by(W, O)` has three children under the `waits_for/3` goal, one per fact, and each leads to a `critical_section/4` goal with a bound address that unifies with exactly one fact:

```
?- blocked_by(W, O).
W = 101,
O = 103 ;
W = 103,
O = 104 ;
W = 104,
O = 103.

?- blocked_by(102, O).
false.
```

Thread 102 is not blocked; it is the running thread of Chapter 1, and the `false.` records that no `waits_for/3` fact has it as first argument. The answers for the other three threads are the first links of what the memory analysis catalog calls a **Wait Chain (Critical Sections)** (page 384): a sequence of threads in which each waits for a critical section owned by the next, so that the first thread's wait is explained by the last thread's activity, and the analyst's attention moves along the chain to its end. Following the chain one link at a time is a conjunction of `blocked_by/2` goals sharing variables:

```
?- blocked_by(101, O), blocked_by(O, O2).
O = 103,
O2 = 104.

?- blocked_by(101, O), blocked_by(O, O2),
   blocked_by(O2, O3).
O = O3, O3 = 103,
O2 = 104.
```

Thread 101 waits for 103, which waits for 104, which waits for 103 again. The toplevel reports the third answer as `O = O3, O3 = 103`: the two variables were bound to each other and then to 103, and the chain has returned to a thread it already contained. A wait chain that closes on itself is a **Deadlock (Critical Sections)** (page 388): a set of threads each waiting for a critical section owned by another thread of the set, so that none of them can proceed. The two-thread case is the symmetric conjunction of Chapter 1, now over the dump:

```
?- blocked_by(A, B), blocked_by(B, A).
A = 103,
B = 104 ;
```

```
A = 104,
B = 103.
```

Each query so far names the length of the chain in the number of its goals. The transitive closure of Section 4.5 removes that limitation:

```
% blocked_through(TID, Other): TID is blocked by
% Other directly or through a chain of owners.
blocked_through(TID, Other) :-
    blocked_by(TID, Other).
blocked_through(TID, Other) :-
    blocked_by(TID, Owner),
    blocked_through(Owner, Other).

?- blocked_through(101, X).
X = 103 ;
X = 104 ;
X = 103 ;
X = 104 ;
...
```

The predicate is `reaches/2` over the wait-for graph, and the graph contains the cycle 103 → 104 → 103, so the search tree is infinite in the manner of Section 4.5: the threads that block 101 are reported, correctly, and then reported again for as long as the user asks. A **Wait Chain (Critical Sections)** (page 384) is followed to its end; a **Deadlock (Critical Sections)** (page 388) is a wait chain without an end; and a predicate that follows chains by recursion over the data does not terminate on the dumps that have a deadlock in them. The predicate is correct under the declarative reading and unusable under the procedural one on exactly the input for which it is needed. The remedy is to carry the list of threads already visited and refuse to revisit one, which requires lists (Chapter 5), or to let SWI-Prolog remember the answers already found, which is tabling (Chapter 11); Chapter 20 writes the recognizers for both patterns in both styles.

Disjunction has its own use on these facts. A thread is involved with a critical section when it owns it or waits for it, and the two cases share nothing but the head:

```
% involved(TID, Addr): TID owns or waits for the
% critical section at Addr.
involved(TID, Addr) :-
    (   critical_section(Addr, TID, _, _),
        TID \== 0
    ;   waits_for(TID, critical_section, Addr)
    ).

?- involved(T, 0x7ff6_1d13_a0c0).
T = 103 ;
T = 101 ;
T = 104.

?- involved(T, 0x7ff6_1d13_a1c0).
false.
```

```
?- involved(103, A).
A = 140695026507968 ;
A = 140695026508096.
```

The first answer of the first query comes from the left alternative, the owner, and the next two from the right, the waiters, in the order of the `waits_for/3` facts; the order is that of the search tree, not a ranking. The second query shows the third critical section of the dump, which is unowned and has no waiters; without the `TID \== 0` test the left alternative would have reported thread 0. The third query lists both sections thread 103 is involved with, printed in decimal as Chapter 2 warned: it owns the first and waits for the second, which is the half of the deadlock the thread contributes.

Chapter 5. Lists and Recursion

The knowledge base of a dump is flat: one fact per frame, one per module, one per thread. Analysis is not flat. A stack trace is read as a sequence from the top frame down; two stack traces are compared frame by frame from the bottom up; the modules of a thread are collected and counted. The term that holds a sequence in Prolog is the list, introduced as a term shape in Chapter 2, and the way a program works through a list, or builds one, is recursion. This chapter covers both: list notation and its unification behavior, the recursive clause pairs that define relations over lists, the predicates of `library(lists)` that package the most common ones, accumulators, tail recursion, and difference lists.

The diagnostic use is the stack trace itself. A predicate `stack_trace/2` collects the frames of a thread into a list in frame order by recursion over the frame numbers, and from that list follow the questions an analyst asks of a stack trace: how deep it is, how many times its top function repeats, which modules it passes through, and what two traces have in common at their base. The last of these is a use of `append/3` in a mode that has no counterpart in a conventional language, and the visited list promised in Chapter 4 makes the wait chain of `dump1.pl` terminate.

5.1 Lists as Terms

A **list** is either the empty list `[]` or a compound term with two arguments, the **head**, which is any term, and the **tail**, which is a list. Chapter 2 gave the functor, `'[|]'/2` in SWI-Prolog 7 and later, and the bracket notation that hides it. The notation `[H | T]` writes a list by its head and tail; `[a, b | T]` names two leading elements and the remaining list; `[a, b, c]` is `[a | [b | [c | []]]]`. These are three spellings of one term shape, and unification treats them as such.

```
?- [H | T] = [ntdll, kernelbase, kernel32].
H = ntdll,
T = [kernelbase, kernel32].

?- [H | T] = [ntdll].
H = ntdll,
T = [].

?- [H | T] = [].
false.

?- [a, b | T] = [a, b, c, d].
T = [c, d].
```

The third query fails because `[]` is not a compound term and has neither head nor tail. Every recursive definition over lists rests on this distinction: a list is empty, or it has a head and a tail, and nothing else is a list.

A list whose tail is an unbound variable is a **partial list**. Binding the variable extends it, and binding the variable to `[]` closes it. A list that ends in `[]` is a **proper list**; `is_list/1` succeeds for proper lists only.

```
?- L = [1, 2 | T], T = [3].
L = [1, 2, 3],
T = [3].

?- is_list([a | T]).
false.

?- length([a | T], 3).
T = [_, _].
```

The last query shows `length/2` closing a partial list: asked for a list of three elements that begins with `a`, it binds the tail to a list of two fresh variables. Partial lists are the mechanism behind the accumulators and difference lists of Sections 5.4 and 5.5.

5.2 Recursion over Lists

A predicate is **recursive** when a clause of it has a goal for the same predicate in its body. Chapter 4 wrote one, `reaches/2`, over a graph, and found that its termination depended on the data. Recursion over lists is better behaved, because a list has a structure that every recursive step makes smaller: the tail of a list has one element fewer than the list. A definition with one clause for `[]` and one for `[H | T]` whose recursive goal is on `T` is **structural recursion**, and it terminates on every proper list.

The relation “X is an element of the list” is the first example. An element is either the head or an element of the tail:

```
% list_member(X, List): X is an element of List.
list_member(X, [X | _]).
list_member(X, [_ | T]) :-
    list_member(X, T).
```

There is no clause for `[]`, because nothing is an element of the empty list; a goal `list_member(X, [])` unifies with neither head and fails. Read declaratively, the two clauses are a complete definition of membership. Read procedurally, the predicate searches the list from the front, and the search tree of Chapter 4 has one branch per element:

```
?- list_member(X, [ntdll, kernel32, ntdll]).
X = ntdll ;
X = kernel32 ;
X = ntdll ;
false.

?- list_member(kernel32, [ntdll, kernel32]).
true ;
false.
```

```
?- list_member(appserver, [ntdll, kernel32]).
false.
```

The trailing `false.` in the first two transcripts is the second clause, still a candidate when the last element has been reached, being tried on the empty tail. The library predicate `member/2` is the same relation, defined so that SWI-Prolog's indexing can see that no alternative remains when the last element is reached:

```
?- member(X, [ntdll, kernel32, ntdll]).
X = ntdll ;
X = kernel32 ;
X = ntdll.

?- member(kernel32, [ntdll, kernel32]).
true.

?- member(ntdll, [ntdll, kernel32, ntdll]).
true ;
true.
```

The third query succeeds twice, once for each occurrence, because `member/2` is a relation between an element and a list and does not know that the two occurrences are the same atom. A predicate that merely checks membership and should succeed once is `memberchk/2`, whose behavior is that of `member/2` followed by the cut of Chapter 7, and which is deferred to it.

The second example is concatenation. The relation "`Zs` is `Xs` followed by `Ys`" holds when `Xs` is empty and `Zs` is `Ys`, or when `Xs` and `Zs` have the same head and the relation holds between their tails and `Ys`:

```
% list_append(Xs, Ys, Zs): Zs is the concatenation
% of Xs and Ys.
list_append([], Ys, Ys).
list_append([X | Xs], Ys, [X | Zs]) :-
    list_append(Xs, Ys, Zs).

?- list_append([a, b], [c], Zs).
Zs = [a, b, c].

?- list_append(Xs, Ys, [a, b, c]).
Xs = [],
Ys = [a, b, c] ;
Xs = [a],
Ys = [b, c] ;
Xs = [a, b],
Ys = [c] ;
Xs = [a, b, c],
Ys = [] ;
false.

?- list_append(Xs, [c], [a, b, c]).
Xs = [a, b] ;
false.
```

The second and third queries run the definition backwards. Given only the result, the first clause offers the split with an empty prefix, and each application of the second clause moves one element from the suffix to the prefix; the predicate enumerates every way of splitting a list in two. Given the result and a suffix, it finds the prefix. Nothing in the definition says which arguments are input; the three clauses state a relation, and the search of Chapter 4 finds whatever instances of it the query leaves open. The library predicate `append/3` has exactly this definition, and Section 5.6 uses the third mode to compare stack traces.

Recursion also transforms lists. The following predicate relates a list of module names to the same list with one module's occurrences removed, which is a filter, and shows the shape of every list filter written by hand: a clause for the empty list, a clause for a head that is kept, and a clause for a head that is dropped.

```
% without_module(Ms, M, Rest): Rest is Ms without
% the occurrences of module M.
without_module([], _, []).
without_module([M | Ms], M, Rest) :-
    without_module(Ms, M, Rest).
without_module([M0 | Ms], M, [M0 | Rest]) :-
    M0 \== M,
    without_module(Ms, M, Rest).

?- without_module([ntdll, kernel32, ntdll,
   appserver], ntdll, Rest).
Rest = [kernel32, appserver] ;
false.
```

The second and third clauses are distinguished by the test `M0 \== M`, placed after head unification has bound `M0`, as Chapter 3 requires; without the test the third clause would also apply to a head equal to `M` and the predicate would have a second, wrong answer in which `ntdll` is kept. The `\==` makes the predicate correct for the mode in which `M` is bound, which is the mode the comment states; called with `M` unbound it would compare an atom to a variable and keep everything. Chapter 10 introduces `exclude/3` and `include/3`, which take the test as an argument and remove the need to write this pattern each time.

A recursion need not visit the whole list. The leading run of a list is the longest prefix whose elements are all the same term:

```
% leading_run(List, X, Run): Run is the longest
% prefix of List all of whose elements are X.
leading_run([], _, []).
leading_run([X | Xs], X, [X | Run]) :-
    leading_run(Xs, X, Run).
leading_run([Y | _], X, []) :-
    Y \== X.

?- leading_run([a, a, b, a], a, Run).
Run = [a, a] ;
false.
```

```
?- leading_run([a, a, b, a], X, Run).
X = a,
Run = [a, a] ;
Run = [].
```

The second query, with `X` unbound, has a second answer in which `X` is left unbound and the run is empty, because the third clause's test `Y \== X` succeeds for an unbound `X`. The predicate is correct when `X` is bound and over-general when it is not. This is the general situation with `\==` in a recursive definition, and the right response is to state the intended mode in the comment and call the predicate accordingly; Chapter 14 shows `dif/2`, which makes such definitions correct in every mode.

5.3 The List Library

The predicates of `library(lists)` are the recursions that programs need most often. In SWI-Prolog they are **autoloaded**: a goal for an undefined predicate that a library exports causes the library to be loaded on first use, when the flag `autoload` has its default value `true`, so that no directive is needed. The predicates used in this book:

Predicate	Relation
`member(?X, ?List)`	`X` is an element of `List`
`append(?Xs, ?Ys, ?Zs)`	`Zs` is `Xs` followed by `Ys`
`length(?List, ?N)`	`List` has `N` elements
`nth0(?I, ?List, ?X)`	`X` is element `I`, from 0
`nth1(?I, ?List, ?X)`	`X` is element `I`, from 1
`last(?List, ?X)`	`X` is the last element
`reverse(?Xs, ?Ys)`	`Ys` is `Xs` reversed
`msort(+List, -Sorted)`	sort, keeping duplicates
`sort(+List, -Sorted)`	sort, removing duplicates
`sort(+K, +Ord, +List, -S)`	sort on key `K` by `Ord`
`sum_list(+Ns, -Sum)`	`Sum` of the numbers
`max_list(+Ns, -Max)`	largest number
`min_list(+Ns, -Min)`	smallest number
`numlist(+L, +H, -Ns)`	`[L, L+1, ..., H]`
`select(?X, ?Xs, ?Rest)`	`Rest` is `Xs` without one `X`
`subtract(+S, +D, -R)`	`R` is `S` without `D`'s elements
`intersection(+A, +B, -R)`	elements of `A` also in `B`
`union(+A, +B, -R)`	`A` plus `B`'s new elements
`list_to_set(+L, -Set)`	`L` without later duplicates
`delete(+L, @X, -R)`	`L` without elements unifying `X`

`length/2` works in both directions and, with both arguments unbound, enumerates lists of every length:

```
?- length([a, b, c], N).
N = 3.

?- length(L, 2).
L = [_, _].

?- length(L, N).
L = [],
N = 0 ;
L = [_],
N = 1 ;
L = [_, _],
N = 2 ;
...
```

The positional predicates `nth0/3` and `nth1/3` differ only in whether the first element has index 0 or 1; given the element, they enumerate its positions. `last/2` fails on the empty list. `reverse/2` is treated in Section 5.4.

```
?- nth0(1, [a, b, c], X).
X = b.

?- nth1(1, [a, b, c], X).
X = a.

?- nth0(I, [a, b, a], a).
I = 0 ;
I = 2.

?- last([a, b, c], X).
X = c.
```

The sorting predicates use the standard order of terms of Chapter 2. `msort/2` sorts and keeps duplicates; `sort/2` sorts and removes duplicates, where two elements are duplicates when they are identical in the sense of `==/2`. `sort/4` generalizes both: its first argument is 0 to compare whole elements or the index of the argument on which to compare, and its second is `@<` or `@>` to sort ascending or descending and remove duplicates, or `@=<` or `@>=` to sort and keep them. When sorting on an argument, duplicates are elements with equal keys, and only one of them, in this edition the first, survives `@<`.

```
?- msort([c, a, b, a], L).
L = [a, a, b, c].

?- sort([c, a, b, a], L).
L = [a, b, c].

?- sort(0, @>=, [c, a, b, a], L).
L = [c, b, a, a].

?- sort(1, @<, [f(2, a), f(1, b), f(2, c)], L).
L = [f(1, b), f(2, a)].
```

```
?- sort(1, @=<, [f(2, a), f(1, b), f(2, c)], L).
L = [f(1, b), f(2, a), f(2, c)].

?- msort([ntdll, 10, "s", f(a), 2.0, X], L).
L = [X, 2.0, 10, "s", ntdll, f(a)].
```

The last query is the standard order in one line: variables, then numbers by value, then strings, then atoms, then compound terms. The sort of `sort/4` with `@=<` or `@>=` is stable: elements with equal keys keep their relative order, which is why `f(2, a)` precedes `f(2, c)`. Sorting is the usual way to impose the order that Chapter 4 said must never be assumed of a predicate's answers, and Chapter 9 adds `predsort/3` and `keysort/2` for orders the standard order does not express.

The numeric predicates take a list of numbers. `sum_list/2` sums it, with 0 for the empty list; `max_list/2` and `min_list/2` fail on the empty list; `numlist/3` builds the list of consecutive integers between two bounds and fails when the upper bound is below the lower. The older name `sumlist/2` is deprecated in SWI-Prolog and should not appear in new code.

```
?- sum_list([120, 15, 8450], S).
S = 8585.

?- max_list([120, 15, 8450], M).
M = 8450.

?- max_list([], M).
false.

?- numlist(0, 4, L).
L = [0, 1, 2, 3, 4].
```

`select/3` removes one occurrence of an element and, with the element unbound, enumerates each element together with the rest of the list. The set-like predicates `subtract/3`, `intersection/3`, and `union/3` treat lists as sets whose elements are compared by unification, and so are meant for lists of ground terms without duplicates; `list_to_set/2` removes every element identical to an earlier one and keeps the order of first occurrences; `delete/3` removes every element that unifies with its second argument without binding it.

```
?- select(X, [a, b, c], Rest).
X = a,
Rest = [b, c] ;
X = b,
Rest = [a, c] ;
X = c,
Rest = [a, b].

?- subtract([a, b, c, d], [b, d], R).
R = [a, c].

?- intersection([a, b, c], [c, a, x], R).
R = [a, c].
```

```
?- union([a, b], [b, c], R).
R = [a, b, c].

?- list_to_set([a, b, a, c, b], S).
S = [a, b, c].

?- delete([f(1), f(2), g(1)], f(_), R).
R = [g(1)].
```

`list_to_set/2` and `sort/2` both remove duplicates and differ in the order they leave: the first keeps the input order, the second imposes the standard order. Which is wanted depends on whether the list came from a sequence, as a stack trace does, or from an unordered collection.

5.4 Accumulators and Tail Recursion

`list_append/3` builds its result as it descends: the head of the result is fixed in the clause head, and the recursive goal fills in the tail. Some relations cannot be built this way, because the first element of the result is known only when the recursion has reached the end of the input. Reversal is the standard case. The direct definition reverses the tail and appends the head to the result:

```
% naive_reverse(Xs, Ys): Ys is Xs reversed.
naive_reverse([], []).
naive_reverse([X | Xs], Ys) :-
    naive_reverse(Xs, Zs),
    list_append(Zs, [X], Ys).
```

It is correct and slow: `list_append/3` walks the whole partial result at every step, so reversing a list of a thousand elements takes about half a million steps. The remedy is an **accumulator**, an extra argument that carries the result built so far. Each step moves one element from the input to the front of the accumulator, and when the input is exhausted the accumulator is the result:

```
% reverse_acc(Xs, Acc, Ys): Ys is Xs reversed,
% followed by Acc.
reverse_acc([], Ys, Ys).
reverse_acc([X | Xs], Acc, Ys) :-
    reverse_acc(Xs, [X | Acc], Ys).

% list_reverse(Xs, Ys): Ys is Xs reversed.
list_reverse(Xs, Ys) :-
    reverse_acc(Xs, [], Ys).

?- list_reverse([a, b, c], Ys).
Ys = [c, b, a].

?- reverse_acc([a, b], [x], Ys).
Ys = [b, a, x].
```

The accumulating predicate has a two-argument wrapper that supplies the initial value, here `[]`, and the comment of the accumulating predicate states the invariant that relates the three arguments at every step. This is the discipline of every accumulator: the wrapper is the predicate the rest of the

program calls, and the invariant is what makes the recursive clause checkable by reading. The library's `reverse/2` is `list_reverse/2` under its usual name. Accumulators that count or sum need arithmetic, and Chapter 6 writes them; the visited list of Section 5.6 is an accumulator that records what a search has already seen.

In `reverse_acc/3` the recursive goal is the last goal of the clause. A predicate whose recursive goals are all in this position is **tail recursive**, and SWI-Prolog, like most Prolog systems, performs **last-call optimization**: when the last goal of a clause is called and no choice point has been left in the clause, the clause's stack frame is reused for the called goal. A tail-recursive predicate without choice points therefore runs in constant stack space whatever the length of its input, where `naive_reverse/2`, whose recursive goal is followed by another goal, needs a stack frame per element. The condition about choice points matters: a predicate that leaves a choice point at each level, as `list_member/2` does, keeps its frames whether or not the recursive call is last. Chapter 7 shows how to write recursive predicates that leave none.

5.5 Difference Lists

A partial list together with its unbound tail is a list that can be extended at the end in one step. The pair is a **difference list**, conventionally written `List-Hole` with the operator `-/2` used purely as a pair constructor: `[a, b | H]-H` is the difference list whose elements are `a` and `b`, and `H-H` is the empty one. The name comes from reading `List-Hole` as "the elements of `List` up to `Hole`". Appending two difference lists is a single unification: the hole of the first is bound to the list of the second, and the result's hole is the second's.

```
% dl_append(Xs, Ys, Zs): the difference list Zs is
% Xs followed by Ys.
dl_append(Front-Mid, Mid-Back, Front-Back).

?- dl_append([a, b | H1]-H1, [c | H2]-H2, Zs).
H1 = [c|H2],
Zs = [a, b, c|H2]-H2.

?- dl_append([a, b | H1]-H1, [c | H2]-H2, L-[]).
H1 = [c],
H2 = [],
L = [a, b, c].
```

The first answer shows the appended difference list still open at `H2`; the toplevel prints the partial list with its tail variable after the bar. The second query closes it by unifying the hole with `[]` and obtains a proper list. The cost of `dl_append/3` does not depend on the lengths involved, where `append/3` walks its first argument. The price is that a difference list can be appended to only once, since its hole is then bound, and that a program must keep the two halves of the pair together. Difference lists appear in this book in two places: Section 5.6 uses one to concatenate stack traces as they are built, and Chapter 12 shows that definite clause grammars are difference lists in a notation that hides the pair.

5.6 Diagnostic Use

The shared dump is consulted as in the previous chapters. One toplevel setting is changed for this section: the flag `toplevel_print_anon`, when set to `false`, stops the toplevel from printing the bindings of variables whose names begin with an underscore. A stack trace as a list of seven `frame/5` terms is wider than a page, and the queries below bind it to a variable named `_Frames` and print only what is derived from it.

```
:- [dump1].

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

A **Stack Trace** (page 329) in the memory analysis catalog is the sequence of frames of one thread as the debugger reconstructs it, read from the top, which was executing when the dump was written, towards the bottom, the thread's entry point; it is the first object of analysis for any thread, and the catalog's caution that it is a reconstruction, not an observation, was given in Chapter 1. In `dump1.pl` the frames of a thread are facts with consecutive frame numbers from 0, and the list is built by recursion over those numbers:

```
% stack_trace(TID, Frames): Frames is the list of
% frame/5 terms of thread TID, top frame first.
stack_trace(TID, Frames) :-
    frames_from(TID, 0, Frames).

% frames_from(TID, N, Frames): Frames is the list
% of frames of TID numbered N and above.
frames_from(TID, N,
            [frame(TID, N, M, F, O) | Frames]) :-
    frame(TID, N, M, F, O),
    succ(N, N1),
    frames_from(TID, N1, Frames).
frames_from(TID, N, []) :-
    \+ frame(TID, N, _, _, _).
```

Two built-in predicates appear here ahead of the chapters that treat them. `succ/2` relates a non-negative integer to its successor: `succ(N, N1)` holds when `N1` is `N + 1`, and either argument, though not both, may be unbound; Chapter 6 places it among the arithmetic predicates, where it is the one that is a relation rather than an evaluation. `\+/1`, **negation as failure**, succeeds exactly when its goal has no solution under the current bindings and binds nothing; the goal here is `frame/5` with its first two arguments bound and the rest anonymous, so it asks whether thread `TID` has a frame numbered `N`, and nothing less definite. Chapter 7 states the conditions under which `\+` means logical negation and the ways it can go wrong when its goal has unbound variables that matter.

```
?- succ(3, N).
N = 4.
```

```
?- succ(N, 4).
N = 3.

?- \+ frame(100, 7, _, _, _).
true.

?- stack_trace(100, _Frames), length(_Frames, N).
N = 7 ;
false.

?- stack_trace(999, Frames).
Frames = [].
```

The first clause of `frames_from/3` applies while frame `N` exists and the second when it does not, and the two are exclusive, so the predicate has one answer. The `;` after `N = 7` reports that the second clause was still a candidate at every level and had to be tried before SWI-Prolog could be sure; Chapter 7 removes the choice point with if-then-else. The last query shows the base case meeting a thread that is not in the dump: the stack trace of an absent thread is the empty list, which is correct for the question asked and is a reminder that the predicate cannot distinguish a thread with no frames from a thread that does not exist. A list of frames is printed one frame per line by two recursions, the first projecting each frame to its module and function and the second writing the pairs:

```
% frame_symbols(Frames, Symbols): Symbols is the
% list of Module-Function pairs of Frames.
frame_symbols([], []).
frame_symbols([frame(_, _, M, F, _) | Frames],
              [M-F | Symbols]) :-
    frame_symbols(Frames, Symbols).

% print_symbols(Symbols): print one
% Module!Function pair per line.
print_symbols([]).
print_symbols([M-F | Symbols]) :-
    write(M), write('!'), write(F), nl,
    print_symbols(Symbols).

% print_stack_trace(TID): print the stack trace of
% TID in the debugger's module!function form.
print_stack_trace(TID) :-
    stack_trace(TID, Frames),
    frame_symbols(Frames, Symbols),
    print_symbols(Symbols).

?- print_stack_trace(109).
dbclient!Buffer::Copy
dbclient!Connection::Receive
appserver!Session::Fetch
appserver!Worker::Run
ucrtbase!thread_start<...>
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
true ;
false.
```

The output is the stack trace of the faulting thread in the form a debugger prints it, without offsets, which need the hexadecimal formatting of Chapter 6. The pair `M-F` is the conventional Prolog representation of a key with a value, and `print_symbols/1` takes it apart by unification in the clause head.

Three properties of a stack trace follow from the list at once. Its depth is the length of the list. The run of identical frames at its top is `leading_run/3` applied to the symbols. The set of modules it passes through is the sorted list of its modules without duplicates, which `sort/2` produces, or the modules in the order first met, which `list_to_set/2` produces.

```
?- stack_trace(107, _Frames), length(_Frames, N).
N = 14 ;
false.

?- stack_trace(107, _Frames),
   frame_symbols(_Frames, _Symbols),
   leading_run(_Symbols, Symbol, _Run),
   length(_Run, N).
Symbol = reportengine-'Node::Visit',
N = 8 ;
N = 0 ;
false.

?- stack_trace(101, _Frames),
   frame_symbols(_Frames, _Symbols),
   leading_run(_Symbols, _, _Run),
   length(_Run, N).
N = 1 ;
N = 0 ;
false.
```

Thread 107 has fourteen frames, the deepest stack trace in the dump, and the top eight are the same function. Chapter 3 found two adjacent `Node::Visit` frames with a frame template; the list makes the length of the run available, and the catalog's **Frame Regularity** (page 342) is now a question about that length rather than about two fixed frame numbers. The second answer of each query, `N = 0`, is the over-general case noted in Section 5.2, in which `leading_run/3` leaves its second argument unbound; Chapter 7 removes it. Chapter 18 formalizes the pattern and relates run length to the stack depth at which **Stack Overflow (User Mode)** (page 366) becomes the hypothesis.

```
% frame_modules(Frames, Modules): Modules is the
% list of module names of Frames, in frame order.
frame_modules([], []).
frame_modules([frame(_, _, M, _, _) | Frames],
              [M | Modules]) :-
    frame_modules(Frames, Modules).

?- stack_trace(105, _Frames),
   frame_modules(_Frames, Ms).
Ms = [ntdll, ntdll, kernel32, ntdll] ;
false.
```

```
?- stack_trace(105, _Frames),
   frame_modules(_Frames, _Ms), sort(_Ms, Set).
Set = [kernel32, ntdll] ;
false.

?- stack_trace(105, _Frames),
   frame_modules(_Frames, _Ms),
   list_to_set(_Ms, Set).
Set = [ntdll, kernel32] ;
false.
```

Comparing two stack traces is the use of `append/3` promised in Section 5.2. Two threads started by the same code share the bottom of their stacks, and the question is how far up the agreement extends. A common suffix of two symbol lists is a list that is a suffix of both, and `append(_, Suffix, List)` states that `Suffix` is a suffix of `List`:

```
% common_suffix(Xs, Ys, Suffix): Suffix is a
% suffix of both Xs and Ys; the longest is first.
common_suffix(Xs, Ys, Suffix) :-
    append(_, Suffix, Xs),
    append(_, Suffix, Ys).

% common_tail(TID1, TID2, Symbols): Symbols is a
% common suffix of the two threads' stack traces.
common_tail(TID1, TID2, Symbols) :-
    stack_trace(TID1, Frames1),
    stack_trace(TID2, Frames2),
    frame_symbols(Frames1, Symbols1),
    frame_symbols(Frames2, Symbols2),
    common_suffix(Symbols1, Symbols2, Symbols).
```

The first `append/3` goal enumerates the suffixes of `Xs` from the whole list downwards, as the second transcript of `list_append/3` showed, and the second goal tests each against `Ys`. The first suffix to pass is therefore the longest common one, and the later answers are its own suffixes down to `[]`. The comparison is on module and function, not on `frame/5` terms, which carry the thread identifier and frame number and would never agree between two threads.

```
?- common_tail(101, 103, _S),
   print_symbols(_S), length(_S, N).
appserver!Worker::Run
ucrtbase!thread_start<...>
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
N = 4 ;
ucrtbase!thread_start<...>
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
N = 3 ;
...
```

```
?- common_tail(101, 104, _S),
   print_symbols(_S), length(_S, N).
ucrtbase!thread_start<...>
kernel32!BaseThreadInitThunk
ntdll!RtlUserThreadStart
N = 3 ;
...
```

Threads 101 and 103 agree on four frames: both are worker threads that reached their wait through `Worker::Run`. Thread 104 shares only the three thread-start frames with 101, because it was started by the same runtime but runs a different loop, `Reader::Loop` in the database client. The three bottom frames are common to every worker in the dump and say nothing about any of them; the frame above them is where the traces begin to differ, and it is the first frame worth reading. Chapter 18 returns to common frames between stack traces. A predicate that stops after the longest suffix needs `once/1` from Chapter 7; here the remaining answers are shown as `...`.

The visited list of Chapter 4 is an accumulator in the sense of Section 5.4: it carries the threads a search has passed through, and each step adds one. With it, following a wait chain terminates on cyclic data.

```
% blocked_by(Waiter, Owner): Waiter waits for a
% critical section that Owner owns.
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, critical_section, Addr),
    critical_section(Addr, Owner, _, _),
    Owner \== 0.

% blocked_through(TID, Other): Other blocks TID
% directly or through a chain of owners.
blocked_through(TID, Other) :-
    blocked_through(TID, [TID], Other).

% blocked_through(TID, Visited, Other): as above,
% never passing through a thread in Visited.
blocked_through(TID, Visited, Other) :-
    blocked_by(TID, Other),
    \+ member(Other, Visited).
blocked_through(TID, Visited, Other) :-
    blocked_by(TID, Owner),
    \+ member(Owner, Visited),
    blocked_through(Owner, [Owner | Visited],
                    Other).

?- blocked_through(101, T).
T = 103 ;
T = 104 ;
false.

?- blocked_through(103, T).
T = 104 ;
false.

?- blocked_through(103, 103).
false.
```

Thread 101 is held up by 103 and, through 103, by 104, and the search stops there because 103 is already in the visited list when the chain returns to it. The predicate is the **Wait Chain (Critical Sections)** (page 384) of Chapter 4 made to terminate, and its last answer shows what the visited list costs: `blocked_through(103, 103)` fails, because the thread that would close the cycle is the one the search started from and so is excluded from the first step. The deadlock is still in the data, but this predicate no longer reports it; the two-thread conjunction of Chapter 4 does, and Chapter 20 writes a cycle finder that carries the visited list and still recognizes the return to the start. A visited list buys termination at the price of a predicate that must be read carefully for what it has been told to ignore.

Finally, the difference list of Section 5.5 builds the frames of two threads into one list without `append/3`. The recursion is that of `frames_from/3` with the hole carried as a fourth argument, and the base case unifies the hole with the frames that follow:

```
% frames_dl(TID, N, Frames, Hole): Frames is the
% frames of TID from N on, followed by Hole.
frames_dl(TID, N,
          [frame(TID, N, M, F, O) | Frames],
          Hole) :-
    frame(TID, N, M, F, O),
    succ(N, N1),
    frames_dl(TID, N1, Frames, Hole).
frames_dl(TID, N, Hole, Hole) :-
    \+ frame(TID, N, _, _, _).

% two_stack_traces(TID1, TID2, Frames): the frames
% of TID1 followed by those of TID2.
two_stack_traces(TID1, TID2, Frames) :-
    frames_dl(TID1, 0, Frames, Hole),
    frames_dl(TID2, 0, Hole, []).


?- two_stack_traces(105, 106, _Frames),
   length(_Frames, N).
N = 8 ;
false.

?- two_stack_traces(105, 106, _Frames),
   frame_modules(_Frames, _Ms),
   list_to_set(_Ms, Set).
Set = [ntdll, kernel32] ;
false.
```

The first call leaves its result open at `Hole`; the second fills the hole with the second thread's frames and closes the list with `[]`. The same construction extended over every thread of the dump is the catalog's **Stack Trace Collection (Unmanaged Space)** (page 330), the list of all stack traces of a process, which Chapter 9 builds with `findall/3` once enumerating all threads is available and Chapter 18 treats as a pattern.

Chapter 6. Arithmetic

Nothing in the first five chapters added two numbers. Unification does not evaluate: `X = 1 + 2` binds `X` to the compound term `+(1, 2)`, as Chapter 3 showed, and a program that needs the number 3 has to ask for it. This chapter is about asking. It covers `is/2`, which evaluates an arithmetic expression and unifies the result; the six arithmetic comparison predicates, which evaluate both sides; the functions available inside expressions, on integers, floats, and bits; the relational predicates `succ/2`, `plus/3`, and `between/3`; and `format/2`, which prints numbers and other terms under the control of a directive string and, in particular, prints integers in hexadecimal. The toplevel prints integers in decimal, and until now every address in a transcript has appeared as a long decimal number; after this chapter they appear as the debugger shows them.

The diagnostic uses are arithmetic over addresses and over times. An address belongs to the module whose base and size enclose it, which is one inequality on each side; the offset of an address within a module, the page containing it, and its alignment are subtractions and bit operations; whether a faulting address lies in the first page of the address space is a comparison. A thread's share of the processor time consumed by a process is a division, and the **Spiking Thread** (page 335) pattern, which Chapter 1 could only promise, is recognized by comparing that share with a threshold.

6.1 Evaluation

`is/2` is the predicate that evaluates. `X is Expr` evaluates the term `Expr` as an **arithmetic expression** and unifies `X` with the result. The right side must be an expression whose variables are all bound to numbers at the time of the call; the left side may be anything that unifies with a number.

```
?- X = 1 + 2.
X = 1+2.

?- X is 1 + 2.
X = 3.

?- 3 is 1 + 2.
true.

?- 1 + 2 = 3.
false.
```

The second and fourth queries are the distinction this chapter turns on. `1 + 2` is a term with principal functor `+/2`; `3` is a number; they do not unify, because they are different terms. `is/2` evaluates its right side and the result, 3, unifies with the left. The **evaluation** of an expression proceeds from the inside out: a number evaluates to itself, a variable bound to a number evaluates to that number, and a compound term whose name and arity denote an **arithmetic function** evaluates to the function applied to the evaluated arguments. Anything else is an error.

```
?- X is Y + 1.
ERROR: Arguments are not sufficiently instantiated
...

?- X is foo + 1.
ERROR: Arithmetic: `foo/0' is not a function
...

?- X is 1 / 0.
ERROR: Arithmetic: evaluation error: <...>
...
```

The three errors are the three things that can go wrong. An unbound variable in the expression is an **instantiation error**: Prolog does not solve equations, and `3 is X + 1` raises the same error as the query above rather than binding `X` to 2. An atom or compound term that is not an arithmetic function is a type error. A function applied outside its domain is an **evaluation error**. Chapter 7 shows how to catch these errors and inspect their terms; here the first line of each message is shown and the rest, which names the goal and its context, is elided.

`is/2` is not a relation in the sense that `append/3` is. It has one working mode, `-Result is +Expression`, and a program that needs to go the other way, from a result to an operand, must use a predicate written for that direction. Three such predicates exist for the simplest cases. `succ/2`, which Chapter 5 borrowed, relates a non-negative integer to the next one in either direction. `plus/3` relates three integers of which any two determine the third. Both are SWI-Prolog predicates outside the ISO standard, and both raise an instantiation error when too few arguments are bound.

```
?- succ(X, 4).
X = 3.

?- plus(1, Y, 3).
Y = 2.

?- plus(1, 2, Z).
Z = 3.
```

6.2 Comparison

Six predicates compare the values of two expressions, evaluating both sides first: `=:=/2` equal, `=\=/2` not equal, `</2`, `>/2`, `=</2`, and `>=/2`. The spelling `=<` is deliberate: `<=` is not an operator in Prolog, because it would be read as an arrow, and a goal written with it is a syntax error.

```
?- 1 + 2 =:= 3.
true.

?- 2 =< 2, 3 >= 3, 2 =\= 3.
true.

?- 2 < 3.0.
true.
```

```
?- 1 =:= 1.0.
true.

?- 1 == 1.0.
false.
```

The last two queries separate arithmetic equality from the identity of Chapter 3. The integer 1 and the float 1.0 have the same value and are different terms. `=:=` compares values; `==` compares terms; `=` would fail as `==` does. The same distinction holds between arithmetic order and the standard order of terms of Chapter 2: `<` compares values and raises an error on a non-number, while `@<` compares any two terms and orders a float before an integer of the same value and an atom of digits by its characters.

```
?- 1.0 @< 1.
true.

?- 1.0 < 1.
false.

?- '10' @< '9'.
true.

?- a < 1.
ERROR: Arithmetic: `a/0' is not a function
...
```

The comparison predicates, like `is/2`, require their arguments to be bound to evaluable terms at the time of the call. A test such as `User + Kernel > 5000` must therefore come after the goal that binds `User` and `Kernel`, which is the rule Chapter 3 gave for `\==` and Chapter 4 gave for every test: tests follow the goals that bind what they test.

6.3 Integers and Floats

SWI-Prolog integers are **unbounded**, as Chapter 2 said; `2 ** 64` is computed exactly, and the sum of two 64-bit addresses cannot wrap. Floats are IEEE double precision, printed with the shortest sequence of digits that reads back as the same value. An expression with an integer and a float operand yields a float. Division is the one operation whose result type is not fixed by its operands. In SWI-Prolog, `/` applied to two integers yields an integer when the division is exact and a float otherwise; when the flag `iso` is `true` it always yields a float, as the ISO standard requires, and when the flag `prefer_rationals` is `true`, which it is not by default in this edition, it yields a rational number.

```
?- X is 2 ** 64.
X = 18446744073709551616.

?- X is 10 / 4.
X = 2.5.
```

```
?- X is 10 / 5.
X = 2.

?- X is 1 + 2.0.
X = 3.0.
```

Integer division has three forms, which differ on negative operands. `//` truncates toward zero, under the default value `toward_zero` of the flag `integer_rounding_function`, which the ISO standard leaves to the implementation; `div` rounds toward negative infinity. `mod` is the remainder of `div`, and takes the sign of the divisor; `rem` is the remainder of `//`, and takes the sign of the dividend. For non-negative operands the two pairs agree, and offsets and sizes in a dump are non-negative.

```
?- X is -7 // 2, Y is -7 div 2.
X = -3,
Y = -4.

?- X is -7 mod 2, Y is -7 rem 2.
X = 1,
Y = -1.
```

The remaining integer functions used in this book: `abs/1`, `sign/1`, `min/2` and `max/2`, the greatest common divisor `gcd/2`, and `msb/1`, the position of the most significant set bit of a positive integer, which is the integer part of its base-two logarithm. Exponentiation has two spellings. `**` and `^` both yield an integer for integer operands and a non-negative exponent and a float when either operand is a float; for a negative integer exponent both yield a float in this edition under the default flags, where ISO requires `^` on integers to yield an integer and treats a negative exponent as an error unless the base is 1 or -1.

```
?- X is abs(-3), Y is sign(-3), Z is sign(0).
X = 3,
Y = -1,
Z = 0.

?- X is min(2, 3.0), Y is max(2, 3.0).
X = 2,
Y = 3.0.

?- X is gcd(12, 18).
X = 6.

?- X is msb(1024), Y is msb(0x7ffb_8d10_31aa).
X = 10,
Y = 46.

?- X is 2 ** 10, Y is 2 ^ 100.
X = 1024,
Y = 1267650600228229401496703205376.

?- X is 2 ** 0.5.
X = 1.4142135623730951.
```

The value of `msb/1` for the address says that it needs 47 bits, which places it in the lower, user-mode half of a 64-bit address space. Conversion between integers and floats is explicit. `float/1` converts

an integer to a float; `integer/1` rounds a float to the nearest integer; `truncate/1` drops the fraction toward zero; `round/1` rounds to nearest with halves away from zero; `ceiling/1` and `floor/1` round up and down. `sqrt/1` always yields a float.

```
?- X is float(3), Y is sqrt(16).
X = 3.0,
Y = 4.0.

?- X is integer(2.5), Y is truncate(2.5).
X = 3,
Y = 2.

?- X is round(-2.5), Y is ceiling(2.1).
X = -3,
Y = 3.

?- Z is floor(-2.1).
Z = -3.
```

Floats carry the usual caution. The decimal fractions a trace records as timestamps are not representable exactly in binary, and the sum of two of them is not the float that would be written for the mathematical sum.

```
?- X is 0.1 + 0.2.
X = 0.30000000000000004.

?- 0.1 + 0.2 =:= 0.3.
false.
```

Chapter 25 computes time deltas from the timestamps of `trace1.pl` and compares them with thresholds, and the comparisons are written as inequalities against a tolerance, never as `=:=` against a constant. Integers have no such problem, which is one reason the shared schema records CPU times in milliseconds as integers rather than in seconds as floats.

6.4 Bits

Six functions operate on the binary representation of integers: bitwise and `/\`, or `\/`, exclusive or `xor`, complement `\`, and the shifts `<<` and `>>`. Since integers are unbounded, the representation is a two’s complement with infinitely many leading sign bits: `\ 0` is -1, and `\ 0xfff` is -4096, a number with ones in every position above the lowest twelve. The and of an address with it clears those twelve bits, which is the base of the 4 KB page containing the address; the and with `0xfff` itself is the offset within the page. The toplevel prints the results in decimal; Section 6.6 prints them in hexadecimal.

```
?- X is 0xff /\ 0x0f, Y is 0xf0 \/ 0x0f.
X = 15,
Y = 255.
```

```
?- X is 0xff xor 0x0f, Y is \ 0.
X = 240,
Y = -1.

?- X is 1 << 12.
X = 4096.

?- X is 0x7ffb_8d10_31aa /\ 0xfff.
X = 426.

?- X is 0x7ffb_8d10_31aa /\ \ 0xfff.
X = 140718380167168.

?- X is (0x7ffb_8d10_31aa >> 12) << 12.
X = 140718380167168.
```

The last two queries compute the page base in two ways, by masking and by shifting out and back the twelve low bits, and agree. The shift right by 32 separates the high and low halves of a 64-bit address, which is how a debugger prints one, with a backquote between the halves; Section 6.6 reproduces that notation. Bit tests also answer alignment questions: a pointer to an object aligned on eight bytes has `Addr /\ 7 =:= 0`, and a pointer that fails the test is either misaligned, which on most structures is a sign of corruption, or carries information in its low bits, a convention some runtimes use and the analyst must know about before reading the pointer as an address.

6.5 Generating Integers

`between/3` is the predicate that enumerates integers. `between(Low, High, X)` with `X` unbound succeeds once for each integer from `Low` to `High` inclusive, in increasing order, and with `X` bound tests whether it lies in the range. `High` may be the atom `inf` for an unbounded enumeration. The predicate is the standard way to drive a search over a range of frame numbers, page numbers, or candidate values; Chapter 30 uses it to generate test data by the hundred thousand.

```
?- between(1, 3, X).
X = 1 ;
X = 2 ;
X = 3.

?- between(1, 3, 2).
true.

?- between(1, 3, 5).
false.

?- between(1, inf, X).
X = 1 ;
X = 2 ;
X = 3 ;
...
```

Combined with the failure-driven loop of Chapter 4, `between/3` visits every value in a range and does something with each. The following query prints the squares of the first four integers, each produced by `is/2` from the value `between/3` supplies, and then fails as every such loop does.

```
?- between(1, 4, N), S is N * N,
   write(S), nl, fail.
1
4
9
16
false.
```

The accumulator of Chapter 5 that sums a list of numbers, left for this chapter, uses `is/2` in its recursive clause and nothing else:

```
% sum_acc(Ns, Acc, Sum): Sum is Acc plus the sum
% of the numbers in Ns.
sum_acc([], Sum, Sum).
sum_acc([N | Ns], Acc0, Sum) :-
    Acc is Acc0 + N,
    sum_acc(Ns, Acc, Sum).

?- sum_acc([120, 40, 8450, 1230], 0, Sum).
Sum = 9840.

?- sum_acc([], 0, Sum).
Sum = 0.
```

The two accumulator variables are named `Acc0` and `Acc`, the value before and after the step, a convention this book keeps for every accumulator. The library's `sum_list/2` is this predicate with the initial value supplied, and the recursion is tail recursive as Section 5.4 defined the term, so it runs in constant stack on a list of any length.

6.6 Formatted Output

`format/2` prints text under the control of a **format string**, whose characters are printed as they are except for **directives** introduced by `~`, each of which consumes an argument from the list given as the second argument or acts on the output position. `format/1` takes a format string without arguments, and `format/3`, which writes to a stream or into an atom, appears in Chapters 8 and 13. The directives used in this book:

Directive	Argument	Prints
`~w`	any term	as `write/1`
`~p`	any term	as `print/1`, quoted
`~q`	any term	as `writeq/1`, quoted
`~a`	atomic	the text, unquoted
`~d`	integer	decimal
`~D`	integer	decimal, groups of three

Directive	Argument	Prints
`~Nd`	integer	decimal point `N` from the right
`~Nf`	number	fixed, `N` decimals (default 6)
`~Ne`	number	exponential, `N` decimals
`~Ng`	number	as C's `%g`
`~Nr`	integer	radix `N`, lowercase digits
`~NR`	integer	radix `N`, uppercase digits
`~c`	code	the character
`~Nc`	code	the character `N` times
`~s`	string, codes	the text
`~i`	any term	nothing; skips the argument
`~*c`	count, code	the character, count times
`~n`	none	newline
`~~`	none	a tilde
`~N|`	none	column stop at column `N`
`~N+`	none	column stop `N` past the last
`~t`	none	fill point for the next column

The character directives first. `~w` prints an argument as `write/1` does, `~q` quotes atoms that need it, `~a` prints an atomic argument's text and raises an error on a compound term, and `~n` ends the line.

```
?- format("~w and ~a~n", [f(X), 'Buffer::Copy']).
f(_<...>) and Buffer::Copy
true.

?- format("~q ~w~n",
   ['Buffer::Copy', 'Buffer::Copy']).
'Buffer::Copy' Buffer::Copy
true.

?- format("~c~c ~s ~*c~n",
   [0'O, 0'K, "done", 3, 0'-]).
OK done ---
true.

?- format("~a~n", [f(x)]).
ERROR: Illegal argument to format sequence <...>
...
```

The unbound variable in the first query is printed with an internal name that varies between runs, which the transcript shows as `<...>`. The numeric directives are strict about their argument types. `~d` prints an integer and raises an error on a float; `~D` is `~d` with a comma between groups of three digits; `~Nd` with a numeric argument inserts a decimal point `N` digits from the right, which turns a count of milliseconds into seconds without any floating-point arithmetic. `~f` accepts integers and floats, and `~Nf` fixes the number of decimals.

```
?- format("~d ~D~n", [12040, 12040]).
12040 12,040
true.

?- format("~3d s~n", [9680]).
9.680 s
true.

?- format("~f ~3f ~0f ~3f~n", [2.5, 2.5, 2.5, 2]).
2.500000 2.500 2 2.000
true.

?- format("~e ~4e ~g~n",
   [12040.0, 12345.678, 0.5]).
1.204000e+04 1.2346e+04 0.5
true.

?- format("~d~n", [1.5]).
ERROR: Illegal argument to format sequence ~d: 1.5
...
```

`~Nr` prints an integer in radix `N`, from 2 to 36, with lowercase letters for the digits above 9, and `~NR` with uppercase. `~16r` is the directive this book uses for every address, exception code, and offset:

```
?- format("~16r ~16R ~8r ~2r~n",
   [255, 255, 255, 5]).
ff FF 377 101
true.

?- format("~16r~n", [0x7ffb_8d10_31aa]).
7ffb8d1031aa
true.

?- format("~16r~n", [0xc0000005]).
c0000005
true.
```

A debugger prints a 64-bit address with sixteen hexadecimal digits, leading zeros included, and `~16r` prints only the significant ones. Padding is the business of the **column directives**. `~N|` declares a column stop at column `N` of the current line, and `~N+` one at `N` positions past the previous stop. The text emitted since the previous stop is padded to reach the new one, and `~t` marks where the padding goes: before the text for right alignment, after it for left alignment, on both sides to center it. The padding character is a space unless the `~t` is written as `~c`t, `with the fill character`c`after a backquote. **Zero-padding**[[format!zero-padding]] an address to sixteen digits is therefore`~0t~16r~16|: fill with zeros, then the hexadecimal digits, then a stop at column 16.

```
?- format("~`0t~16r~16|~n", [0x7ffb_8d10_31aa]).
00007ffb8d1031aa
true.

?- format("0x~`0t~16r~18|~n", [0x7ffb_8d10_31aa]).
0x00007ffb8d1031aa
true.
```

```
?- format("~`0t~16r~8|~n", [0x18]).
00000018
true.
```

In the second query the stop moves to column 18 because the two characters of the prefix count toward it. The notation a Windows debugger uses, two groups of eight digits separated by a backquote, comes from splitting the address with the shift and mask of Section 6.4 and padding each half to a stop eight digits wide, the second stop nine columns past the first because the backquote occupies one:

```
% print_address(Addr): print a 64-bit address in
% the debugger's hhhhhhhh`llllllll notation.
print_address(Addr) :-
    High is Addr >> 32,
    Low is Addr /\ 0xffff_ffff,
    format("~`0t~16r~8|`~`0t~16r~9+~n",
           [High, Low]).

?- print_address(0x7ffb_8d10_31aa).
00007ffb`8d1031aa
true.

?- print_address(0x18).
00000000`00000018
true.
```

The same column stops build tables. With `~t` before a number and a stop after it, the number is right-aligned in its column; with the stop directly after an atom, the atom is left-aligned and padded. The following prints a module name in a 12-column field and a number right-aligned in the next 8:

```
?- format("~a~t~12|~t~d~8+~n", [ntdll, 160]).
ntdll            160
true.

?- format("~a~t~12|~t~d~8+~n",
   [reportengine, 9680]).
reportengine    9680
true.
```

The arguments of `format/2` are a list even when there is one, and a single non-list argument is accepted as a list of one; the number of directives that consume arguments must match the length of the list, and a shortfall is reported as an error after the text up to that point has been printed. The first argument may be a string, an atom, or a code list; this book writes it as a string.

6.7 Diagnostic Use

The dump is consulted, and the first question is which module an address belongs to. A module occupies the addresses from its base up to, but not including, its base plus its size, and `address_module/2` states exactly that:

```
:- [dump1].

% address_module(Addr, Module): Addr lies in the
% address range of Module.
address_module(Addr, Module) :-
    module(Module, Base, Size),
    Base =< Addr,
    Addr < Base + Size.

% print_symbol(Addr): print Addr as module+offset.
print_symbol(Addr) :-
    address_module(Addr, Module),
    module(Module, Base, _),
    Offset is Addr - Base,
    format("~a+0x~16r~n", [Module, Offset]).
```

The two comparisons are tests, and come after the `module/3` goal that binds `Base` and `Size`; with the module known they check containment, and with the module unknown they select it from the ten in the dump. The exception record of thread 109 gives an address to try:

```
?- exception_record(109, Code, Addr),
   address_module(Addr, M).
Code = 3221225477,
Addr = 140718380167594,
M = dbclient ;
false.

?- exception_record(109, Code, Addr),
   format("~16r at ", [Code]), print_symbol(Addr).
c0000005 at dbclient+0x31aa
Code = 3221225477,
Addr = 140718380167594 ;
false.
```

The faulting instruction is in `dbclient`, at offset `0x31aa` from its base, and the exception code is the access violation `c0000005` that the toplevel showed as 3221225477. The trailing `false.` is the choice point among the remaining `module/3` facts, which `address_module/2` has to try because the address ranges of modules are not known to it to be disjoint. An address that lies in no module is the complementary case, and the predicate fails on it:

```
?- address_module(0x0000_0210_a000_0000, M).
false.

?- address_module(0x18, M).
false.
```

The first address is the base of the process heap, which is data, not code; the second is the target of the faulting read, from `exception_info/3`. The lowest 64 KB of a Windows user-mode address space are reserved and never mapped, so an access to an address below `0x1_0000` is an access through a null pointer, and the address itself is the offset that was added to it: `0x18`, 24 bytes, the position of some field in a structure the code expected to find. The memory analysis catalog names this configuration **NULL Pointer (Data)** (page 360), a read or write through a null pointer to data, as

distinct from a jump through one, and Chapter 19 writes the recognizer from `exception_info/3`; the arithmetic it needs is the one comparison below.

```
?- exception_info(109, target, T), T < 0x1_0000,
   format("null + 0x~16r~n", [T]).
null + 0x18
T = 24.
```

Page arithmetic uses the bit operations of Section 6.4. The page base and the page offset of the faulting instruction, and the number of pages a module occupies:

```
?- exception_record(109, _, A), P is A /\ \ 0xfff,
   O is A /\ 0xfff, format("~16r ~16r~n", [P, O]).
7ffb8d103000 1aa
A = 140718380167594,
P = 140718380167168,
O = 426.

?- module(thirdparty_hook, _, Size),
   Pages is Size // 0x1000.
Size = 73728,
Pages = 18.
```

`between/3` walks the pages of a module. The following prints the first three page bases of the hooking module, in the debugger's notation, with the failure-driven loop of Chapter 4:

```
?- module(thirdparty_hook, Base, _),
   between(0, 2, I), Page is Base + I * 0x1000,
   print_address(Page), fail.
00007ffb`6f300000
00007ffb`6f301000
00007ffb`6f302000
false.
```

The second use is processor time. The `thread_time/3` facts give each thread's user and kernel time in milliseconds, and their sum is the thread's CPU time:

```
% cpu_time(TID, Ms): the CPU time of TID in
% milliseconds, user and kernel together.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.


?- cpu_time(102, Ms).
Ms = 9680.

?- cpu_time(TID, Ms), Ms > 1000.
TID = 102,
Ms = 9680 ;
TID = 107,
Ms = 4100 ;
false.
```

The second query is the recognizer of the style guide's example: a thread whose CPU time exceeds a constant. A constant in milliseconds is the wrong measure, because the same 4100 ms is unremarkable in a process that has run for an hour and alarming in one that has run for five seconds, and because the catalog's intent is comparative. **Spiking Thread** (page 335) names a thread that has consumed a share of the process's processor time out of proportion to the other threads, which is what `!runaway` shows and what an analyst sees as a thread stuck in a loop while the others wait. The share needs the total over all threads, and collecting a total over every fact of a relation is what `findall/3` and `aggregate_all/3` do in Chapter 9. With the constructs available so far it is done by visiting the threads in order of their identifiers: the thread after a given one is the thread with the smallest identifier above it, and the sum is accumulated along the walk.

```
% next_thread(TID, Next): Next is the thread with
% the smallest identifier greater than TID.
next_thread(TID, Next) :-
    thread(Next, _),
    Next > TID,
    \+ ( thread(Other, _),
         Other > TID,
         Other < Next
       ).

% cpu_sum_above(TID, Acc, Total): Total is Acc
% plus the CPU time of every thread above TID.
cpu_sum_above(TID, Acc0, Total) :-
    next_thread(TID, Next),
    cpu_time(Next, Ms),
    Acc is Acc0 + Ms,
    cpu_sum_above(Next, Acc, Total).
cpu_sum_above(TID, Total, Total) :-
    \+ next_thread(TID, _).

% total_cpu(Total): the CPU time of all threads.
total_cpu(Total) :-
    cpu_sum_above(0, 0, Total).

% cpu_share(TID, Percent): the CPU time of TID as
% a percentage of the total.
cpu_share(TID, Percent) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Percent is 100.0 * Ms / Total.
```

`next_thread/2` uses the negation of Chapter 5 over a conjunction: there is no other thread strictly between `TID` and `Next`. The variable `Other` occurs only inside the negated goal, where it is existentially quantified, "no `Other` such that", which is the use of an unbound variable under `\+` that Chapter 7 will call safe. The walk starts above identifier 0, since thread identifiers in the schema are positive integers, and ends at the thread with no successor. The factor `100.0` makes the percentage a float in every case; with `100`, a thread with no CPU time would get the integer `0` under the division rule of Section 6.3.

```
?- next_thread(0, T).
T = 100 ;
```

```
false.

?- next_thread(104, T).
T = 105 ;
false.

?- total_cpu(Total).
Total = 14135 ;
false.

?- cpu_share(102, P).
P = 68.48249027237354 ;
false.

?- cpu_share(108, P).
P = 0.0 ;
false.
```

Thread 102 alone accounts for more than two thirds of the processor time the process has used. The recognizer compares the share with a threshold, and the threshold is a fact rather than a number in the clause body, so that an analyst can see it, change it, and later let Chapter 17 generalize it:

```
% spiking_threshold(Percent): a thread whose share
% of the CPU time is at least Percent is spiking.
spiking_threshold(25).

% spiking_thread(TID): TID has consumed at least
% the threshold share of the process CPU time.
spiking_thread(TID) :-
    cpu_share(TID, Percent),
    spiking_threshold(Threshold),
    Percent >= Threshold.

?- spiking_thread(T).
T = 102 ;
T = 107 ;
false.

?- spiking_thread(100).
false.
```

Threads 102 and 107 are the two threads that were running when the dump was written, the **Active Thread** (page 337) instances of Chapter 1, and with ten threads the threshold of 25 percent is two and a half times an equal share. The threshold is a judgment, not a fact about the dump; 107's 29 percent is spiking next to eight threads near zero and would not be in a process with three busy workers. What the predicate adds to `!runaway` is the division and the comparison done for every thread without error; what it does not add is the judgment, which Chapter 17 discusses under thresholds and Chapter 18 under the pattern.

Finally, the table an analyst would write by hand. `format/2` prints one row per thread with the identifier, the CPU time in milliseconds, and the share to one decimal, right-aligned under column stops, and the failure-driven loop prints every row:

```
% print_cpu_share(TID): one table row for TID.
print_cpu_share(TID) :-
    cpu_time(TID, Ms),
    cpu_share(TID, Percent),
    format("~d~t~6|~t~D~8+ ms~3+~t~1f~8+%~n",
           [TID, Ms, Percent]).

?- print_cpu_share(102).
102      9,680 ms    68.5%
true ;
false.

?- thread(T, _), print_cpu_share(T), fail.
100        160 ms     1.1%
101         20 ms     0.1%
102      9,680 ms    68.5%
103         40 ms     0.3%
104         35 ms     0.2%
105         25 ms     0.2%
106         15 ms     0.1%
107      4,100 ms    29.0%
108          0 ms     0.0%
109         60 ms     0.4%
false.
```

The percentages are rounded by `~1f` for display and compared unrounded by `spiking_thread/1`; a report and a recognizer should not share a rounding. The `~D` groups the digits of the one time that needs it. The table is the `!runaway` output of the debugger recomputed from facts, with a share column that the debugger leaves to the analyst, and it took three clauses and a format string. Chapter 9 replaces the ordered walk of `total_cpu/1` by a single aggregation goal and sorts the rows by share, and Chapter 10 builds the same table with `foldl/4`; the arithmetic in the rows does not change.

Chapter 7. Control

Every predicate written so far states a relation and leaves the search to Prolog. The search is exhaustive: every clause that can apply is tried, and the toplevel reports the leftovers with a `;` after an answer that is in fact the only one. For a knowledge base of a few hundred facts the cost is invisible; for the meaning of a program it is not. A predicate that classifies a thread should give one class, not one class and the promise of more; a lookup of a value that may be absent should say so once; and a predicate handed an argument of the wrong kind should complain rather than fail as if the dump had nothing to say. This chapter covers the constructs that control the search: the cut, which discards alternatives; if-then-else, which chooses between two continuations without leaving a choice point; negation as failure, which turns the absence of a solution into success; `forall/2` and the failure-driven loop, which visit every solution; and exceptions, which leave the search altogether.

Under the cut and under negation the declarative and procedural readings of Chapter 1 part ways, and a program using them is correct only under conditions this chapter states: a cut is harmless when the alternatives it discards could not have succeeded, and negation is logical negation when the negated goal has no unbound variables that matter. The diagnostic use is classification and robust lookup. A predicate `thread_class/2` assigns each thread of `dump1.pl` exactly one class by an ordered chain of tests; the stack trace predicate of Chapter 5 is rewritten so that it leaves no choice point; and lookups of facts the dump may not contain are given defaults, with an error, not a failure, when the question itself is malformed.

7.1 Determinism and Choice Points

A goal is **deterministic** when it succeeds exactly once and leaves no choice point; it is **semideterministic** when it succeeds at most once, that is, it fails or it succeeds once without a choice point; and it is **nondeterministic** when it may succeed more than once. The terms describe a predicate in a given mode: `member/2` is nondeterministic with its first argument unbound, and `memberchk/2` of Section 7.2 is semideterministic in every mode.

The toplevel shows determinism. After printing an answer it prints `;` and waits only when a **choice point** remains, that is, when some goal in the proof has an alternative that has not been tried; when none remains it prints `.` and the query is finished. The two transcripts of `member/2` in Chapter 5 are the two cases:

```
?- member(a, [a, b]).
true ;
false.

?- member(a, [b, a]).
true.
```

In the first query `a` is found at the head of the list and the clause for the tail is still a candidate; the toplevel asks, the candidate is tried on `[b]`, and the query fails. In the second, `a` is found in the last

position, SWI-Prolog's clause indexing sees that no clause can match an empty tail, and no choice point is left. The `;` records that the first proof was not known to be the only one.

A choice point is not free. Chapter 5 said that last-call optimization applies only when no choice point has been left in the clause, so a recursive predicate that leaves one at each level uses stack in proportion to its input; a choice point also keeps alive every binding made since it was created. And it has a meaning: it says that the predicate may have another answer, which a predicate that computes one value from its inputs, such as the class of a thread, should not say. The constructs of this chapter remove choice points, and the transcripts show the removal as the change from `;` to `.`.

Clause order and head unification already give determinism in many cases. The following predicate classifies a 32-bit status code by its two most significant bits, which Windows defines as its severity: 0 for success, 1 for informational, 2 for warning, 3 for error.

```
% severity_class(Bits, Class): the class of a
% status code whose two top bits are Bits.
severity_class(0, success).
severity_class(1, informational).
severity_class(2, warning).
severity_class(3, error).

% code_class(Code, Class): Class is the severity
% class of the 32-bit status code Code.
code_class(Code, Class) :-
    Bits is Code >> 30,
    severity_class(Bits, Class).

?- code_class(0xc0000005, C).
C = error.

?- code_class(0x80000003, C).
C = warning.

?- code_class(0, C).
C = success.
```

Every answer ends in `.` because the first argument of `severity_class/2` is bound at the time of the call and SWI-Prolog's **first-argument indexing** selects the one clause that can match. A predicate whose clauses are distinguished by a bound first argument needs no cut; the construct is needed when clauses are distinguished by a test in the body, which head unification cannot see.

7.2 The Cut

The **cut**, written `!`, is a goal that always succeeds and, as a side effect on the search, discards every choice point created since the clause containing it was entered: the alternatives of the goals to the left of the cut in the clause body, and the remaining clauses of the predicate. After the cut, the clause is committed to: if a later goal in the body fails, the whole predicate fails, and no other clause is tried.

The standard first example is the larger of two numbers. Without a cut, the predicate has two clauses whose tests are complementary:

```
% max_plain(X, Y, Max): Max is the larger of X, Y.
max_plain(X, Y, X) :-
    X >= Y.
max_plain(X, Y, Y) :-
    X < Y.

?- max_plain(3, 5, M).
M = 5.

?- max_plain(5, 3, M).
M = 5 ;
false.
```

The second query leaves a choice point, because when the first clause succeeded the second had not yet been tried, and SWI-Prolog cannot know from the clause heads that its test will fail. A cut after the test of the first clause removes it:

```
% max_green(X, Y, Max): Max is the larger of X, Y.
max_green(X, Y, X) :-
    X >= Y,
    !.
max_green(X, Y, Y) :-
    X < Y.

?- max_green(5, 3, M).
M = 5.

?- max_green(3, 5, M).
M = 5.
```

This is a **green cut**: the clauses are exclusive, the second test would fail anyway, and the cut changes nothing about the set of answers; removing it leaves the predicate correct and slower. A **red cut** is one whose removal changes the answers. The tempting simplification of `max_green/3` is to drop the second test, since the cut ensures the second clause is reached only when the first test failed:

```
% max_red(X, Y, Max): Max is the larger of X, Y.
% WRONG: not steadfast.
max_red(X, Y, X) :-
    X >= Y,
    !.
max_red(_, Y, Y).

?- max_red(5, 3, M).
M = 5.

?- max_red(3, 5, M).
M = 5.

?- max_red(5, 3, 3).
true.
```

The third query is wrong: 3 is not the larger of 5 and 3. The head `max_red(X, Y, X)` does not unify with `max_red(5, 3, 3)`, so the first clause is skipped before its test and cut are reached, and the second clause, which no longer has a test, succeeds. The predicate is correct when its third argument is unbound and wrong when it is bound. A predicate that behaves the same whether its output arguments are bound at the call or unified afterwards is **steadfast**, and `max_red/3` is not; the cause is output unification in the head before the cut, and the remedy is to unify the output after it:

```
% max_of(X, Y, Max): Max is the larger of X and Y.
max_of(X, Y, Max) :-
    X >= Y,
    !,
    Max = X.
max_of(_, Y, Y).

?- max_of(5, 3, 3).
false.

?- max_of(5, 3, M).
M = 5.
```

The cut is still red, since removing it would let `max_of(5, 3, 3)` succeed through the second clause, but the predicate is steadfast. The rule is general: in a clause with a cut, an output argument is unified after the cut, never in the head. Section 7.3 gives the construct that makes both the cut and the rule unnecessary for predicates of this kind.

Conjunction and disjunction are **transparent to the cut**: a cut inside `( A ; B )` in a clause body cuts the whole clause. Negation and if-then-else are **opaque**: a cut inside `\+ G` or inside the condition of an if-then-else is local to that goal, as is a cut in a toplevel query.

Two built-in predicates package the most common uses of the cut. `once/1` calls its argument as a goal and succeeds at most once, discarding the goal's remaining choice points; it is `Goal, !` as a predicate, and the right way to take the first solution of a goal that has several. `ignore/1` calls its argument once and succeeds whether or not the goal did, the idiom for an optional action whose failure should not fail the clause.

```
?- once(member(X, [a, b, c])).
X = a.

?- ignore(member(x, [a, b, c])).
true.

?- ignore(member(X, [a, b, c])).
X = a.
```

`memberchk/2`, promised in Chapter 5, is `once(member(X, L))` under a shorter name. In a conjunction that only needs to know whether a module is in a list, `memberchk/2` is correct and `member/2` leaves a choice point that is never wanted.

```
?- member(kernel32, [ntdll, kernel32, ntdll]).
true ;
false.

?- memberchk(kernel32, [ntdll, kernel32, ntdll]).
true.

?- member(ntdll, [ntdll, kernel32, ntdll]).
true ;
true.

?- memberchk(ntdll, [ntdll, kernel32, ntdll]).
true.
```

The first query leaves the choice point for the rest of the list and the third succeeds twice, once per occurrence, as Chapter 5 showed; the two `memberchk/2` queries succeed once and are finished.

7.3 If-Then-Else

The **if-then-else** construct is written `( Cond -> Then ; Else )`. `Cond` is called as by `once/1`: if it has a solution, the first one is kept, its remaining choice points are discarded, and `Then` is called with the bindings `Cond` made; if it has no solution, `Else` is called, with no bindings from the attempt. Exactly one of the two branches runs, and no choice point is created by the choice itself, though `Then` or `Else` may leave their own. The operator `->/2` has lower priority than `,` and higher than `;`, so the condition and the branches may be conjunctions without further parentheses, and the whole construct is parenthesized because it is an argument of `;/2`.

```
?- ( 3 >= 5 -> M = 3 ; M = 5 ).
M = 5.

?- ( member(X, [1, 2, 3]), X > 1
   -> true ; X = none ).
X = 2.

?- ( member(X, [1, 2, 3]), X > 5
   -> true ; X = none ).
X = none.
```

In the second query the condition has two solutions, 2 and 3, and the construct commits to the first. In the third it has none, and the else branch runs with `X` still unbound. `( C -> T ; E )` means “T if C, otherwise E” only under the convention that `C` is tried once and its failure is taken as its falsity, which is the convention of Section 7.4.

The larger of two numbers is now written without a cut and is steadfast by construction, because the output is unified in a branch, after the test:

```
% larger(X, Y, Max): Max is the larger of X and Y.
larger(X, Y, Max) :-
    (   X >= Y
    ->  Max = X
    ;   Max = Y
    ).

?- larger(5, 3, M).
M = 5.

?- larger(3, 5, M).
M = 5.

?- larger(5, 3, 3).
false.
```

The layout is the one this book uses for every if-then-else: the opening parenthesis, the arrow, and the semicolon aligned in a column, each branch indented to the same depth, and the closing parenthesis under the opening one. A chain of conditions is written by nesting in the else branch; no inner parentheses are needed, so a chain reads as a sequence of guarded alternatives, the last of them unguarded:

```
% cpu_class(Ms, Class): the class of a thread by
% its CPU time in milliseconds.
cpu_class(Ms, Class) :-
    (   Ms >= 1000
    ->  Class = heavy
    ;   Ms > 0
    ->  Class = light
    ;   Class = idle
    ).

?- cpu_class(9680, C).
C = heavy.

?- cpu_class(40, C).
C = light.

?- cpu_class(0, C).
C = idle.
```

The conditions are tested in order and the first that succeeds selects its branch, so a later condition is tested under the assumption that the earlier ones failed: `Ms > 0` in the second guard means `0 < Ms < 1000`. The same predicate as three clauses would need each clause to state the full condition, or a cut in each, and would still leave choice points.

An if-then-else without an else branch, `( Cond -> Then )`, fails when `Cond` fails; it is `( Cond -> Then ; fail )`. A clause that should do nothing when the condition fails needs `( Cond -> Then ; true )`, and the omission is a common error.

```
?- ( 3 >= 5 -> M = 3 ).
false.

?- ( 3 >= 5 -> M = 3 ; true ).
true.
```

The **soft cut** `*->/2` is the variant that does not commit: `( Cond *-> Then ; Else )` runs `Then` for every solution of `Cond`, and `Else` only when `Cond` has none. It is not in the ISO standard, though several systems provide it, and it is the construct for "all the X such that, or a default if there are none":

```
?- ( member(X, [1, 2, 3]), X > 1
   *-> true ; X = none ).
X = 2 ;
X = 3.

?- ( member(X, [1, 2, 3]), X > 5
   *-> true ; X = none ).
X = none.
```

The idiom of this chapter's Diagnostic Use is the lookup with a default: a fact may or may not be present, and the predicate should bind a value either way, once. The shape is `( Lookup -> Value = Found ; Value = Default )`, deterministic whether the fact is present or not. The arithmetic analogue is a guarded operation that would otherwise raise an error, such as a predecessor that stops at zero:

```
% pred_or_zero(N, P): P is N - 1, or 0 for N = 0.
pred_or_zero(N, P) :-
    (   N > 0
    ->  succ(P, N)
    ;   P = 0
    ).

?- pred_or_zero(5, P).
P = 4.

?- pred_or_zero(0, P).
P = 0.
```

7.4 Negation as Failure

`\+/1` is **negation as failure**: `\+ Goal` succeeds if and only if `Goal` has no solution under the bindings in force when `\+` is called, and fails if `Goal` has at least one. In either case it binds nothing: the attempt to prove `Goal` is undone, bindings included, and only the outcome survives. The operator reads "not provable"; the spelling `not/1` in older code is the same predicate under a name the ISO standard does not use.

```
?- \+ member(c, [a, b]).
true.
```

```
?- \+ member(a, [a, b]).
false.

?- \+ 1 =:= 2.
true.
```

Whether `\+ Goal` means the logical negation of `Goal` depends on the variables of `Goal`. When `Goal` is ground, the two agree. When `Goal` has unbound variables, `\+ Goal` asks whether no instance of `Goal` is provable: the variables are existentially quantified inside the negation. This is the use Chapter 6 made in `next_thread/2`, where `Other` occurred only inside `\+` and the goal read "there is no `Other` strictly between"; the use is safe exactly because `Other` was not needed outside.

The unsafe use is a negated goal whose variable is bound later in the clause. The two queries below differ only in the order of their goals and have different answers:

```
?- \+ X = a, X = b.
false.

?- X = b, \+ X = a.
X = b.
```

The first fails because when `\+ X = a` is called `X` is unbound, `X = a` has a solution, and the negation fails. The second succeeds because `X` is already `b` when the negation is tried. Read declaratively both say "X is not `a` and X is `b`"; read procedurally the first asks whether there is no term `X` could be equal to `a`, which is false. A negated goal with a variable that is bound later in the clause is said to **flounder**, and the rule that avoids it is the one Chapter 3 gave for `\==` and Chapter 6 for arithmetic: a negation comes after the goals that bind the variables it tests, and a variable that occurs only inside the negation is the one case where an unbound variable under `\+` is correct. The inequality `\=/2` of Chapter 3 is `\+ X = Y` under its own name and follows the same rule.

A negation does not bind, which has a use of its own. `\+ \+ Goal` succeeds exactly when `Goal` has a solution and leaves every variable as it was: it tests whether a goal can succeed without committing to any of its bindings.

```
?- \+ \+ member(X, [a, b]).
true.

?- \+ \+ member(X, [a, b]), X = c.
X = c.
```

Negation as failure also fixes the meaning of absence. `\+ exception_record(105, _, _)` succeeds because no `exception_record/3` fact names thread 105. Under the **closed-world assumption** this reads "thread 105 raised no exception"; without it, it says only that the dump, as rendered into facts, records none. The difference is the one Chapter 1 drew between what a dump contains and what happened, and every recognizer built on a negated goal inherits it.

`forall/2` is a negation in disguise. `forall(Cond, Action)` succeeds if and only if every solution of `Cond` makes `Action` succeed, and it is defined as `\+ ( Cond, \+ Action )`: there is no solution of

`Cond` for which `Action` fails. Like `\+`, it binds nothing, and it succeeds trivially when `Cond` has no solution.

```
?- forall(member(X, [1, 2, 3]), X > 0).
true.

?- forall(member(X, [1, 2, 3]), X > 1).
false.

?- forall(member(X, []), X > 1).
true.
```

The predicate states an invariant over a knowledge base: every frame names a loaded module, every waiter is in a waiting state. A `forall/2` that fails names a violated invariant and nothing else; the violating instance is found by the conjunction `Cond, \+ Action`.

7.5 Loops

Prolog has no loop statement. Iteration over a sequence of solutions is backtracking, and iteration over a list is recursion. The **failure-driven loop** of Chapter 4 is a conjunction ending in `fail`: each solution of the goals before `fail` runs the side-effecting goals once, `fail` sends the search back for the next solution, and the loop ends when the generating goal has no more. The query fails as a whole, and every binding made in an iteration is undone before the next, so nothing computed in one iteration is available in the following one.

```
?- member(M, [ntdll, kernel32]),
   write(M), nl, fail.
ntdll
kernel32
false.

?- between(1, 3, N), S is N * N,
   write(S), nl, fail.
1
4
9
false.
```

`forall/2` is the same loop stated as a check. With a side-effecting action it visits every solution of the condition and succeeds at the end instead of failing, which makes it usable as a goal in a clause body without the `; true` a failure-driven loop needs there. Its bindings are undone in the same way, and the two idioms are interchangeable for printing.

```
?- forall(member(M, [ntdll, kernel32]),
   ( write(M), nl )).
ntdll
kernel32
true.
```

The remaining loop construct is `repeat/0`, a predicate with infinitely many solutions, each of them `true`. A conjunction `repeat, Goal, Test` runs `Goal` until `Test` succeeds, and the loop is then left by a cut. Its use is reading from a stream until the end of the file, where the number of iterations is not the number of solutions of any goal. The following predicate is illustrative and is not run here; Chapter 13 reads files.

```
% read_all_terms: read and print every term on the
% current input stream.
read_all_terms :-
    repeat,
    read(Term),
    (   Term == end_of_file
    ->  !
    ;   print(Term), nl,
        fail
    ).
```

A `repeat` loop whose body fails on every iteration spins forever; the construct is used in this book only where a stream supplies the end.

7.6 Exceptions

A goal that cannot be answered as asked has two ways out. It can fail, which says that the relation does not hold; or it can raise an **exception**, which abandons the search and passes a term, the **ball**, to the nearest enclosing handler. Chapter 6 met exceptions raised by arithmetic; this section raises and handles them in programs.

`throw/1` raises an exception with its argument as the ball. `catch/3` installs a handler: `catch(Goal, Catcher, Recovery)` calls `Goal`; if `Goal` succeeds or fails, `catch/3` does likewise and remains transparent to the choice points of `Goal`; if an exception is raised inside `Goal`, the bindings made since the call are undone, a copy of the ball is unified with `Catcher`, and on success `Recovery` is called in place of `Goal`. If the unification fails, the exception continues outwards to the next enclosing `catch/3`. An exception that reaches the toplevel is printed as an error message, and the query is abandoned.

```
?- catch(throw(my_ball), Ball, true).
Ball = my_ball.

?- catch(throw(my_ball), other, true).
ERROR: Unhandled exception: <...>
...

?- catch(member(X, [a, b]), _, true).
X = a ;
X = b.

?- catch(fail, _, true).
false.
```

In the second query the catcher does not unify and the exception passes through to the toplevel, which does not recognize the term and prints it as an unknown message; the third shows `catch/3` leaving the choice points of a goal that raised nothing.

The errors that built-in predicates raise have a standard shape. The ISO standard defines the ball of an error as `error(Formal, Context)`, where `Formal` describes the error and `Context` is implementation-defined; SWI-Prolog puts in `Context` a term naming the predicate that raised the error and, when available, a message. The formal terms met in this book:

Formal	Meaning
`instantiation_error`	an argument was unbound
`type_error(T, V)`	`V` is not of type `T`
`domain_error(D, V)`	`V` has the type, wrong value
`existence_error(K, V)`	no `K` named `V` exists
`evaluation_error(E)`	arithmetic failed, e.g. `zero_divisor`

A catcher of the form `error(Formal, _)` catches every standard error and binds the formal term, which is the form this book uses when the point of a transcript is the error and not the message:

```
?- catch(X is 1 / 0, error(E, _), true).
E = evaluation_error(zero_divisor).

?- catch(X is foo + 1, error(E, _), true).
E = type_error(evaluable, foo/0).

?- catch(atom_length(X, L), error(E, _), true).
E = instantiation_error.

?- catch(X is 1 / 0, error(_, Context), true).
Context = context((/)/2, _).
```

The last query shows the context half of the ball: SWI-Prolog records the predicate indicator of `//2`, and the second argument of the context, which would hold a message, is unbound here.

A program raises standard errors through `library(error)`, which is autoloaded. `must_be/2` checks that a term is of a type and raises the appropriate error when it is not: a type error for a term of the wrong kind, an instantiation error for an unbound one. The types it knows include `integer`, `atom`, `callable`, `list`, `boolean`, `positive_integer`, `nonneg`, `oneof(List)`, and `list(Type)`. The predicates `type_error/2`, `domain_error/2`, and `existence_error/2` raise the corresponding error terms directly, with the context left unbound.

```
?- catch(must_be(integer, a), error(E, _), true).
E = type_error(integer, a).

?- catch(must_be(integer, X), error(E, _), true).
E = instantiation_error.
```

```
?- catch(must_be(positive_integer, 0),
   error(E, _), true).
E = type_error(positive_integer, 0).

?- must_be(integer, a).
ERROR: Type error: `integer' expected, <...>
...

?- catch(existence_error(thread, 999),
   error(E, _), true).
E = existence_error(thread, 999).
```

The uncaught error of the fourth query is printed by the toplevel with a backtrace; the transcript shows its first line, since the frame numbers and variable names in the rest vary between runs. Chapter 2 promised `must_be/2` as the right response to a fact that violates the schema, and the reason is one of kind: a failure says "the relation does not hold for these arguments", which is a statement about the dump; an error says "these are not arguments the relation is defined on", which is a statement about the caller. A lookup of a thread that is not in the dump is of the first kind; a lookup of a thread identifier that is not an integer is of the second. This book follows that convention throughout.

An **existence error** has one case the toplevel treats specially. A goal for a predicate with no clauses raises `existence_error(procedure, Name/Arity)` when the flag `unknown` has its default value `error`. The toplevel, however, checks a query for undefined predicates before running it, so the error is seen in a `catch/3` only when the undefined goal is called from a clause:

```
% call_missing(E): E is the formal error term
% raised by a goal for an undefined predicate.
call_missing(E) :-
    catch(missing(1), error(E, _), true).

?- call_missing(E).
E = existence_error(procedure, missing/1).
```

Consulting the block prints no warning, since SWI-Prolog does not check at load time that every goal in a body has a definition; `check/0`, in Chapter 13, does.

Two further predicates belong with exceptions. `setup_call_cleanup/3` calls its first argument once, then its second, and runs its third, the cleanup, when the second has finished: on failure, on an exception, or when the goal has succeeded and left no choice point. Its use is to release a resource whatever happens to the goal in between; Chapter 13 uses it around every file it opens. `print_message/2` prints a message term to the error stream in the format of the toplevel's own messages, with a kind that selects the prefix: `error` prints the `ERROR:` lines a standard error term would produce, `warning` a `Warning:` prefix, `informational` nothing at the default verbosity. `catch(G, E, (print_message(error, E), fail))` reports an error and fails, and is the shape of the handler at the top of a script.

```
?- setup_call_cleanup(true, member(X, [a, b]),
   (write(cleanup), nl)).
X = a ;
cleanup
X = b.

?- setup_call_cleanup(true, fail,
   (write(cleanup), nl)).
cleanup
false.

?- catch(setup_call_cleanup(true, throw(oops),
   (write(cleanup), nl)), B, true).
cleanup
B = oops.

?- print_message(error,
   error(type_error(integer, a), _)).
ERROR: Type error: `integer' expected, <...>
true.
```

In the first query the cleanup runs when the second solution is produced, because `member/2` then leaves no choice point; it would run on the first solution had `once/1` enclosed the goal.

7.7 Diagnostic Use

The dump is consulted, and the classification of status codes from Section 7.1 is applied to its one exception record:

```
:- [dump1].

?- exception_record(109, Code, _),
   code_class(Code, C).
Code = 3221225477,
C = error.
```

The code is `0xc0000005`, the access violation, and its severity bits say `error`. A breakpoint, `0x80000003`, would be classed as a warning, which is the first thing an analyst checks when a dump of a process that did not crash shows an exception record; Chapter 23 returns to it under **False Positive Dump** (page 455).

The classification of threads is the chapter's main use. Each thread is in exactly one of the following conditions, tested in this order: it has an exception record; it waits for a critical section; it was running; it was suspended; it is an idle thread of the thread pool, recognizable by `TppWorkerThread` in its stack trace; or it is waiting for something else. The order matters, because the conditions are not exclusive: thread 109 has an exception record and a state of `waiting(exception)`, and the first test wins; a thread blocked on a critical section is in a waiting state too, and `blocked` is the more informative class.

```
% thread_class(TID, Class): Class is the one class
% of thread TID, by the first condition to hold.
thread_class(TID, Class) :-
    thread(TID, _),
    class_of(TID, Class).

% class_of(TID, Class): as thread_class/2, for a
% TID known to be a thread of the dump.
class_of(TID, Class) :-
    (   exception_record(TID, _, _)
    ->  Class = faulting
    ;   waits_for(TID, _, _)
    ->  Class = blocked
    ;   thread_state(TID, running)
    ->  Class = running
    ;   thread_state(TID, suspended)
    ->  Class = suspended
    ;   frame(TID, _, ntdll, 'TppWorkerThread', _)
    ->  Class = pool_idle
    ;   Class = waiting
    ).

?- thread_class(109, C).
C = faulting.

?- thread_class(101, C).
C = blocked.

?- thread_class(105, C).
C = pool_idle.

?- thread_class(100, C).
C = waiting.

?- thread_class(999, C).
false.
```

Every answer ends in `.`: the chain leaves no choice point, and `thread/2` with a bound first argument is selected by indexing. The query for thread 999 fails because the thread is not in the dump, which is the right answer to that question. With the identifier unbound, the predicate enumerates the threads, and the `;` after each answer belongs to `thread/2`, which still has facts to try:

```
?- thread_class(T, C).
T = 100,
C = waiting ;
T = 101,
C = blocked ;
T = 102,
C = running ;
T = 103,
C = blocked ;
T = 104,
C = blocked ;
T = 105,
C = pool_idle ;
T = 106,
C = pool_idle ;
```

```
T = 107,
C = running ;
T = 108,
C = suspended ;
T = 109,
C = faulting.

?- thread_class(T, blocked).
T = 101 ;
T = 103 ;
T = 104 ;
false.
```

The classes correspond to patterns of the memory analysis catalog that earlier chapters introduced: `running` is **Active Thread** (page 337), a thread that was executing when the dump was written; `blocked` is **Blocked Thread (Software)** (page 336), a thread stopped by a synchronization object that software controls, here a critical section; `suspended` is **Suspended Thread**, a thread the system will not schedule until it is resumed; and `waiting` and `pool_idle` together are **Passive Thread** (page 334), a thread waiting with nothing to show for it, of which the idle pool threads are the least informative kind. The `faulting` class has no single pattern name; it marks the thread whose stack trace is the **Exception Stack Trace** (page 358) of Chapter 19. The second query, with the class bound, keeps the threads whose first holding condition is the wait; its trailing `false.` is `thread/2` trying its last facts.

A lookup that should complain about a malformed argument wraps the same chain in a type check and an existence check, following Section 7.6: a non-integer is a type error, an integer that is not a thread of the dump is an existence error, and a thread of the dump gets its class.

```
% lookup_class(TID, Class): as thread_class/2 for
% a bound TID; an error on a non-integer or an
% absent TID.
lookup_class(TID, Class) :-
    must_be(integer, TID),
    (   thread(TID, _)
    ->  class_of(TID, Class)
    ;   existence_error(thread, TID)
    ).

?- lookup_class(102, C).
C = running.

?- catch(lookup_class(999, C), error(E, _), true).
E = existence_error(thread, 999).

?- catch(lookup_class(abc, C), error(E, _), true).
E = type_error(integer, abc).

?- lookup_class(999, C).
ERROR: thread `999' does not exist
...
```

A recognizer that enumerates threads wants `thread_class/2`, which fails quietly and serves as a generator; a report given a thread identifier by a user wants `lookup_class/2`, so that a typing error is reported rather than answered with silence.

The stack trace predicate of Chapter 5 is next. Its two clauses were exclusive, and the toplevel still reported a choice point at every answer, because the exclusiveness rested on a `\+` in the second clause that SWI-Prolog could not see from the heads. The if-then-else states the two cases as a choice:

```
% stack_trace(TID, Frames): Frames is the list of
% frame/5 terms of thread TID, top frame first.
stack_trace(TID, Frames) :-
    frames_from(TID, 0, Frames).

% frames_from(TID, N, Frames): Frames is the list
% of frames of TID numbered N and above.
frames_from(TID, N, Frames) :-
    (   frame(TID, N, M, F, O)
    ->  Frames = [frame(TID, N, M, F, O) | Rest],
        succ(N, N1),
        frames_from(TID, N1, Rest)
    ;   Frames = []
    ).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- stack_trace(100, _Frames), length(_Frames, N).
N = 7.

?- stack_trace(107, _Frames), length(_Frames, N).
N = 14.

?- stack_trace(999, Frames).
Frames = [].
```

The answers end in `.` where Chapter 5's ended in `;`. The lookup `frame(TID, N, M, F, O)` is the condition, so its first solution is taken and any choice point it left is discarded; and the output list is unified in the branches, after the condition, so the predicate is steadfast and leaves no choice point at any level. The recursive goal is then a true last call, and the predicate runs in constant stack for a trace of any depth. The flag `toplevel_print_anon` is set as in Chapter 5 so that the lists themselves are not printed.

The run of identical functions at the top of a stack trace, which Chapter 5 computed with an over-general `leading_run/3`, is now a deterministic count over frame numbers. The top frame fixes the function; the recursion counts the frames below it that name the same function and stops at the first that does not:

```
% top_run(TID, Function, Count): the top Count
% frames of TID all name Function, and frame Count
% does not.
top_run(TID, Function, Count) :-
    frame(TID, 0, _, Function, _),
    run_from(TID, 1, Function, 1, Count).

% run_from(TID, N, Function, Acc0, Count): Count
% is Acc0 plus the frames from N on that continue
% the run of Function.
run_from(TID, N, Function, Acc0, Count) :-
    (   frame(TID, N, _, Function, _)
    ->  Acc is Acc0 + 1,
        succ(N, N1),
        run_from(TID, N1, Function, Acc, Count)
    ;   Count = Acc0
    ).

?- top_run(107, F, N).
F = 'Node::Visit',
N = 8.

?- top_run(101, F, N).
F = 'NtWaitForAlertByThreadId',
N = 1.

?- top_run(T, F, N), N > 1.
T = 107,
F = 'Node::Visit',
N = 8 ;
false.
```

The second answer of Chapter 5, `N = 0` with the function unbound, is gone: the top frame binds the function before the count begins. The third query is the **Frame Regularity** (page 342) question of Chapters 3 and 5 asked over the whole dump, and it finds thread 107 and nothing else; its trailing `false.` is `frame/5` trying its remaining top frames. Chapter 18 formalizes the pattern with a threshold on the run length.

The lookups with defaults of Section 7.3 apply to facts the dump records for some threads and modules and not for others. The `last_error/2` relation has facts for six threads; a lookup for any other thread should say so rather than fail. The `company` key of `module_info/3` is present for five modules, and one of those records `'Unknown'`, which is what the debugger reports when a module's version resource names no vendor. The default must not be `'Unknown'`, because `thirdparty_hook` was examined and had no vendor, where `ntdll` was not examined. The default is an atom of its own, `unrecorded`:

```
% thread_last_error(TID, Code): Code is the last
% error code recorded for TID, or unrecorded.
thread_last_error(TID, Code) :-
    must_be(integer, TID),
    (   last_error(TID, Code0)
    ->  Code = Code0
    ;   Code = unrecorded
    ).

% module_company(Module, Company): Company is the
% vendor recorded for Module, or unrecorded.
module_company(Module, Company) :-
    must_be(atom, Module),
    (   module_info(Module, company, Company0)
    ->  Company = Company0
    ;   Company = unrecorded
    ).

?- thread_last_error(109, C).
C = 998.

?- thread_last_error(105, C).
C = unrecorded.

?- catch(thread_last_error(x, C),
   error(E, _), true).
E = type_error(integer, x).

?- module_company(thirdparty_hook, C).
C = 'Unknown'.

?- module_company(ntdll, C).
C = unrecorded.

?- module_company(appserver, C).
C = 'Example Corp'.
```

The last error of thread 109 is 998, `0x3e6`, which Windows names `ERROR_NOACCESS` and which is consistent with the access violation in the same thread; the toplevel prints it in decimal. The type checks fix the mode: these are lookups for a known thread or module, and a caller that wants to enumerate uses the underlying facts. Three values can now come out of a vendor lookup, a vendor name, `'Unknown'`, and `unrecorded`, and they say different things about the dump; Chapter 15 returns to the representation of unknowns in a schema.

Negation answers the questions of absence. Chapter 2 asked which frames are not well formed and deferred the answer; it is the frame relation in conjunction with the negated well-formedness test, and over `dump1.pl` it has no solution. The corresponding `forall/2` states the invariant positively:

```
% well_formed(Frame): Frame is a frame/5 term
% whose arguments have the kinds the schema
% prescribes.
well_formed(frame(TID, N, M, F, O)) :-
    integer(TID),
    integer(N),
    atom(M),
    atom(F),
    integer(O).

?- frame(T, N, M, F, O),
   \+ well_formed(frame(T, N, M, F, O)).
false.

?- forall(frame(T, N, M, F, O),
   well_formed(frame(T, N, M, F, O))).
true.

?- forall(frame(_, _, M, _, _), module(M, _, _)).
true.

?- forall(waits_for(T, _, _),
   thread_state(T, waiting(user_request))).
true.

?- forall(thread_state(T, waiting(_)),
   frame(T, 0, ntdll, _, _)).
false.

?- thread_state(T, waiting(_)),
   \+ frame(T, 0, ntdll, _, _).
T = 109.
```

The first four queries are schema checks of the kind a parser's output should pass before any analysis: every frame is well formed, every frame's module is loaded, every thread recorded as waiting for a critical section is in a waiting state. The fifth is a plausible invariant that does not hold, every waiting thread's top frame is in `ntdll`, and the sixth finds the thread that breaks it: 109 is waiting because of its exception, and its top frame is the faulting instruction in `dbclient`. A `forall/2` that fails and the conjunction that explains the failure are a pair, and the explanation is the more useful half.

Two further negated queries describe the dump by what is not in it:

```
% unreferenced_module(M): M is loaded and appears
% in no frame of any stack trace.
unreferenced_module(M) :-
    module(M, _, _),
    \+ frame(_, _, M, _, _).
```

```
% idle_thread(TID): TID has no exception, waits
% for nothing recorded, and was not running.
idle_thread(TID) :-
    thread(TID, _),
    \+ exception_record(TID, _, _),
    \+ waits_for(TID, _, _),
    \+ thread_state(TID, running).

?- unreferenced_module(M).
M = msvcp140 ;
M = thirdparty_hook.

?- idle_thread(T).
T = 100 ;
T = 105 ;
T = 106 ;
T = 108 ;
false.
```

Two of the ten modules have no frame on any stack: the C++ runtime library, which is unremarkable, and the hooking module, which is not, since Chapter 2 found it hooks two functions of `kernelbase` and the dump shows none of its code executing. Under the closed-world reading, `unreferenced_module/1` says that no thread was inside these modules when the dump was written; it does not say that their code did not run earlier, and Chapter 18 discusses what the raw stack can add under **Execution Residue (User Space)** (page 350). The idle threads are the four no other class claimed, and `idle_thread/1` agrees with `thread_class/2` on them by construction, since each `\+` is the failure of one condition in the chain.

Finally, the whole classification as a report, printed by `forall/2` with the column stops of Chapter 6:

```
% print_thread_classes: one line per thread with
% its class and CPU time.
print_thread_classes :-
    forall(thread_class(TID, Class),
           ( thread_time(TID, U, K),
             Ms is U + K,
             format("~d~t~6|~a~t~16|~t~d~6+ ms~n",
                    [TID, Class, Ms])
           )).

?- print_thread_classes.
100   waiting      160 ms
101   blocked       20 ms
102   running     9680 ms
103   blocked       40 ms
104   blocked       35 ms
105   pool_idle     25 ms
106   pool_idle     15 ms
107   running     4100 ms
108   suspended      0 ms
109   faulting      60 ms
true.
```

The report is deterministic: `forall/2` succeeds once and leaves nothing for the toplevel to report. The table is the `~*k` and `!runaway` output of a debugger reduced to one line per thread; the two running threads with their CPU times are the **Spiking Thread** (page 335) instances of Chapter 6, next to the eight threads that explain, between them, why the process was not making progress. Chapter 9 sorts such tables and totals their columns.

Chapter 8. Atoms, Strings, and Term Inspection

The facts of a dump are terms, and the chapters so far have treated their atoms as indivisible names: `dbclient` is compared with `dbclient` and with nothing else. The debugger that produced them did not see it that way. It printed `dbclient!Buffer::Copy+0x1a`, one piece of text in which a module, a function, a C++ class, a method, and a hexadecimal offset are joined by punctuation, and the parser of Chapter 12 will have to take such text apart to produce `frame/5` facts at all. A trace message is text from the start, and nearly every question about it, which words it contains, which number follows “ms”, whether it begins with “Error”, is a question about its parts. This chapter covers the built-in predicates that look inside atoms and strings: conversion between the four representations of text, concatenation and splitting, substring search in every mode, case conversion, and the reading and writing of terms as text. It then covers the predicates that look inside compound terms: `functor/3`, `arg/3`, and `=../2`, which make a program able to inspect a term whose shape it did not know when it was written.

The diagnostic use is symbol decoding. A predicate `symbol_parts/4` relates a symbol in the debugger’s `module!function+0xoffset` notation to its three components, in both directions; a companion splits a C++ qualified name at its scope operator; and the frames the debugger could not resolve, which the schema records with the function `'Unknown'`, are recognized and traced to the modules without symbol files that caused them.

8.1 Text in Four Forms

Chapter 2 gave the four representations of text in SWI-Prolog: the **atom**, an interned name; the **string**, an unshared sequence of characters that is a term kind of its own; the **code list**, a list of integer character codes; and the **char list**, a list of one-character atoms. The ISO standard has atoms, code lists, and char lists; strings are SWI-Prolog’s addition, and the default meaning of double quotes as strings is SWI-Prolog’s choice, as the flag `double_quotes` of Chapter 2 records. Every text predicate in this chapter is one of two kinds: it converts between representations, or it operates on one representation and has a twin for the other.

The conversion predicates are named by their two ends. `atom_codes/2` and `atom_chars/2` relate an atom to its code list or char list; `string_codes/2` and `string_chars/2` do the same for strings; `atom_string/2` relates an atom to a string; `char_code/2` relates one char to its code. Each works in both directions and is deterministic.

```
?- atom_codes(ntdll, Cs).
Cs = [110, 116, 100, 108, 108].

?- atom_chars(ntdll, Cs).
Cs = [n, t, d, l, l].

?- atom_codes(A, [0'n, 0't]).
A = nt.
```

```
?- atom_string(A, "ntdll").
A = ntdll.

?- string_chars(S, [a, b]).
S = "ab".

?- char_code(C, 0'a).
C = a.
```

Numbers have their own conversions, because the text `'26'` and the integer 26 are different terms and a parser meets the first when it needs the second. `atom_number/2` relates an atom to the number it spells, `number_codes/2` a code list, and `number_string/2` a string. All three read the text with the syntax of Prolog numbers, so `'0x1a'` is 26, `'1_000'` is 1000, and `'3.5'` is a float; `atom_number/2` fails on text that is not a number, where `number_codes/2` raises a syntax error. The text-to-number direction is the one a parser uses, and the number-to-text direction produces the shortest decimal spelling.

```
?- atom_number('0x1a', N).
N = 26.

?- atom_number('1a', N).
false.

?- atom_number(A, 26).
A = '26'.

?- number_string(N, "44").
N = 44.

?- number_string(N, "44 ms").
false.
```

The last query fails because `"44 ms"` is not the spelling of a number; extracting the 44 from it is the business of Section 8.2. `atom_length/2` and `string_length/2` give the number of characters. SWI-Prolog is lenient about the first argument of the text predicates: where an atom is expected, a string or a number is accepted and its text used, so `atom_length(12345, L)` gives 5 rather than a type error. In ISO mode the type error is raised. The leniency is convenient and hides a mistake in a schema, since a predicate that was meant to receive atoms will not notice strings; Chapter 2's type tests are the way to notice.

```
?- atom_length('Buffer::Copy', L).
L = 12.

?- atom_length(12345, L).
L = 5.

?- string_length("Query end", L).
L = 9.
```

Case conversion is `upcase_atom/2` and `downcase_atom/2` for atoms, `string_upper/2` and `string_lower/2` for strings. Module names in the shared schema are lowercase atoms, and a parser

that meets `NTDLL` in one place and `ntdll` in another normalizes with `downcase_atom/2`; Chapter 9 lowercases every word of a trace before counting it, so that `Error` and `error` are one word.

```
?- downcase_atom('NTDLL', D).
D = ntdll.

?- string_lower("Buffer::Copy", L).
L = "buffer::copy".
```

8.2 Concatenation, Substrings, and Splitting

`atom_concat/3` relates two atoms to their concatenation, and `string_concat/3` does the same for strings. Both accept any atomic terms as the parts, and both work backwards: with the whole bound and one part bound, the other part is found, and with the whole bound and both parts unbound, every split is enumerated, as `append/3` did for lists in Chapter 5.

```
?- atom_concat(dbclient, '!Buffer::Copy', A).
A = 'dbclient!Buffer::Copy'.

?- atom_concat(X, '!Buffer::Copy',
   'dbclient!Buffer::Copy').
X = dbclient.

?- atom_concat(X, Y, abc).
X = '',
Y = abc ;
X = a,
Y = bc ;
X = ab,
Y = c ;
X = abc,
Y = ''.

?- string_concat(X, "end", "Query end").
X = "Query ".
```

`atomic_list_concat/2` concatenates a list of atomic terms into one atom, and `atomic_list_concat/3` inserts a separator between them. The three-argument form has a **split mode**: with the list unbound, a non-empty separator, and the whole bound, it splits the atom at every occurrence of the separator and yields the parts as atoms. It is the quickest way to split at a fixed delimiter, and it yields atoms, which is what module and function names should be.

```
?- atomic_list_concat([dbclient, '!', 'Copy'], A).
A = 'dbclient!Copy'.

?- atomic_list_concat([a, 1, "s", 2.5], A).
A = 'a1s2.5'.

?- atomic_list_concat([a, b, c], '::', A).
A = 'a::b::c'.
```

```
?- atomic_list_concat(Parts, '::', 'a::b::c').
Parts = [a, b, c].

?- atomic_list_concat(Parts, ' ', 'end: 1 row').
Parts = ['end:', '1', row].
```

The general substring predicate is `sub_atom/5`. `sub_atom(Atom, Before, Length, After, Sub)` holds when `Sub` is the part of `Atom` that has `Before` characters before it, `Length` characters in it, and `After` characters after it. Any of the five arguments may be bound, and the predicate enumerates whatever is left open. The modes that matter in practice have names: a **prefix** test or extraction binds `Before` to 0; a **suffix** binds `After` to 0; a **search** binds `Sub` and leaves the positions open, which finds every occurrence from left to right; and an **extraction** binds `Before` and `Length`, or `Before` and `After`, and is deterministic. `sub_string/5` is the same relation on strings and yields strings. In SWI-Prolog both accept either kind of text as their first argument and yield the kind their name says.

```
?- sub_atom(abcde, 1, 3, A, Sub).
A = 1,
Sub = bcd.

?- sub_atom(abcde, B, 2, 0, Sub).
B = 3,
Sub = de.

?- sub_atom('dbclient!Copy', B, _, A, '!').
B = 8,
A = 4 ;
false.

?- sub_atom('a::b::c', B, 2, A, '::').
B = 1,
A = 4 ;
B = 4,
A = 1 ;
false.

?- sub_atom('RtlUserThreadStart', 0, 3, _, P).
P = 'Rtl'.

?- sub_atom('BaseThreadInitThunk', _, _, 0,
   'Thunk').
true.

?- sub_string("Query end: 1 row, 44 ms", B, _, 0,
   "ms").
B = 21.
```

The third query is the search mode: `'!'` occurs once, at position 8, with 4 characters after it, and the trailing `false.` is the search continuing to the end of the atom in case of another occurrence. The fourth finds both occurrences of `':::'` in order. With the position of a delimiter known, the parts on either side of it are two deterministic extractions, `sub_atom(Atom, 0, B, _, Left)` for the text before it and `sub_atom(Atom, _, A, 0, Right)` for the text after, which is the mechanism of `symbol_parts/4` in Section 8.5. A search with `Sub` bound and all positions anonymous is a

containment test; `sub_atom(F, _, _, _, 'Wait')` succeeds for every function name with `Wait` somewhere in it.

Splitting at a class of characters is `split_string/4`. `split_string(String, SepChars, PadChars, Parts)` splits `String` at every character in `SepChars`, then removes from both ends of every part any characters in `PadChars`, and yields the parts as strings. The two character sets are independent. With separators and no padding, two adjacent separators produce an empty part between them. When the padding set contains the separators, SWI-Prolog treats a run of adjacent separators as one and drops separators at the ends of the string, so that no empty parts arise; this is the documented behavior and the usual way to split at any of several characters.

```
?- split_string("end: 1 row, 44 ms", " ", "", P).
P = ["end:", "1", "row,", "44", "ms"].

?- split_string("end: 1 row, 44 ms", " ", ",:",
   P).
P = ["end", "1", "row", "44", "ms"].

?- split_string("end: 1 row, 44 ms", " ,:", "",
   P).
P = ["end", "", "1", "row", "", "44", "ms"].

?- split_string("end: 1 row, 44 ms", " ,:", " ,:",
   P).
P = ["end", "1", "row", "44", "ms"].

?- split_string("  padded  ", "", " ", P).
P = ["padded"].

?- split_string("Request 1 done: 200, 309 ms",
   ",", " ", P).
P = ["Request 1 done: 200", "309 ms"].
```

The second query is one idiom for words: split at spaces and strip punctuation from the ends of each part. The third shows the empty parts that arise when `:` and `,` are separators and are followed by a space, and the fourth removes them by padding with the same characters. The fifth, with no separators, only strips; the sixth splits a trace message into its two clauses at the comma. The 44 in `"44 ms"` is now reachable: split, take the part, and convert with `number_string/2`.

```
?- split_string("end: 1 row, 44 ms", " ", ",:",
   P), nth1(4, P, S), number_string(N, S).
P = ["end", "1", "row", "44", "ms"],
S = "44",
N = 44.
```

Chapter 12 does the same with a grammar, which is the right tool when the text has a structure deeper than a sequence of fields; for fixed layouts, `split_string/4` and `number_string/2` are enough and are what Chapter 9 and Part IV use. One more string predicate belongs here. `string_code/3` relates a one-based index into a string to the character code at that position, and is the way to look at a single character without converting the whole string to a list.

```
?- string_code(1, "abc", C), char_code(Ch, C).
C = 97,
Ch = a.

?- string_code(I, "abc", 0'c).
I = 3.

?- string_code(4, "abc", C).
false.
```

8.3 Terms as Text

A term can be written as text and read back from text. `term_to_atom/2` relates a term to the atom that `writeq/1` would print for it, and in the other direction parses the atom as Prolog syntax; `term_string/2` does the same with a string. Reading uses the full Prolog reader, with operators and `0x` numbers, so the text of a fact in a `.pl` file is readable in this way, and the variables in the text become fresh variables, with repeated names shared.

```
?- term_to_atom(frame(109, 0, 'Copy', 0x1a), A).
A = 'frame(109,0,\'Copy\',26)'.

?- term_string(T, "frame(109, 0, 'Copy', 0x1a)").
T = frame(109, 0, 'Copy', 26).

?- term_string(T, "foo(X, Y, X)").
T = foo(_A, _, _A).
```

The quotes inside the atom in the first answer are escaped with backslashes because the toplevel writes the atom quoted; the atom itself contains plain single quotes. In the third answer the toplevel names the shared variable `_A` and the singleton `_`. `read_term_from_atom/3` reads a term from an atom with the options of `read_term/2`, which Chapter 13 covers with files; it is the predicate to use when the variable names in the text are wanted, through the option `variable_names(Bindings)`.

```
?- read_term_from_atom('f(X, Y)', T,
   [variable_names(Vs)]).
T = f(_A, _B),
Vs = ['X'=_A, 'Y'=_B].
```

The other direction, building text from a term under the control of a format string, is `format/3` with a **sink** as its first argument. `format(atom(A), Format, Args)` runs `format/2` and collects the output in the atom `A`; `string(S)`, `codes(Cs)`, and `chars(Cs)` collect it in the other representations. The hexadecimal formatting of Chapter 6 thereby produces text a program can keep rather than only print, and the debugger's symbol notation is one format string:

```
?- format(atom(A), "~a!~a+0x~16r",
   [dbclient, 'Buffer::Copy', 26]).
A = 'dbclient!Buffer::Copy+0x1a'.
```

```
?- format(string(S), "~`0t~16r~16|",
   [0x7ffb_8d10_31aa]).
S = "00007ffb8d1031aa".

?- format(atom(A), "0x~16r", [0xc0000005]),
   atom_number(A, N).
A = '0xc0000005',
N = 3221225477.
```

The third query goes round the circle: a code is formatted as hexadecimal text with a `0x` prefix, and the text reads back as the same integer. `with_output_to/2` generalizes the sink to any goal: `with_output_to(string(S), Goal)` runs `Goal` and collects everything it writes to the current output in `S`, which turns the printing predicates of earlier chapters into text producers without rewriting them.

```
?- with_output_to(string(S),
   forall(member(X, [a, b]), (write(X), nl))).
S = "a\nb\n".
```

Four printing predicates complete the set. `print/1` writes a term as `writeq/1` does, quoted, but through the **portray** hook, a user-defined `portray/1` that a program may define to print certain terms its own way; the toplevel prints answers through the same hook, and `~p` in a format string calls it. `write_term/2` is the predicate the others are defined in terms of, with an option list: `quoted(true)` quotes atoms and strings that need it, `max_depth(N)` abbreviates nested terms and long lists beyond depth `N` with `...`, which is how the toplevel keeps a long answer on one line, and `ignore_ops(true)` writes operators in canonical form. `portray_clause/1` writes a term as a clause of a program, with the layout of this book, variables renamed to `A`, `B`, and a terminating period; it is the predicate that writes a knowledge base to a file in a form `consult/1` reads back, which Chapter 30 uses to save generated facts.

```
?- print([a, 'B c', "str", 1.5]), nl.
[a,'B c',"str",1.5]
true.

?- write_term([1, 2, 3, 4], [max_depth(3)]), nl.
[1,2|...]
true.

?- write_term(1 + 2 * 3, [ignore_ops(true)]), nl.
+(1,*(2,3))
true.

?- portray_clause(frame(101, 4, cachelib,
   'Unknown', 0x2a10)).
frame(101, 4, cachelib, 'Unknown', 10768).
true.
```

```
?- portray_clause((a :- b, c ; d)).
a :-
    (   b,
        c
    ;   d
    ).
true.
```

Variables in output are a last detail. A term with variables is written with internal names such as `_G123` that differ between runs, as Chapter 6 noted for `format/2`. `numbervars/3` binds each variable of a term to a term `'$VAR'(N)` with consecutive integers from a start value, and the writing predicates print `'$VAR'(0)` as `A`, `'$VAR'(1)` as `B`, and so on, when their `numbervars(true)` option is in force, as it is for `print/1`, `writeq/1`, and `portray_clause/1`. The bindings are ordinary bindings and are undone on backtracking.

```
?- T = f(X, Y, X), numbervars(T, 0, End),
   print(T), nl.
f(A,B,A)
T = f(A, B, A),
X = A,
Y = B,
End = 2.

?- T = f(X, Y), numbervars(T, 0, _),
   write_canonical(T), nl.
f('$VAR'(0),'$VAR'(1))
T = f(A, B),
X = A,
Y = B.
```

8.4 Term Inspection

The predicates of Section 8.3 look at a term from the outside, as text. Three built-in predicates look at a compound term from the inside, by its functor and arguments, and they are what a program uses to handle a term whose shape it did not know when it was written: a fact of a schema that may change, an argument that may be any of several relations, a term to be printed field by field.

`functor/3` relates a term to the name and arity of its principal functor. With the term bound it decomposes; with the name and arity bound it constructs a term with that functor and fresh variables as arguments, which is the template of Chapter 3 built from its description. An atomic term has itself as name and 0 as arity.

```
?- functor(frame(109, 0, dbclient, 'Copy', 26),
   N, A).
N = frame,
A = 5.

?- functor(T, frame, 5).
T = frame(_, _, _, _, _).
```

```
?- functor(ntdll, N, A).
N = ntdll,
A = 0.

?- functor([a], N, A).
N = '[|]',
A = 2.
```

The last answer is the list functor of Chapter 2 seen through `functor/3`. `arg/3` relates an argument position, from 1, to the argument at that position of a compound term. With the position bound it is a deterministic lookup; with the position unbound, SWI-Prolog enumerates the arguments in order, which the ISO standard does not require. It raises a type error on an atomic first argument and fails on a position outside the arity.

```
?- arg(3, frame(109, 0, dbclient, 'Copy', 26), X).
X = dbclient.

?- arg(N, frame(109, 0, dbclient, 'Copy', 26),
   dbclient).
N = 3 ;
false.

?- arg(N, f(a, b), X).
N = 1,
X = a ;
N = 2,
X = b.

?- arg(6, frame(109, 0, dbclient, 'Copy', 26), X).
false.
```

`=../2`, pronounced **univ**, relates a term to the list of its name followed by its arguments, in both directions. It is the most general of the three, and the one to use when a term is to be rebuilt with different arguments or when its arguments are to be processed as a list.

```
?- frame(109, 0, dbclient, 'Copy', 26) =.. L.
L = [frame, 109, 0, dbclient, 'Copy', 26].

?- T =.. [frame, 109, 0].
T = frame(109, 0).

?- ntdll =.. L.
L = [ntdll].

?- T =.. [foo].
T = foo.
```

A term with a name and no arguments is an atom, in standard Prolog, and `functor(T, foo, 0)` and `T =.. [foo]` both yield the atom `foo`. SWI-Prolog 7 and later also has compound terms with zero arguments, written `foo()`, which are not atoms; they are constructed and decomposed with `compound_name_arity/3` and `compound_name_arguments/3`, which otherwise behave as `functor/3` and `=../2` do and are preferred in new code that must handle both. The shared schema has no zero-arity compounds, and this book uses the standard predicates.

One predicate in this family is to be avoided. `setarg/3` replaces an argument of a compound term in place, destructively; the change is undone on backtracking, and `nb_setarg/3` makes a change that is not. Both break the rule that a term, once built, never changes, on which everything in Chapters 3 and 4 depends: a term that has been passed to another goal, stored in a list, or shared by `copy_term/2`, which does not copy ground subterms, changes everywhere at once.

```
?- X = f(a, b), setarg(1, X, c).
X = f(c, b).

?- X = f(a, b), copy_term(X, Y), setarg(1, Y, z).
X = Y, Y = f(z, b).
```

The second query is the warning in one line: `Y` was meant to be a copy, `copy_term/2` shared the ground term, and `setarg/3` changed `X` through it. The uses of destructive assignment in SWI-Prolog are in low-level library code and in the global variables of Chapter 11; an analysis program builds new terms and leaves old ones alone.

The three inspection predicates make a schema addressable by field name. The following relates a field name of the `frame/5` schema to its position, and then any frame to any of its fields, without a clause per field:

```
% frame_field(Name, Index): argument Index of a
% frame/5 term holds the field Name.
frame_field(tid, 1).
frame_field(number, 2).
frame_field(module, 3).
frame_field(function, 4).
frame_field(offset, 5).

% field_of(Frame, Name, Value): Value is the field
% Name of the frame/5 term Frame.
field_of(Frame, Name, Value) :-
    functor(Frame, frame, 5),
    frame_field(Name, Index),
    arg(Index, Frame, Value).

?- field_of(frame(109, 0, dbclient, 'Copy', 26),
   module, M).
M = dbclient.

?- field_of(frame(109, 0, dbclient, 'Copy', 26),
   Name, 26).
Name = offset.

?- field_of(frame(109, 0, dbclient, 'Copy', 26),
   Name, V).
Name = tid,
V = 109 ;
Name = number,
V = 0 ;
Name = module,
V = dbclient ;
```

```
Name = function,
V = 'Copy' ;
Name = offset,
V = 26.
```

The `functor/3` goal rejects any term that is not a `frame/5`. With the name unbound the predicate enumerates the fields, which is how a generic printer walks a term it was not written for; Chapter 15 extends `frame_field/2` to every relation of the schema and Chapter 29 uses the same three predicates to walk goals in a meta-interpreter.

8.5 Diagnostic Use

The dump is consulted. The debugger that produced it prints a frame as a module, a function, and an offset joined by `!` and `+0x`, and the `frame/5` facts hold the three parts separately. The relation between the two forms is `symbol_parts/4`: `symbol_parts(Symbol, Module, Function, Offset)` holds when the atom `Symbol` is the debugger's text for the frame with those parts. The decoding direction finds the `!`, splits the atom there, and splits the right half again at `+0x`; the hexadecimal digits after it are prefixed with `0x` and read with `atom_number/2`.

```
:- [dump1].

% symbol_parts(Symbol, Module, Function, Offset):
% Symbol is the atom Module!Function+0xOffset, or
% Module+0xOffset for a frame without a symbol.
symbol_parts(Symbol, Module, Function, Offset) :-
    (   sub_atom(Symbol, B, _, A, '!')
    ->  sub_atom(Symbol, 0, B, _, Module),
        sub_atom(Symbol, _, A, 0, Rest),
        name_offset(Rest, Function, Offset)
    ;   name_offset(Symbol, Module, Offset),
        Function = 'Unknown'
    ).

% name_offset(Text, Name, Offset): Text is
% Name+0xOffset, or Name alone with Offset 0.
name_offset(Text, Name, Offset) :-
    (   sub_atom(Text, B, _, A, '+0x')
    ->  sub_atom(Text, 0, B, _, Name),
        sub_atom(Text, _, A, 0, Hex),
        atom_concat('0x', Hex, HexAtom),
        atom_number(HexAtom, Offset)
    ;   Name = Text,
        Offset = 0
    ).
```

Each `sub_atom/5` search is the condition of an if-then-else, so the first occurrence of the delimiter is taken and the predicate is deterministic. The two halves are then extracted by position: `B` characters from the start for the left part, `A` characters up to the end for the right. The second clause of the choice handles the form the debugger uses when it has no symbol for a frame, `module+0xoffset` with no `!`, and records the function as `'Unknown'`, which is the schema's convention for the same situation. A text with neither delimiter is a bare name with offset 0.

```
?- symbol_parts('dbclient!Buffer::Copy+0x1a',
   M, F, O).
M = dbclient,
F = 'Buffer::Copy',
O = 26.

?- symbol_parts('ucrtbase!thread_start<...>+0x42',
   M, F, O).
M = ucrtbase,
F = 'thread_start<...>',
O = 66.

?- symbol_parts('cachelib+0x2a10', M, F, O).
M = cachelib,
F = 'Unknown',
O = 10768.

?- symbol_parts('appserver!wmain', M, F, O).
M = appserver,
F = wmain,
O = 0.
```

The template parameter in `thread_start<...>` and the scope operator in `Buffer::Copy` pass through untouched, because neither contains the delimiters. A function name containing `+`, as a C++ `operator+` would, is split at the first `+0x`; the predicate takes the first occurrence, and a name with `+0x` inside it would be misread. Whether the debugger ever prints such a name depends on its symbol demangling, which these facts do not record.

The encoding direction is `format/3` with an atom sink, with the `'Unknown'` convention reversed:

```
% frame_symbol(Frame, Symbol): Symbol is the
% debugger's text for the frame/5 term Frame.
frame_symbol(frame(_, _, M, F, O), Symbol) :-
    (   F == 'Unknown'
    ->  format(atom(Symbol), "~a+0x~16r", [M, O])
    ;   format(atom(Symbol), "~a!~a+0x~16r",
               [M, F, O])
    ).

% print_frame(TID, N): print frame N of TID as the
% debugger would, with its number.
print_frame(TID, N) :-
    frame(TID, N, M, F, O),
    frame_symbol(frame(TID, N, M, F, O), Symbol),
    format("~`0t~d~2| ~a~n", [N, Symbol]).

?- frame(101, 4, M, F, O),
   frame_symbol(frame(101, 4, M, F, O), S).
M = cachelib,
F = 'Unknown',
O = 10768,
S = 'cachelib+0x2a10'.
```

```
?- forall(frame(T, N, M, F, O),
   ( frame_symbol(frame(T, N, M, F, O), S),
     symbol_parts(S, M, F, O) )).
true.

?- forall(frame(109, N, _, _, _),
   print_frame(109, N)).
00 dbclient!Buffer::Copy+0x1a
01 dbclient!Connection::Receive+0x8c
02 appserver!Session::Fetch+0x44
03 appserver!Worker::Run+0x120
04 ucrtbase!thread_start<...>+0x42
05 kernel32!BaseThreadInitThunk+0x14
06 ntdll!RtlUserThreadStart+0x21
true.
```

The second query is the round trip over all 78 frames of the dump: every frame's text decodes back to the frame's own module, function, and offset, which is the test that the two predicates are inverses on this data. The third prints the stack trace of the faulting thread in the form Chapter 5 could print only without offsets; it is the `k` output of the debugger stripped of its address columns, which the facts do not hold.

Function names from C++ code carry a structure of their own. `Buffer::Copy` is the method `Copy` of the class `Buffer`, and a nested namespace produces `a::b::c`. The scope operator that matters is the last one, since everything before it is the scope and everything after it the name; `sub_atom/5` finds the occurrences from left to right, so the last is the one whose right-hand part contains no further `::`, and `once/1` takes the first solution that satisfies the test:

```
% scoped_name(Function, Scope, Name): Function is
% Scope::Name, split at the last scope operator.
scoped_name(Function, Scope, Name) :-
    once(( sub_atom(Function, B, 2, A, '::'),
           sub_atom(Function, _, A, 0, Name),
           \+ sub_atom(Name, _, _, _, '::')
         )),
    sub_atom(Function, 0, B, _, Scope).

?- scoped_name('Buffer::Copy', S, N).
S = 'Buffer',
N = 'Copy'.

?- scoped_name('a::b::c', S, N).
S = 'a::b',
N = c.

?- scoped_name(wmain, S, N).
false.

?- frame(T, 0, _, F, _),
   scoped_name(F, Class, Method).
T = 102,
F = 'Table::Find',
Class = 'Table',
Method = 'Find' ;
```

```
T = 107,
F = 'Node::Visit',
Class = 'Node',
Method = 'Visit' ;
T = 109,
F = 'Buffer::Copy',
Class = 'Buffer',
Method = 'Copy'.
```

A name without a scope operator makes the predicate fail, which is the right answer: `wmain` is a C function and has no scope. The last query finds the three threads whose top frame is a C++ method, and the classes, `Table`, `Node`, `Buffer`, are the vocabulary in which Chapter 18 describes what each thread was doing. The split is textual, and so is its limit: a template argument list containing `::`, as in `vector<std::string>::push_back`, would be split at its last `::` and give a wrong scope, and a parser that must handle such names needs the grammar of Chapter 12.

The frames the debugger could not resolve are the last use. The schema records them with the function `'Unknown'`, and Chapter 2 found one, frame 4 of thread 101 in `cachelib`. The relation between such frames and the symbol information of their modules is checked over the whole dump:

```
% unknown_frame(TID, N, Module): frame N of TID is
% in Module and has no resolved function.
unknown_frame(TID, N, Module) :-
    frame(TID, N, Module, 'Unknown', _).

?- unknown_frame(T, N, M).
T = 101,
N = 4,
M = cachelib.

?- unknown_frame(T, N, M),
   module_info(M, symbols, S).
T = 101,
N = 4,
M = cachelib,
S = none ;
false.

?- forall(unknown_frame(_, _, M),
   module_info(M, symbols, none)).
true.

?- frame(_, _, M, F, _),
   module_info(M, symbols, none), F \== 'Unknown'.
false.
```

The dump has one unresolved frame, its module has no symbols, every unresolved frame is in a module without symbols, and no frame in a module without symbols has a name: the two facts of the schema that describe the same situation, `'Unknown'` in `frame/5` and `none` in `module_info/3`, agree with each other. The situation is the **No Component Symbols** (page 431) pattern that Chapter 2 named from `module_info/3` alone: a module for which the debugger had no symbol file, so that every frame in it is an offset from the module base and the stack trace has a gap wherever the

module appears. The unresolved frame is where the pattern shows in a stack trace, and `symbol_parts/4` is where the parser of Chapter 12 will recognize it, since `cachelib+0x2a10` has no `!`. Thread 101 is blocked in a critical section, and the frame that called into the critical section code is the one the debugger could not name; whatever `cachelib` was doing with the lock, the dump does not say, and an analyst who needs to know must obtain the module's symbols or read its code at the offset.

A second degree of the same problem is in `dbclient`, whose `symbols` value is `export`: the debugger had the module's export table and nothing else, and names a frame by the nearest preceding exported function, which may be the right function or may be an unrelated export some distance before it. The five `dbclient` frames in the dump have plausible names and small offsets, and nothing in the facts can confirm them. Chapter 22 treats both degrees under the pattern, with `export` as the weaker form of missing symbols, and Chapter 19 keeps the caveat in view when it names `Buffer::Copy` as the faulting function.

```
?- frame(T, N, dbclient, F, O),
   format("~d/~d ~a+0x~16r~n", [T, N, F, O]),
   fail.
103/4 Connection::Send+0x58
104/5 Connection::OnReply+0x1f
104/6 Reader::Loop+0xa4
109/0 Buffer::Copy+0x1a
109/1 Connection::Receive+0x8c
false.
```

Chapter 9. Collecting Solutions and Aggregation

A query so far has produced its answers one at a time, and a program that needed all of them has either printed them in a failure-driven loop or, as Chapter 6 did to total the CPU time of a process, walked the facts in an order it had to construct itself. Neither gives the program the answers as a term it can go on computing with. This chapter covers the predicates that do: `findall/3`, which collects every solution of a goal into a list; `bagof/3` and `setof/3`, which do the same but group the solutions by the variables the goal leaves free; `aggregate_all/3` and `aggregate/3` from `library(aggregate)`, which count, sum, and take extremes without building the list; `library(solution_sequences)`, which filters, orders, and limits the answers of a goal while it still produces them one at a time; and the pair convention `Key-Value` with `library(pairs)`, `keysort/2`, and `predsort/3`, which sort collected answers by keys and by orders the standard order does not express.

The diagnostic uses are the collections an analyst reads as a whole. The **Stack Trace Collection (Unmanaged Space)** (page 330) of the memory analysis catalog is every thread's stack trace at once, a `findall/3` over threads; the **Module Variety** (page 427) of a process is a count over its modules; the total CPU time of Chapter 6 becomes one aggregation goal, and the table of shares is sorted by it; and the **Vocabulary Index** (page 487) of a software trace, the words of its messages with their frequencies, is a histogram over `trace1.pl` built with the `split_string/4` of Chapter 8.

9.1 Collecting All Solutions

`findall/3` is the basic all-solutions predicate. `findall(Template, Goal, List)` calls `Goal`, and for each of its solutions makes a copy of `Template` under that solution's bindings; `List` is the list of these copies, in the order the solutions were found. The predicate always succeeds exactly once: when `Goal` has no solution, `List` is `[]`. It is the collecting counterpart of the failure-driven loop, which visits the same solutions in the same order and keeps none of them.

The examples of this chapter's first sections use a miniature trace of eight messages, each with a number, a thread, and a kind. The flag of Chapter 5 that hides the bindings of variables beginning with an underscore is set for the whole chapter, since collections are long and the queries bind them only to derive something shorter.

```
% event(N, TID, Kind): message N, on thread TID,
% of the given Kind.
event(1, 100, start).
event(2, 101, query).
event(3, 102, query).
event(4, 101, build).
event(5, 101, done).
event(6, 102, warning).
event(7, 102, warning).
event(8, 100, error).
```

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- findall(K, event(_, 101, K), Ks).
Ks = [query, build, done].

?- findall(N-K, event(N, 102, K), Ps).
Ps = [3-query, 6-warning, 7-warning].

?- findall(K, event(_, 105, K), Ks).
Ks = [].

?- findall(K, event(_, _, K), _Ks),
   length(_Ks, N).
N = 8.
```

The template may be any term. The third query collects pairs, and the last counts the solutions by collecting them and taking the length, which is the simplest way to count and the one `aggregate_all/3` improves on in Section 9.3. The order of the list is the order of the search tree of Chapter 4, which for facts is the order of the clauses; Chapter 4 said that order must never be assumed to mean anything, and a program that needs the list in a particular order sorts it, as Section 9.5 does.

Two properties of `findall/3` follow from the copying. The variables of the template are **copied** for each solution, so the elements of the list do not share variables with the template or with each other, and the bindings `Goal` made are undone when the collection is complete: after `findall/3` the variables of the template and the goal are as unbound as they were before. Chapter 3 promised this as the reason bindings made inside `findall/3` are not visible outside it.

```
?- findall(X, member(X, [a, b]), L), X = c.
X = c,
L = [a, b].

?- findall(X-Y, member(X, [a, b]), L).
L = [a-_, b-_].
```

In the first query `X` is free after the collection and can be bound to `c`; in the second, `Y` was never bound, and each copy of the template has its own fresh variable in its place, which the toplevel prints as `_`. A template variable that is meant to be shared across the elements is therefore not possible with `findall/3`, and a program that needs it binds the variable before the call.

`findall/4` takes a tail: `findall(Template, Goal, List, Tail)` makes `List` the solutions followed by `Tail`, which is the difference list of Chapter 5 and lets several collections be concatenated without `append/3`.

```
?- findall(K, event(_, 101, K), L, [end]).
L = [query, build, done, end].
```

```
?- findall(K, event(_, 101, K), L, T),
   findall(K, event(_, 102, K), T, []).
L = [query, build, done, query, warning, warning],
T = [query, warning, warning].
```

9.2 Bags, Sets, and Free Variables

`bagof/3` and `setof/3` collect solutions as `findall/3` does, with two differences that make them a different kind of predicate. First, they fail when the goal has no solution, where `findall/3` yields `[]`. Second, they treat the **free variables** of the goal, those that occur in the goal but neither in the template nor under an explicit quantifier, as a grouping key: one answer is produced for each distinct binding of the free variables, with the list of the solutions that share that binding, and the answers are enumerated on backtracking in the standard order of the key.

```
?- bagof(K, event(N, 101, K), Ks).
N = 2,
Ks = [query] ;
N = 4,
Ks = [build] ;
N = 5,
Ks = [done].

?- bagof(K, event(_, 105, K), Ks).
false.
```

The first query groups by `N`, the message number, which is free, and since each message has its own number there are three groups of one. This is rarely wanted, and the **existential quantifier** `^/2` is the way to say that a variable is not a grouping key: `Var^Goal` is `Goal` with `Var` bound inside the collection and invisible outside it, and several variables are quantified as `V1^V2^Goal`.

```
?- bagof(K, N^event(N, 101, K), Ks).
Ks = [query, build, done].

?- bagof(K, N^event(N, T, K), Ks).
T = 100,
Ks = [start, error] ;
T = 101,
Ks = [query, build, done] ;
T = 102,
Ks = [query, warning, warning].
```

The second query is the grouping that is wanted: with `N` quantified and `T` free, one answer per thread, each with the kinds of that thread's messages in message order. The groups are produced as the toplevel asks for them, which makes `bagof/3` a generator of groups rather than a collector of one list, and the predicate has as many answers as there are threads.

An anonymous variable is a free variable like any other. `bagof(K, event(_, 101, K), Ks)` groups by the anonymous first argument exactly as the query with `N` did, and produces the same three groups, because each `_` is a distinct variable that the goal binds. This is the usual surprise with

`bagof/3`: a goal written with _ for the arguments that do not matter, which works in `findall/3`, groups by them in `bagof/3`. Every argument that is not a grouping key must be quantified by name.

```
?- bagof(K, event(_, 101, K), Ks).
Ks = [query] ;
Ks = [build] ;
Ks = [done].
```

`setof/3` is `bagof/3` followed by `sort/2`: the list of each group is sorted in the standard order of terms and its duplicates removed, and the predicate fails on no solution and groups by free variables in the same way.

```
?- setof(K, N^T^event(N, T, K), Ks).
Ks = [build, done, error, query, start, warning].

?- setof(T, N^K^event(N, T, K), Ts).
Ts = [100, 101, 102].

?- setof(K, N^event(N, T, K), Ks).
T = 100,
Ks = [error, start] ;
T = 101,
Ks = [build, done, query] ;
T = 102,
Ks = [query, warning].
```

The second query is the set of threads that have messages, and the third the set of kinds per thread; the two `warning` messages of thread 102 have become one element. `setof/3` with every variable but the template's quantified is the idiom for "the distinct values of", and is the one of the three collectors whose result does not depend on the order of the clauses.

The choice between the three follows from the differences. `findall/3` when the solutions are wanted as they come, with `[]` for none; `bagof/3` when the solutions are to be grouped by some of the goal's arguments and the absence of solutions is a failure the caller should see; `setof/3` when the result is a set. In practice `findall/3` and `setof/3` do most of the work, and `bagof/3` is used for its grouping.

9.3 Aggregation

Collecting a list in order to take its length, its sum, or its largest element is wasteful when the list is large and is wordy when it is small. `library(aggregate)`, which is autoloaded, provides `aggregate_all/3`: `aggregate_all(Spec, Goal, Result)` runs `Goal` to exhaustion and computes from its solutions whatever `Spec` asks for. The specifications are `count`, the number of solutions; `sum(E)`, the sum of the values of the expression `E`; `max(E)` and `min(E)`, the largest and smallest value, which must be numbers; `max(E, W)` and `min(E, W)`, the same with a **witness** term `W` recorded alongside, with the result in the form `max(Value, Witness)`; and `bag(E)` and `set(E)`, the list of values as `findall/3` and `setof/3` would give it. Like `findall/3`, `aggregate_all/3` succeeds once and does not group: free variables in the goal are bound and unbound as the goal is run, and nothing

is grouped by them. On no solution, `count` is 0, `sum` is 0, `bag` is `[]`, `set` is `[]`, and `max` and `min` fail, since there is no largest element of nothing.

```
?- aggregate_all(count, event(_, 101, _), N).
N = 3.

?- aggregate_all(count, event(_, 105, _), N).
N = 0.

?- aggregate_all(sum(N), event(N, 101, _), S).
S = 11.

?- aggregate_all(max(N), event(N, 102, _), M).
M = 7.

?- aggregate_all(max(N), event(N, 105, _), M).
false.

?- aggregate_all(max(N, K), event(N, 101, K), M).
M = max(5, done).

?- aggregate_all(set(K), event(_, 102, K), Ks).
Ks = [query, warning].
```

The witness form answers the question that `max/1` alone cannot: not only the number of the last message of thread 101 but what kind it was. The expression of `max/1` must evaluate to a number; `max(N-K)` would raise a type error, because a pair is not a number, and the witness form is the way to carry a term along with the number. Several specifications may be combined in one compound term, `aggregate_all(r(count, max(N)), Goal, r(C, M))`, and the goal is run once for all of them.

`aggregate/3` is to `aggregate_all/3` what `bagof/3` is to `findall/3`: it groups by the free variables of the goal, produces one result per group on backtracking, and fails when there is no solution. The quantifier `^` applies as in Section 9.2.

```
?- aggregate(count, N^K^event(N, T, K), C).
T = 100,
C = 2 ;
T = 101,
C = 3 ;
T = 102,
C = 3.

?- aggregate(count, N^event(N, T, K), C).
T = 100,
K = error,
C = 1 ;
T = 100,
K = start,
C = 1 ;
T = 101,
K = build,
C = 1 ;
```

```
T = 101,
K = done,
C = 1 ;
T = 101,
K = query,
C = 1 ;
T = 102,
K = query,
C = 1 ;
T = 102,
K = warning,
C = 2.
```

The first query is the number of messages per thread, and the second, with `K` also free, the number per thread and kind: the two-dimensional count that a histogram over two attributes needs, produced one cell at a time in the standard order of the key. Section 9.6 uses `aggregate/3` for messages per component.

`forall/2` of Chapter 7 is an aggregation in this sense, with a Boolean result: `forall(C, A)` succeeds when the count of solutions of `C` for which `A` fails is zero. The two formulations are interchangeable, and the aggregation has the advantage when the number of violations is wanted rather than their absence.

```
?- forall(event(_, T, _), T >= 100).
true.

?- aggregate_all(count, (event(_, T, _), T < 100),
   N).
N = 0.
```

9.4 Solution Sequences

The collectors of the previous sections turn a sequence of solutions into a term and give up the one-at-a-time behavior of the toplevel. `library(solution_sequences)`, also autoloaded, transforms the sequence and keeps the behavior: each of its predicates takes a goal and behaves as a goal with the same solutions, filtered, reordered, or cut short. They are the Prolog counterparts of the `DISTINCT`, `ORDER BY`, `LIMIT`, and `OFFSET` clauses of a database query, and the resemblance is deliberate.

`distinct/1` succeeds for each solution of its goal whose bindings differ from every earlier one; `distinct/2` compares on a witness term instead of the whole goal. The caution of Section 9.2 returns in a different form: `distinct(event(_, T, _))` finds eight distinct solutions, because the anonymous variables are part of the goal and are bound differently each time, where `distinct(T, event(_, T, _))` finds three.

```
?- distinct(T, event(_, T, _)).
T = 100 ;
T = 101 ;
T = 102 ;
```

```
false.

?- distinct(K, event(_, _, K)).
K = start ;
K = query ;
K = build ;
K = done ;
K = warning ;
K = error.
```

The trailing `false.` of the first query is `distinct/2` finding that the remaining solutions of its goal are all repeats; the second query gives the kinds in the order of first occurrence, which `setof/3` could not, since it sorts. `limit/2` produces at most the given number of solutions and `offset/2` skips that many; together they page through a sequence. `order_by/2` produces the solutions sorted by a list of `asc(V)` and `desc(V)` keys; it collects all the solutions internally before producing the first, so it does not terminate on a goal with infinitely many. `call_nth/2` succeeds for the solution with the given ordinal.

```
?- limit(2, event(N, T, K)).
N = 1,
T = 100,
K = start ;
N = 2,
T = 101,
K = query.

?- limit(2, offset(1, event(N, _, _))).
N = 2 ;
N = 3.

?- order_by([desc(N)], event(N, 102, K)).
N = 7,
K = warning ;
N = 6,
K = warning ;
N = 3,
K = query.

?- call_nth(event(N, _, _), 3).
N = 3.
```

`group_by/4` is `bagof/3` with the quantification made explicit: `group_by(By, Template, Goal, Bag)` yields one solution per distinct value of `By`, with the bag of `Template` instances for it, and every other variable of the goal is treated as quantified. The groups come in the standard order of `By`.

```
?- group_by(T, K, event(_, T, K), Ks).
T = 100,
Ks = [start, error] ;
T = 101,
Ks = [query, build, done] ;
T = 102,
Ks = [query, warning, warning].
```

The predicates compose. `limit(3, order_by([desc(Ms)], cpu_time(T, Ms)))` is the top three threads by CPU time, and Section 9.6 writes it over the dump. The composition is of goals, and the result is still a goal: it can be the condition of `forall/2`, the goal of `findall/3`, or a line of a toplevel query, which is what makes these predicates convenient in interactive analysis.

9.5 Pairs and Sorting

A collected list is sorted by `msort/2` and `sort/2` in the standard order of its elements, which Chapter 5 gave. When the elements are to be sorted by one part of themselves, the convention is to make that part a **key** and the rest a **value** and to join them with the operator `-/2` into a **pair** `Key-Value`. Nothing enforces the convention; the pair is an ordinary compound term with principal functor `-/2`, and the standard order compares pairs by key first, which is what makes sorting a list of pairs sort it by key.

`library(pairs)` has the predicates that build and take apart such lists. `pairs_keys_values/3` relates a list of pairs to the list of its keys and the list of its values, in every direction; `pairs_keys/2` and `pairs_values/2` give one of the two; `transpose_pairs/2` swaps keys and values and sorts the result by the new keys, keeping duplicates.

```
?- pairs_keys_values(P, [a, b], [1, 2]).
P = [a-1, b-2].

?- pairs_keys_values([a-1, b-2], Ks, Vs).
Ks = [a, b],
Vs = [1, 2].

?- pairs_values([a-1, b-2], Vs).
Vs = [1, 2].

?- transpose_pairs([a-1, b-2, c-1], T).
T = [1-a, 1-c, 2-b].
```

`keysort/2` sorts a list of pairs by key, in the standard order, keeping duplicates and keeping the original order of pairs with equal keys: it is a **stable sort**, and the stability is what distinguishes it from `msort/2`, which would also compare the values. `sort/4` of Chapter 5 generalizes it: `sort(1, @>=, Pairs, Sorted)` sorts by key descending and keeps duplicates, where `sort(1, @>, ...)` would keep one pair per key.

```
?- keysort([b-1, a-2, b-0, a-1], S).
S = [a-2, a-1, b-1, b-0].

?- msort([b-1, a-2, b-0, a-1], S).
S = [a-1, a-2, b-0, b-1].

?- sort(1, @>=, [b-1, a-2, b-0, a-1], S).
S = [b-1, b-0, a-2, a-1].

?- sort(1, @>, [b-1, a-2, b-0, a-1], S).
S = [b-1, a-2].
```

An order the standard order does not express, such as by count descending and then by name ascending, needs `predsort/3`. `predsort(Pred, List, Sorted)` sorts with a user-supplied comparison: `Pred` is called with three arguments, an order variable to be bound to `<`, `=`, or `>`, and the two elements to compare, exactly as `compare/3` of Chapter 2 would be. Elements the comparison declares equal are merged into one, as `sort/2` merges duplicates, so a comparison that says `=` for two elements that are not the same term loses one of them, and a comparison that is meant to keep every element must break ties on something that is distinct.

```
% by_count_desc(Order, K1-N1, K2-N2): order pairs
% by count descending, then by key ascending.
by_count_desc(Order, K1-N1, K2-N2) :-
    compare(Order0, N2, N1),
    (   Order0 == (=)
    ->  compare(Order, K1, K2)
    ;   Order = Order0
    ).

?- predsort(by_count_desc,
   [query-2, build-1, warning-2, done-1], S).
S = [query-2, warning-2, build-1, done-1].
```

The counts are compared in reverse, `N2` before `N1`, which produces the descending order, and the keys break the tie in the ordinary direction. Since keys in a histogram are distinct, no two elements compare equal and none is lost.

A histogram is the common case of all this. The values are collected, sorted so that equal values are adjacent, and the runs of equal values are counted; the result is a list of `Value-Count` pairs that `predsort/3` then orders by count. The run counting is a recursion of the kind Chapter 5 wrote, with the if-then-else of Chapter 7 keeping it deterministic:

```
% run_counts(Sorted, Counts): Counts is a list of
% X-N pairs, one per run of N identical elements X
% in the sorted list Sorted.
run_counts([], []).
run_counts([X | Xs], [X-N | Counts]) :-
    run_length(Xs, X, 1, N, Rest),
    run_counts(Rest, Counts).

% run_length(Xs, X, N0, N, Rest): N is N0 plus the
% leading elements of Xs identical to X; Rest is
% what follows them.
run_length(Xs, X, N0, N, Rest) :-
    (   Xs = [Y | Xs1],
        Y == X
    ->  N1 is N0 + 1,
        run_length(Xs1, X, N1, N, Rest)
    ;   N = N0,
        Rest = Xs
    ).
```

```
% histogram(Values, Counts): Counts is the list of
% Value-Count pairs of Values, most frequent
% first.
histogram(Values, Counts) :-
    msort(Values, Sorted),
    run_counts(Sorted, Counts0),
    predsort(by_count_desc, Counts0, Counts).

?- histogram([b, a, c, a, b, a], H).
H = [a-3, b-2, c-1].

?- findall(K, event(_, _, K), _Ks),
   histogram(_Ks, _H),
   forall(member(K-N, _H),
          format("~a ~d~n", [K, N])).
query 2
warning 2
build 1
done 1
error 1
start 1
true.
```

The same result can be had from `aggregate_all/3` by counting each distinct value separately, with `setof/3` or `aggregate_all(set(V), ...)` to enumerate the values and `aggregate_all(count, ...)` for each; that formulation runs the goal once per distinct value and is the right one when the values are few and the goal is cheap, and `histogram/2` is the right one when the goal is expensive, since it runs it once. Section 9.6 uses both.

9.6 Diagnostic Use

Both shared files are consulted.

```
:- [dump1].
:- [trace1].
```

A stack trace is a collection in the sense of this chapter. The frames of a thread are all the `frame/5` facts with that thread identifier, and `findall/3` collects them; the frame numbers then order them, by `sort/4` on the second argument, so that the result does not depend on the order of the facts in the file. The predicate is the third definition of `stack_trace/2` in this book, after the recursion of Chapter 5 and the deterministic recursion of Chapter 7, and the shortest:

```
% stack_trace(TID, Frames): Frames is the list of
% frame/5 terms of thread TID, top frame first.
stack_trace(TID, Frames) :-
    findall(frame(TID, N, M, F, O),
            frame(TID, N, M, F, O),
            Frames0),
    sort(2, @=<, Frames0, Frames).
```

```
?- stack_trace(109, _Frames), length(_Frames, N).
N = 7.

?- stack_trace(999, Frames).
Frames = [].
```

The mode is `+TID`: with the identifier unbound, `findall/3` would collect the frames of every thread into one list, and `sort/4` would interleave them by frame number. The `@=<` keeps frames with equal numbers, of which a well-formed dump has none for one thread; the sort is a defense against disorder in the file, not a filter.

The **Stack Trace Collection (Unmanaged Space)** (page 330) is the stack traces of all threads of a process taken together, which is what the debugger's `~*k` prints and what an analyst reads first, looking for the threads that stand out from the rest rather than at any one thread. It is a `findall/3` over threads, with each element a pair of the thread identifier and its stack trace:

```
% stack_trace_collection(Collection): Collection
% is the list of TID-Frames pairs, one per thread.
stack_trace_collection(Collection) :-
    findall(TID-Frames,
            ( thread(TID, _),
              stack_trace(TID, Frames)
            ),
            Collection).

% print_depths(Collection): one line per thread
% with the depth of its stack trace.
print_depths(Collection) :-
    forall(member(TID-Frames, Collection),
           ( length(Frames, Depth),
             format("~d~t~6|~t~d~4+~n",
                    [TID, Depth])
           )).

?- stack_trace_collection(_C), length(_C, N).
N = 10.

?- stack_trace_collection(_C), print_depths(_C).
100      7
101     10
102      7
103     10
104     10
105      4
106      4
107     14
108      5
109      7
true.

?- stack_trace_collection(_C),
   aggregate_all(sum(D),
    ( member(_-Fs, _C), length(Fs, D) ), Total).
Total = 78.
```

The collection holds the 78 frames of the dump in ten lists, and the depths alone already separate the threads: the two idle pool threads at four frames, the suspended housekeeping thread at five, the workers at seven and ten, and thread 107 at fourteen, the deepest by a margin that is the eight `Node::Visit` frames of Chapter 7. Chapter 18 treats the collection as a pattern and derives from it the **Stack Trace Signature** (page 338) and **Stack Trace Set** (page 341) patterns, which compare the stack traces with each other.

The modules of a dump are the second collection. **Module Variety** (page 427) is the catalog's name for the number and diversity of the modules loaded in a process, which an analyst reads from `lm` as a first measure of how much third-party code is involved. The count is one aggregation, and the related counts, how many distinct modules appear on any stack and how many vendors are represented, are two more:

```
% module_variety(Count): Count is the number of
% modules loaded in the process.
module_variety(Count) :-
    aggregate_all(count, module(_, _, _), Count).

?- module_variety(N).
N = 10.

?- aggregate_all(set(M), frame(_, _, M, _, _),
   _Ms), length(_Ms, N).
N = 8.

?- aggregate_all(set(C),
   module_info(_, company, C), Cs).
Cs = ['CacheWorks', 'Example Corp', 'Unknown'].

?- setof(M, T^N^F^O^frame(T, N, M, F, O), _Ms),
   module_variety(Total), length(_Ms, OnStack),
   Idle is Total - OnStack.
Total = 10,
OnStack = 8,
Idle = 2.
```

Ten modules, eight of them on some stack, three vendors recorded and five modules with no vendor recorded at all. The two idle modules are the ones Chapter 7 found with negation; here they are a subtraction of two counts. Chapter 22 refines the count into the ratio of non-Microsoft modules, using the `company` key, and treats **Module Variety** (page 427) with the other module patterns. The `setof/3` in the last query quantifies the four arguments it does not collect, as Section 9.2 requires; the `aggregate_all(set(M), ...)` form two queries earlier says the same with anonymous variables, which is why this book prefers it.

The frequency of modules over all frames is the histogram of Section 9.5, and it says which modules the threads were actually in:

```
?- findall(M, frame(_, _, M, _, _), _Ms),
   histogram(_Ms, _H),
   forall(member(M-N, _H),
```

```
          format("~a~t~14|~t~d~4+~n", [M, N])).
ntdll           28
appserver       13
reportengine    13
kernel32        10
ucrtbase         6
dbclient         5
kernelbase       2
cachelib         1
true.
```

The system library leads, as it does in every dump of a waiting process, because each blocked thread contributes four `ntdll` frames of critical section code and every thread ends in `RtlUserThreadStart`; the application's own modules follow. A histogram of the top frames alone, `frame(_, 0, M, _, _)`, would say where the threads were stopped rather than where they had been, and Chapter 18 uses both.

CPU time is the third collection, and it closes the account Chapter 6 opened. The total over all threads, which Chapter 6 computed by walking the threads in order of their identifiers, is one `sum` aggregation, and the thread with the largest time is one `max` with a witness:

```
% cpu_time(TID, Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% total_cpu(Total): the CPU time of all threads.
total_cpu(Total) :-
    aggregate_all(sum(Ms), cpu_time(_, Ms),
                  Total).

% cpu_share(TID, Percent): the CPU time of TID as
% a percentage of the total.
cpu_share(TID, Percent) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Percent is 100.0 * Ms / Total.

?- total_cpu(Total).
Total = 14135.

?- aggregate_all(max(Ms, TID), cpu_time(TID, Ms),
   M).
M = max(9680, 102).

?- limit(3, order_by([desc(Ms)],
   cpu_time(TID, Ms))).
Ms = 9680,
TID = 102 ;
Ms = 4100,
TID = 107 ;
Ms = 160,
TID = 100.
```

The total is the 14135 of Chapter 6 from three clauses fewer, and `total_cpu/1` is now deterministic where the walk left a choice point. The last query is the top three threads by CPU time, and it is the shape of every top-N question: order the solutions of a goal by a key and take the first few. The table of shares that Chapter 6 printed in thread order is printed in order of share with the same `order_by/2` as the condition of `forall/2`:

```
% print_cpu_table: one row per thread, largest
% share first.
print_cpu_table :-
    forall(order_by([desc(Ms)], cpu_time(T, Ms)),
           print_cpu_row(T, Ms)).

% print_cpu_row(TID, Ms): one table row for TID.
print_cpu_row(TID, Ms) :-
    cpu_share(TID, Percent),
    format("~d~t~6|~t~D~8+ ms~3+~t~1f~8+%~n",
           [TID, Ms, Percent]).

?- print_cpu_table.
102      9,680 ms    68.5%
107      4,100 ms    29.0%
100        160 ms     1.1%
109         60 ms     0.4%
103         40 ms     0.3%
104         35 ms     0.2%
105         25 ms     0.2%
101         20 ms     0.1%
106         15 ms     0.1%
108          0 ms     0.0%
true.
```

The two **Spiking Thread** (page 335) instances of Chapter 6 are the first two rows, and the gap to the third row is the whole of the pattern. `cpu_share/2` recomputes the total for every row, which costs ten aggregations over ten facts and is invisible here; on a dump with thousands of threads it would be computed once and passed down, which is the accumulator discipline of Chapter 5 applied to a report.

Grouping has a use on the dump too. The threads waiting for each critical section are the groups of `waits_for/3` by address, and `bagof/3` produces them:

```
?- bagof(TID, waits_for(TID, critical_section, A),
   TIDs), format("~16r~n", [A]).
7ff61d13a0c0
A = 140695026507968,
TIDs = [101, 104] ;
7ff61d13a140
A = 140695026508096,
TIDs = [103].
```

Two threads wait for the first section and one for the second; the first group is the beginning of a queue on one lock, which Chapter 20 treats under **Blocked Queue** (page 396) when the queue is long enough to matter, and the two groups together are the two halves of the deadlock of Chapter

4. The address is printed in hexadecimal by `format/2` and in decimal by the toplevel, as Chapter 6 explained.

The last collection is over the trace. The **Vocabulary Index** (page 487) of a software trace is the list of the distinct words occurring in its messages with the number of occurrences of each, which gives an analyst the trace's vocabulary at a glance: the frequent words name the routine activity, and the rare ones point at what happened once. A word here is a maximal run of characters between spaces, commas, colons, and equals signs, lowercased, which is the `split_string/4` of Chapter 8 with the same characters as separators and padding:

```
% message_word(Word): Word is a lowercased word of
% the text of some message of the trace.
message_word(Word) :-
    msg(_, _, _, _, _, Text),
    split_string(Text, " ,:=", " ,:=", Parts),
    member(Part, Parts),
    string_lower(Part, Word).

% word_count(Word, Count): Word occurs Count times
% in the trace.
word_count(Word, Count) :-
    aggregate_all(count, message_word(Word),
                  Count).

% vocabulary(Index): Index is the list of
% Word-Count pairs of the trace, most frequent
% first.
vocabulary(Index) :-
    findall(Word, message_word(Word), Words),
    histogram(Words, Index).

?- aggregate_all(count, message_word(_), N).
N = 378.

?- vocabulary(_V), length(_V, N).
N = 123.

?- vocabulary(_V),
   forall(limit(8, member(W-N, _V)),
          format("~t~d~4| ~s~n", [N, W])).
  28 request
  21 worker
  13 ms
  11 received
  11 to
  10 start
   9 dispatched
   9 get
true.

?- word_count("error", N).
N = 4.

?- word_count("session", N).
N = 6.
```

The 71 messages contain 378 words, 123 of them distinct. The head of the index is the request lifecycle, `request`, `worker`, `received`, `dispatched`, and the unit `ms` from the completion messages; `error` occurs four times, in the three queue-depth messages and the buffer copy failure. `word_count/2` is the `aggregate_all/3` formulation of Section 9.5, which runs `message_word/1` once per word asked about and is the right one for a few words; `vocabulary/1` is the histogram, which runs it once for all. The two agree, and the first query, which counts words without distinguishing them, is the sum of the index's counts.

The tail of the index is the diagnostically interesting end. The words that occur once are the numbers of individual requests and sizes, which is noise, and the words that name events that happened once, which is not. The following keeps the words of the second kind, those beginning with a letter, and prints them three to a line with the column stops of Chapter 6; the stop of each word depends on its position in the line, so it is taken from the argument list with `~*|`, as `~*c` took its count:

```
% rare_word(Word): Word occurs once in the trace
% and begins with a lowercase letter.
rare_word(Word) :-
    vocabulary(Index),
    member(Word-1, Index),
    string_code(1, Word, C),
    C >= 0'a,
    C =< 0'z.

% print_columns(Words): print Words three to a
% line in 16-character columns.
print_columns(Words) :-
    forall(nth1(I, Words, W),
           ( Stop is 16 * ((I - 1) mod 3 + 1),
             format("~s~t~*|", [W, Stop]),
             (   I mod 3 =:= 0
             ->  nl
             ;   true
             )
           )),
    nl.

?- vocabulary(_V),
   findall(W, member(W-1, _V), _Once),
   length(_Once, N).
N = 62.

?- findall(W, rare_word(W), _Rare),
   length(_Rare, N).
N = 31.

?- findall(W, rare_word(W), _Rare),
   print_columns(_Rare).
buffer          busy            c
cache           capacity        code
commit          configuration   connected
connecting      copy            cpu
exception       failed          initialized
loading         lookup          pool
post            receive         reply
```

```
send            started         starting
stopping        thread          threads
unhandled       version         watchdog
workers
true.
```

Half of the vocabulary occurs once. With the numbers, paths, and hexadecimal codes filtered out, the remainder is the list of things that happened once in this trace: the service started and stopped, a configuration was loaded, a connection was made, a lookup and a commit were begun, a watchdog ran, a copy failed, and an exception went unhandled. Several of these are the **Significant Event** (page 566) instances of Chapter 27, found here by frequency alone and without knowing in advance what to look for, which is the use of a **Vocabulary Index** (page 487). The splitting is crude, and the index shows it: `c` is the drive letter of a path split at its colon, and `\app\app.cfg` is the rest of it. Chapter 24 refines the word definition and treats the pattern with the other structural patterns of a trace; the collection and counting machinery does not change.

The messages per component, finally, is `aggregate/3` grouping by a free variable, with the other five arguments quantified:

```
?- aggregate(count,
   N^T^P^Th^Tx^msg(N, T, P, Th, C, Tx), Count).
C = appserver,
Count = 47 ;
C = cachelib,
Count = 1 ;
C = dbclient,
Count = 12 ;
C = reportengine,
Count = 11.
```

The application server wrote two thirds of the trace and the cache library one line, which is the first fact about the trace's components; Chapter 24 calls the division of a trace by component a **Trace Partition** (page 489) and Chapter 26 asks which components are in the foreground of the activity and which in the background. Both begin with this count.

Chapter 10. Higher-Order Predicates and Libraries

Every goal in the bodies written so far names its predicate in the source. The filter of Chapter 5 that removed a module from a list had its test, `M0 \== M`, written into its third clause, and a filter with a different test would have been a different predicate with the same three-clause shape. Chapter 9 printed every table with a `forall/2` whose action was a `format/2` goal written out for that table. This chapter covers the predicates that take a goal as an argument and so let the shape of a computation be written once: `call/N`, which runs a goal given as a term with further arguments added; `maplist/2` to `maplist/5`, `include/3`, `exclude/3`, `partition/4`, and `foldl/4` to `foldl/6` from `library(apply)`, which map, filter, and accumulate over lists with a goal supplied by the caller; and `library(yall)`, which lets that goal be written in place as a lambda expression instead of as a named predicate. It then covers three libraries of data structures that an analysis program reaches for once its data outgrows lists of facts: `library(assoc)`, balanced trees keyed by terms; SWI-Prolog's dicts, records with named fields; and `library(ugraphs)`, directed graphs with reachability and topological sorting.

The diagnostic uses close two accounts. The wait-for relation among the threads of `dump1.pl`, which Chapter 4 followed by hand and Chapter 5 made terminate with a visited list, is a directed graph, and `library(ugraphs)` answers the questions a **Wait Chain (Critical Sections)** (page 384) and a **Deadlock (Critical Sections)** (page 388) pose about it, reachability and the presence of a cycle, without a recursive predicate being written. The CPU table that Chapter 6 printed in thread order and Chapter 9 sorted by share is built again with `foldl/4`, which computes the total once and threads a running sum through the rows, giving the table the cumulative column that shows how much of the process the two **Spiking Thread** (page 335) instances account for.

10.1 Goals as Arguments

A goal is a term, and a term can be an argument. The predicate `call/1` takes a term and runs it as a goal: `call(G)` succeeds exactly when `G` does, with the same bindings and the same choice points. Chapter 7 met it inside `once/1`, `ignore/1`, and `forall/2`, and Chapter 9 inside `findall/3`; each of those predicates receives a goal as an argument and runs it with `call/1` or its equivalent. Such a predicate is a **meta-predicate**, and the argument that holds the goal is a **meta-argument**, written with the mode `:` in the SWI-Prolog manual and in this book.

`call/N` for `N` from 2 to 8 extends the goal before running it: `call(G, A1, ..., An)` adds the arguments `A1` to `An` at the end of the argument list of `G` and calls the result. The term `G` is then a **closure**, a goal missing its last arguments, and the word is used in that sense throughout the SWI-Prolog manual; it is not the closure of functional programming, since nothing is captured but the arguments already present. The dump is consulted here, since the examples of this chapter are best shown over its relations:

```
:- [dump1].
:- use_module(library(apply)).
:- use_module(library(yall)).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- call(thread_time, 102, U, K).
U = 8450,
K = 1230.

?- call(thread_time(102), U, K).
U = 8450,
K = 1230.

?- call(thread_time(102, U), K).
U = 8450,
K = 1230.

?- G = thread_time(102), call(G, U, K).
G = thread_time(102),
U = 8450,
K = 1230.
```

The four queries run the same goal, `thread_time(102, U, K)`, from four closures of decreasing incompleteness: the bare predicate name, the name with one argument, with two, and the one-argument closure held in a variable. The arguments supplied by `call/N` are always appended, never inserted, which is why the predicates of this chapter are documented with the position of the argument they add. The two directives load `library(apply)` and `library(yall)` explicitly rather than relying on autoloading; Section 10.5 gives the reason.

`call/N` is related to the `=../2` of Chapter 8, which also builds goals from parts, and differs from it in what it is for. `=../2` constructs a term from a name and a complete list of arguments; `call/N` takes a term that exists and adds to it. A closure stored in a variable is passed around as data and completed where it is used, and the errors are those of any goal: a variable is an instantiation error, a non-callable term a type error, an unknown predicate an existence error for the indicator of the completed goal.

```
?- G =.. [thread_time, 102, U, K], call(G).
G = thread_time(102, 8450, 1230),
U = 8450,
K = 1230.

?- catch(call(_, 1), error(E, _), true).
E = instantiation_error.

?- catch(call(1), error(E, _), true).
E = type_error(callable, 1).

?- catch(call(foo, 1), error(E, _), true).
E = existence_error(procedure, foo/1).
```

A program defines its own meta-predicates with `call/N` in the body. The following takes a test, a closure to be completed with one argument, and collects the threads of the dump that pass it:

```
% threads_where(Test, TIDs): TIDs is the list of
% threads of the dump for which call(Test, TID)
% succeeds.
threads_where(Test, TIDs) :-
    findall(TID,
            ( thread(TID, _),
              call(Test, TID)
            ),
            TIDs).

% running(TID): TID was running.
running(TID) :-
    thread_state(TID, running).

% blocked(TID): TID waits for a critical section.
blocked(TID) :-
    waits_for(TID, critical_section, _).

% cpu_time(TID, Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% cpu_above(Ms, TID): TID consumed more than Ms.
cpu_above(Ms, TID) :-
    cpu_time(TID, Total),
    Total > Ms.

?- threads_where(running, Ts).
Ts = [102, 107].

?- threads_where(blocked, Ts).
Ts = [101, 103, 104].

?- threads_where(cpu_above(1000), Ts).
Ts = [102, 107].

?- threads_where(cpu_above(100), Ts).
Ts = [100, 102, 107].
```

The tests are ordinary predicates; `cpu_above/2` takes its threshold first so that the closure `cpu_above(1000)` leaves the thread identifier to be supplied last. This ordering, the fixed arguments first and the varying one last, is the convention every predicate meant to be passed as a closure follows, and it is the convention `call/N` imposes. The predicate `threads_where/2` is a `findall/3` with a hole in it, and the hole is what makes it reusable: Chapter 7 wrote one `thread_class/2` with six tests in a chain, and the same tests can now be passed to `threads_where/2` one at a time.

10.2 Mapping Lists

`maplist/2`, `maplist/3`, `maplist/4`, and `maplist/5` apply a closure to the corresponding elements of one to four lists. `maplist(G, L1)` calls `G` with each element of `L1` in turn and succeeds when every call succeeds; `maplist(G, L1, L2)` calls `G` with the elements of `L1` and `L2` at the same position, and so on. The lists must have the same length, and the predicate fails when they do not; the goal is called on the elements from left to right, and its bindings and choice points are kept, so a goal with several solutions gives `maplist` several.

```
?- maplist(succ, [0, 1], L).
L = [1, 2].

?- maplist(succ, L, [1, 2]).
L = [0, 1].

?- maplist(succ, [0, 1], [1]).
false.

?- maplist(thread_state, [100, 102, 108], Ss).
Ss = [waiting(user_request), running, suspended].

?- maplist(thread_time, [100, 102], Us, Ks).
Us = [120, 8450],
Ks = [40, 1230].

?- maplist(cpu_time, [100, 101, 102], Ms).
Ms = [160, 20, 9680].
```

The second query runs `succ/2` backward on every element; `maplist/3` has no direction of its own, and the mode of the call is the mode of the goal. The first `maplist/3` over the dump maps thread identifiers to states, and `maplist/4` with `thread_time/3` splits the times into two lists at once. The lists are related element by element, which is why `maplist/4` and `maplist/5` exist: a relation of three or four arguments over positions needs no zipping and unzipping.

Two uses of `maplist/2` are idioms. With a test as its goal it states that every element passes the test, and fails on the first that does not; with the closure `=(X)` it states that every element is `X`, which fills a list of known length with one value or checks that a list is uniform:

```
?- maplist(cpu_above(10), [100, 102, 107]).
true.

?- maplist(cpu_above(10), [100, 108]).
false.

?- length(L, 3), maplist(=(0), L).
L = [0, 0, 0].

?- maplist(=(X), [1, 1, 1]).
X = 1.
```

```
?- maplist(=(X), [1, 2]).
false.
```

In the third query `length/2` fixes the length before `maplist/2` fills the list; in the other order, `maplist(=(0), L)` with `L` unbound would enumerate the lists `[]`, `[0]`, `[0, 0]`, and so on without end, as `length/2` with both arguments unbound did in Chapter 5. The partial-list behavior is a consequence of `maplist` being defined by recursion on its lists, exactly as the list predicates of Chapter 5 are, and the same discipline applies: bind the lengths first.

The nondeterminism of the goal passes through. `maplist(cpu_time, Ts, [160, 20])` with `Ts` unbound finds the threads with those times, and the trailing `false.` is `thread_time/3` trying its remaining facts for the second element:

```
?- maplist(cpu_time, Ts, [160, 20]).
Ts = [100, 101] ;
false.

?- maplist(member, [X, Y], [[a, b], [c]]).
X = a,
Y = c ;
X = b,
Y = c.
```

The second query enumerates the ways of choosing one element from each list, which is the search of Chapter 4 over two `member/2` goals written as one `maplist/3`.

10.3 Filtering and Partitioning

`include/3` and `exclude/3` are the filter of Chapter 5 with the test made an argument. `include(Test, List, Included)` keeps the elements for which `call(Test, X)` succeeds and `exclude(Test, List, Excluded)` the elements for which it fails; each element is tested once, the first solution of the test is taken, and both predicates are deterministic. The `without_module/3` of Chapter 5 is `exclude/3` with the closure `==(ntdll)`, whose completion `==(ntdll, X)` is the test `ntdll == X`:

```
?- exclude(==(ntdll), [ntdll, kernel32, ntdll,
   appserver], Rest).
Rest = [kernel32, appserver].

?- include(==(ntdll), [ntdll, kernel32, ntdll,
   appserver], Kept).
Kept = [ntdll, ntdll].

?- findall(T, thread(T, _), _Ts),
   include(blocked, _Ts, Blocked).
Blocked = [101, 103, 104].

?- findall(T, thread(T, _), _Ts),
   exclude(cpu_above(100), _Ts, Quiet).
Quiet = [101, 103, 104, 105, 106, 108, 109].
```

The answers end in `.` where the recursion of Chapter 5 left a choice point after each: `include/3` and `exclude/3` are written with if-then-else, as Chapter 7 recommended for that filter. The test is a goal, and bindings it makes persist; a test that binds its argument changes the list, which is occasionally wanted and more often a surprise, and a test meant to inspect without binding is written with `==/2` and the type tests of Chapter 2 rather than `=/2`.

`partition/4` does both at once: `partition(Test, List, Included, Excluded)` splits a list into the elements that pass and the elements that do not, keeping the order of each. `partition/6` splits three ways. Its test takes two further arguments, the element and an order to be bound to `<`, `=`, or `>`, as `compare/3` binds its first argument, and the elements go to the lists `Less`, `Equal`, and `Greater` accordingly:

```
% cpu_order(Threshold, TID, Order): Order is the
% order of TID's CPU time relative to Threshold.
cpu_order(Threshold, TID, Order) :-
    cpu_time(TID, Ms),
    compare(Order, Ms, Threshold).

?- findall(T, thread(T, _), _Ts),
   partition(running, _Ts, Running, Others).
Running = [102, 107],
Others = [100, 101, 103, 104, 105, 106, 108, 109].

?- findall(T, thread(T, _), _Ts),
   partition(cpu_order(40), _Ts,
             Below, At, Above).
Below = [101, 104, 105, 106, 108],
At = [103],
Above = [100, 102, 107, 109].
```

Thread 103 consumed exactly 40 ms and lands in the middle list. The three-way split is the shape of a comparison against a threshold with the boundary kept separate, which a report wants when the threshold is a round number an analyst chose and the values at the boundary deserve a look of their own.

10.4 Folding

Mapping and filtering produce a list from a list. A computation that produces one value from a list, a sum, a maximum, a count, a structure built up element by element, is an accumulator recursion in the sense of Chapter 5, and `foldl/4` is that recursion with the step made an argument. `foldl(G, List, V0, V)` calls `G` once per element from left to right, each time with three arguments appended: the element, the accumulator so far, and the accumulator after the step. `V0` is the initial value, and `V` is the accumulator after the last element; for the empty list, `V` is `V0`.

```
% add(X, S0, S): S is S0 plus X.
add(X, S0, S) :-
    S is S0 + X.
```

```
% larger(X, M0, M): M is the larger of M0 and X.
larger(X, M0, M) :-
    M is max(M0, X).

% push(X, Xs, [X | Xs]): a step that prepends.
push(X, Xs, [X | Xs]).

?- foldl(add, [1, 2, 3], 0, S).
S = 6.

?- foldl(add, [], 0, S).
S = 0.

?- foldl(larger, [3, 9, 2], 0, M).
M = 9.

?- foldl(push, [a, b, c], [], R).
R = [c, b, a].
```

The argument order of the step is the order of `call/N`: the element first, because it is the varying argument `foldl/4` supplies first, then the accumulator before and after. `push/3` reverses a list, and is the `list_reverse/3` of Chapter 5 without its wrapper; `foldl/4` is the wrapper. The accumulator may be any term. A pair carries two accumulators at once, and a step that counts and sums in one pass over the list is written with `N0-S0` and `N-S` as its accumulators:

```
% count_sum(X, N0-S0, N-S): one more element,
% X more in the sum.
count_sum(X, N0-S0, N-S) :-
    N is N0 + 1,
    S is S0 + X.

?- foldl(count_sum, [10, 20, 30], 0-0, N-S).
N = 3,
S = 60.

?- findall(T, thread(T, _), _Ts),
   maplist(cpu_time, _Ts, _Ms),
   foldl(count_sum, _Ms, 0-0, N-S),
   Mean is S / N.
N = 10,
S = 14135,
Mean = 1413.5.
```

The second query is the mean CPU time of the threads of the dump in three library goals and one arithmetic goal; the total of 14135 ms is the one Chapter 6 computed with a walk over the thread identifiers and Chapter 9 with `aggregate_all/3`. The three formulations divide the work differently. Chapter 9's `aggregate_all(sum(Ms), cpu_time(_, Ms), Total)` runs a goal and sums its solutions without ever holding them in a list; `foldl/4` works on a list that exists. When the values come from a goal, `aggregate_all/3` is the shorter and does not build the list; when the values are already a list, because an earlier step produced or sorted them, `foldl/4` is the natural continuation, and Section 10.9 uses it on a sorted table.

`foldl/5` and `foldl/6` fold two and three lists at once, supplying the corresponding elements of each before the accumulators:

```
% add_product(X, Y, S0, S): S is S0 plus X * Y.
add_product(X, Y, S0, S) :-
    S is S0 + X * Y.

?- foldl(add_product, [1, 2, 3], [4, 5, 6], 0, S).
S = 32.
```

10.5 Lambda Expressions

Each of the steps above was a two-line predicate whose only purpose was to be passed once. `library(yall)` lets such a goal be written where it is used, as a **lambda expression** of the form `[X1, ..., Xn]>>Body`. When `call/N` completes it with arguments, the arguments are unified with the parameters `X1` to `Xn` in order, any further arguments are appended to `Body`, and the body is called:

```
?- call([X, Y]>>(Y is X * X), 4, R).
R = 16.

?- foldl([X, S0, S]>>(S is S0 + X),
   [1, 2, 3], 0, S).
S = 6.

?- maplist([T, T-Ms]>>cpu_time(T, Ms),
   [100, 102], Pairs).
Pairs = [100-160, 102-9680].

?- include([T]>>(cpu_time(T, Ms), Ms > 1000),
   [100, 102, 107], Busy).
Busy = [102, 107].
```

The parameters are terms, not only variables: the `maplist/3` call above has `T-Ms` as its second parameter, so that each output element is built by unification with the parameter and the body only fills in `Ms`. The notation `>>` is an ordinary SWI-Prolog operator and the lambda expression an ordinary term; `library(yall)` defines `>>/2` to `>>/9` as the predicates that `call/N` reaches when it completes one. The library is autoloaded on first use in a toplevel that has not loaded it; this chapter loads it with a directive for a reason that follows.

The body of a lambda expression is called on a **copy** of the expression, made at each call. Variables of the body that are parameters are unified with the arguments of that call; variables that are not parameters are renamed in the copy, so that a binding made to them inside the body does not reach the surrounding clause or query, and a variable of the surrounding context that is unbound when the call is made is a different variable inside the body. A variable that was already bound when the call is made is copied with its value, so a threshold held in a variable of the query does reach the body. The `Ms` of the `include/3` call above is of the renamed kind: it is bound inside the body by `cpu_time/2` and compared there, and is a fresh variable in each call, which is what a per-element temporary should be.

```
?- maplist([X]>>(Y = X), [1]), var(Y).
true.

?- maplist({Y}/[X]>>(Y = X), [1]).
Y = 1.
```

The first query binds `Y` inside the copy and leaves the `Y` of the query unbound. The second declares `Y` as a **free variable** of the lambda expression with the prefix `{Y}/`: the variables listed in the braces are shared between the body and its context, excluded from the copy, and a binding made to them inside is a binding outside. Several are written `{Y, Z}/`. The rule that follows is simple to state: every variable of the surrounding clause that the body uses is declared free, whether the body reads it or binds it.

The rule matters more than the copying semantics suggest, because of a second mechanism. When a lambda expression appears in a clause that SWI-Prolog compiles from a file, and the meta-predicate it is passed to is already loaded, so that the compiler knows which argument is a goal and how many arguments it will receive, the expression is compiled at load time into an auxiliary predicate whose arguments are the declared free variables and the parameters. Variables of the body that were neither become local to that predicate. The following two predicates differ only in the braces:

```
% scale_all(F, Xs, Ys): Ys is Xs with each element
% multiplied by F.
scale_all(F, Xs, Ys) :-
    maplist({F}/[X, Y]>>(Y is X * F), Xs, Ys).

% scale_all_wrong(F, Xs, Ys): as above, with F not
% declared free.
scale_all_wrong(F, Xs, Ys) :-
    maplist([X, Y]>>(Y is X * F), Xs, Ys).

?- scale_all(3, [1, 2], L).
L = [3, 6].

?- catch(scale_all_wrong(3, [1, 2], L),
   error(E, _), true).
E = instantiation_error.

?- F = 3,
   maplist([X, Y]>>(Y is X * F), [1, 2], L).
F = 3,
L = [3, 6].
```

The second predicate raises an instantiation error: `library(apply)` was loaded by the directive at the start of this chapter, `maplist/3` was known when the clause was compiled, the lambda was compiled into an auxiliary predicate, and the `F` in its body is a variable that nothing binds. The same lambda typed at the toplevel in the third query works, because the toplevel calls `>>/4` at run time and the copy is made after `F` has been bound to 3. Had the chapter relied on autoloading, `scale_all_wrong/3` would have worked as well, until some other file loaded `library(apply)` first. A lambda whose correctness depends on the order in which libraries were loaded is a fault waiting

for a change elsewhere, and the braces remove the dependence: with `{F}/` the two paths agree. This is the one caveat of `library(yall)`, and the two directives at the start of the chapter are there so that the compiled behavior, the one a program sees, is the one the transcripts show.

The free-variable prefix is also usable alone. `{X}/Goal` calls a copy of `Goal` in which only `X` is shared, which is the complement of the `^` of Chapter 9: where `Y^p(X, Y)` names the variables to hide, `{X}/p(X, _)` names the variables to keep.

```
?- setof(T, {T}/thread_state(T, waiting(_)), Ts).
Ts = [100, 101, 103, 104, 105, 106, 109].
```

A different lambda notation, `\X^Body`, belongs to `library(lambda)`, a separate library that SWI-Prolog does not ship, and is not recognized by `library(yall)` in this edition (9.0.4); this book uses `>>` only.

Lambda expressions are for goals too small to deserve a name. A step that is used twice, that needs a comment, or that a reader would want to test alone is better as a named predicate, and the sections above wrote `add/3` and `cpu_order/3` as such for that reason. The two forms cost the same at run time once compiled, and the choice is one of readability.

10.6 Association Lists

A list of `Key-Value` pairs, as Chapter 9 built them, is looked up by `member/2` in time proportional to its length. `library(assoc)` provides an **association list**, a mapping from keys to values stored as an AVL tree, a balanced binary tree in which lookup, insertion, and deletion take time proportional to the logarithm of the number of keys. Keys are compared in the standard order of terms and should be ground; values are arbitrary terms. The predicates:

Predicate	Relation
`empty_assoc(-A)`	`A` has no keys
`put_assoc(+K, +A0, +V, -A)`	`A` is `A0` with `K` mapped to `V`
`get_assoc(+K, +A, -V)`	`K` is mapped to `V` in `A`
`del_assoc(+K, +A0, ?V, -A)`	`A` is `A0` without `K`
`list_to_assoc(+Pairs, -A)`	`A` holds the pairs
`assoc_to_list(+A, -Pairs)`	pairs of `A`, by key
`assoc_to_keys(+A, -Ks)`	keys of `A`, in order
`assoc_to_values(+A, -Vs)`	values of `A`, by key
`max_assoc(+A, -K, -V)`	the largest key
`min_assoc(+A, -K, -V)`	the smallest key

```
?- list_to_assoc([101-blocked, 102-running], _A),
   get_assoc(102, _A, Class).
Class = running.
```

```
?- list_to_assoc([101-blocked, 102-running], _A),
   get_assoc(105, _A, Class).
false.

?- list_to_assoc([b-2, a-1, c-3], _A),
   assoc_to_list(_A, L),
   assoc_to_keys(_A, Ks),
   assoc_to_values(_A, Vs).
L = [a-1, b-2, c-3],
Ks = [a, b, c],
Vs = [1, 2, 3].

?- catch(list_to_assoc([a-1, a-2], _A),
   error(E, _), true).
E = domain_error(unique_key_pairs, [a-1, a-2]).
```

`get_assoc/3` fails on an absent key, which makes it a test as well as a lookup; `list_to_assoc/2` requires the keys to be distinct and raises a domain error otherwise, where `put_assoc/4` on an existing key replaces the value. `assoc_to_list/2` yields the pairs in the standard order of their keys whatever the order of insertion, and `assoc_to_keys/2` and `assoc_to_values/2` likewise. The tree is an ordinary term, with functor `t/5` for a node and the atom `t` for the empty tree, and the toplevel prints it as such; the queries above bind it to a variable beginning with an underscore so that the structure is not shown. An association list is **persistent**: `put_assoc/4` does not change its input but builds a new tree sharing most of the old one, so that the old tree remains valid and a program may keep both.

```
?- empty_assoc(_A0),
   put_assoc(k, _A0, 1, _A1),
   put_assoc(k, _A1, 2, _A2),
   get_assoc(k, _A1, V1),
   get_assoc(k, _A2, V2).
V1 = 1,
V2 = 2.
```

The combination of `foldl/4` with `put_assoc/4` as the step builds a mapping in one pass over a list. A histogram, which Chapter 9 built by sorting and counting runs, is the mapping from each value to the number of times it has been seen, and the step looks the value up, adds one or starts at one, and stores it back:

```
% count_into(Key, A0, A): A is A0 with the count
% of Key increased by one, or set to 1 if absent.
count_into(Key, A0, A) :-
    (   get_assoc(Key, A0, N0)
    ->  N is N0 + 1
    ;   N = 1
    ),
    put_assoc(Key, A0, N, A).
```

```
% module_counts(Counts): Counts is the list of
% Module-N pairs, N the number of frames in Module
% over all stack traces of the dump, by module.
module_counts(Counts) :-
    findall(M, frame(_, _, M, _, _), Ms),
    empty_assoc(A0),
    foldl(count_into, Ms, A0, A),
    assoc_to_list(A, Counts).

?- module_counts(_Counts),
   forall(member(M-N, _Counts),
          format("~a~t~14|~t~d~4+~n", [M, N])).
appserver       13
cachelib         1
dbclient         5
kernel32        10
kernelbase       2
ntdll           28
reportengine    13
ucrtbase         6
true.
```

The counts are those of Chapter 9, by module rather than by frequency; `predsort/3` with the `by_count_desc/3` of that chapter would reorder them. The two constructions do the same work in the same order of time, and differ in what they leave behind: the sorted list is gone once counted, where the association list remains and answers `get_assoc(ntdll, A, N)` for any module in logarithmic time. An analysis that counts once and looks up many times keeps the tree.

`library(rbtrees)` provides the same mapping as red-black trees, with the predicates `rb_new/1`, `rb_insert/4`, `rb_lookup/3`, and `rb_update/4` in place of the ones above; the two libraries differ in constants, not in complexity, and the association list is the one most SWI-Prolog code uses. Both are the right structure when a key is a term; when keys are small consecutive integers, `arg/3` on a compound term serves, and when the data is a set of facts the clause index of Chapter 13 is faster than either.

10.7 Dicts

SWI-Prolog has a second kind of structured term beside the compound term, the **dict**, which is not part of ISO Prolog and does not exist in other Prolog systems. A dict is written `Tag{Key1: Value1, Key2: Value2}`: the **tag** is an atom or a variable, the keys are atoms or small integers and must be distinct, and the values are arbitrary terms. The order in which the pairs are written does not matter; two dicts are equal when their tags and their sets of pairs are, and the toplevel prints the pairs in the standard order of the keys.

```
?- D = thread{tid: 102, cpu: 9680}.
D = thread{cpu:9680, tid:102}.
```

```
?- thread{cpu: 9680, tid: 102} ==
   thread{tid: 102, cpu: 9680}.
true.

?- thread{tid: 102} = thread{tid: 102, cpu: 9680}.
false.

?- thread{tid: T} :< thread{tid: 102, cpu: 9680}.
T = 102.
```

Unification of two dicts requires the same key set, so the third query fails. The **select** operator `:</2` unifies the pairs of its left operand with the corresponding pairs of its right and ignores the keys the right has in addition, which is the pattern-matching mode: a dict with the keys of interest as a template, as a `frame/5` term with unbound arguments was in Chapter 3. The access predicates are `get_dict/3`, which fails on an absent key, `put_dict/4`, which yields a new dict with one pair added or replaced, `del_dict/4`, and `dict_pairs/3`, which relates a dict to its tag and its list of pairs in either direction:

```
?- get_dict(cpu, thread{tid: 102, cpu: 9680}, V).
V = 9680.

?- get_dict(state, thread{tid: 102, cpu: 9680},
   V).
false.

?- put_dict(cpu, thread{tid: 102}, 9680, D).
D = thread{cpu:9680, tid:102}.

?- dict_pairs(D, thread, [tid-102, cpu-9680]).
D = thread{cpu:9680, tid:102}.

?- dict_pairs(thread{tid: 102, cpu: 9680},
   Tag, Ps).
Tag = thread,
Ps = [cpu-9680, tid-102].
```

Dicts also support a **functional notation** in the bodies of clauses and in toplevel queries: `D.cpu` stands for the value of key `cpu` in `D`, `D.get(cpu)` for the same, `D.get(cpu, Default)` for the value or the default, and `D.put(state, running)` for the dict with the pair added. The notation is not a predicate. When SWI-Prolog compiles a clause or reads a query, it rewrites each goal containing a `.` expression into a call of `./3` placed immediately before the goal, with a fresh variable in the expression's place; `X = D.cpu` becomes `.(D, cpu, V), X = V`. A dict record of a thread and two predicates reading it with the notation:

```
% thread_record(TID, Record): Record is a dict
% with the identifier, state, and CPU time of TID.
thread_record(TID, Record) :-
    Record = thread{tid: TID, state: S, cpu: Ms},
    thread_state(TID, S),
    cpu_time(TID, Ms).
```

```
% busy(Record): the thread consumed more than a
% second of CPU time.
busy(Record) :-
    Record.cpu > 1000.

% record_state(Record, State): State is the state
% field of Record.
record_state(Record, State) :-
    State = Record.state.


?- thread_record(102, R).
R = thread{cpu:9680, state:running, tid:102}.

?- thread_record(102, R), X = R.cpu.
R = thread{cpu:9680, state:running, tid:102},
X = 9680.

?- thread_record(101, _R), record_state(_R, S).
S = waiting(user_request).

?- findall(T, thread(T, _), _Ts),
   maplist(thread_record, _Ts, _Rs),
   include(busy, _Rs, _Busy),
   maplist([R, T]>>get_dict(tid, R, T),
           _Busy, Ts).
Ts = [102, 107].

?- thread_record(102, _R),
   catch(X = _R.owner,
         error(existence_error(key, K, _), _),
         true).
K = owner.

?- thread_record(102, _R),
   X = _R.get(owner, none).
X = none.
```

The last two queries show the difference between `.key` and `get_dict/3`: an absent key is an existence error in the functional notation and a failure in the predicate, and `.get(Key, Default)` is the lookup with a default of Chapter 7. The rewriting is also the caution. It is done to the goal as written, before anything runs, so a `.` expression works only where the dict is bound by the time the goal before it runs. Inside a lambda body at the toplevel it does not, because the rewriting lifts the `./3` call out of the lambda and places it before the `maplist/3` or `include/3` goal, where the parameter is still unbound; the `maplist/3` in the fourth query uses `get_dict/3` for that reason. A `.` expression inside a term read as data with `read_term/2` is not rewritten either, since the rewriting belongs to the compilation of clauses and queries and not to the reading of terms; a dict itself, written as `Tag{...}`, is read back by `read_term/2` as a dict.

Dicts are the right representation for a record with named fields, some of which may be absent, that is built and consumed inside one program. The shared schema of this book is compound terms, `frame/5` and `msg/6`, because facts in files are read by every Prolog system and because positional arguments are what clause indexing works on (Chapter 13). A dict is what the result of an analysis

looks like when it is handed to a report or converted to JSON (Chapter 13), and `thread_record/2` is the shape of that hand-over.

10.8 Graphs

`library(ugraphs)` represents a directed graph as a list of pairs `Vertex-Neighbors`, one per vertex in the standard order of terms, where `Neighbors` is the sorted list of the vertices reachable by one edge. Such a term is an **unweighted graph**, or ugraph, and the predicates of the library build it from vertices and edges and compute the relations an analyst asks of a graph without the recursion Chapter 4 showed does not terminate on a cycle:

Predicate	Relation
`vertices_edges_to_ugraph(+Vs, +Es, -G)`	build `G`
`vertices(+G, -Vs)`	the vertices
`edges(+G, -Es)`	the edges `From-To`
`neighbours(+V, +G, -Ns)`	one step from `V`
`add_edges(+G0, +Es, -G)`	`G0` plus edges
`del_edges(+G0, +Es, -G)`	`G0` minus edges
`transpose_ugraph(+G0, -G)`	all edges reversed
`reachable(+V, +G, -Vs)`	`Vs` reachable from `V`
`transitive_closure(+G0, -G)`	closure of `G0`
`top_sort(+G, -Vs)`	topological order

```
?- vertices_edges_to_ugraph([d], [a-b, b-c], G).
G = [a-[b], b-[c], c-[], d-[]].

?- vertices_edges_to_ugraph([], [a-b, b-c], G),
   vertices(G, Vs), edges(G, Es),
   neighbours(b, G, Ns).
G = [a-[b], b-[c], c-[]],
Vs = [a, b, c],
Es = [a-b, b-c],
Ns = [c].

?- vertices_edges_to_ugraph([], [a-b, b-c], G),
   reachable(a, G, R1), reachable(c, G, R2).
G = [a-[b], b-[c], c-[]],
R1 = [a, b, c],
R2 = [c].

?- vertices_edges_to_ugraph([], [a-b, b-c], G),
   transitive_closure(G, C).
G = [a-[b], b-[c], c-[]],
C = [a-[b, c], b-[c], c-[]].

?- vertices_edges_to_ugraph([], [a-b, b-c], G),
   top_sort(G, Order).
G = [a-[b], b-[c], c-[]],
Order = [a, b, c].
```

`vertices_edges_to_ugraph/3` takes the vertices that may have no edges, here `d`, and the edges as `From-To` pairs; a vertex that occurs in an edge need not be listed. `reachable/3` gives the vertices reachable from a vertex by zero or more edges, so the start vertex is always included. `transitive_closure/2` is the graph in which every vertex has an edge to everything reachable from it by one or more edges. `top_sort/2` orders the vertices so that every edge goes forward in the order, which exists exactly when the graph has no cycle, and the predicate fails on a graph with one:

```
?- vertices_edges_to_ugraph([], [a-b, b-c, c-a],
   G), top_sort(G, Order).
false.

?- vertices_edges_to_ugraph([], [a-b, b-c, c-a],
   G), transitive_closure(G, C).
G = [a-[b], b-[c], c-[a]],
C = [a-[a, b, c], b-[a, b, c], c-[a, b, c]].

?- vertices_edges_to_ugraph([], [a-b, b-c], G),
   transpose_ugraph(G, T), reachable(c, T, R).
G = [a-[b], b-[c], c-[]],
T = [a-[], b-[a], c-[b]],
R = [a, b, c].
```

In the closure of a cyclic graph a vertex on the cycle has an edge to itself, which is the test for a cycle Section 10.9 uses. `transpose_ugraph/2` reverses every edge, and reachability in the transposed graph is the set of vertices from which a vertex can be reached, the question asked backward. The library's algorithms are all implemented over sorted lists with the ordered-set predicates of `library(ordsets)`, and they terminate on any finite graph, cyclic or not, because they work on the set of vertices rather than on paths; what they do not give is a path, and a program that needs the sequence of vertices from one to another writes the search of Chapter 5 with its visited list, or the tabled one of Chapter 11.

10.9 Diagnostic Use

The wait-for relation of Chapter 4 is a graph. Its vertices are threads and its edges go from a waiting thread to the owner of the critical section it waits for, which is the `blocked_by/2` of that chapter, and the ugraph is built from the edges `findall/3` collects:

```
% blocked_by(Waiter, Owner): Waiter waits for a
% critical section that Owner owns.
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, critical_section, Addr),
    critical_section(Addr, Owner, _, _),
    Owner \== 0.

% wait_for_graph(Graph): Graph is the ugraph whose
% edges are the blocked_by/2 pairs of the dump.
wait_for_graph(Graph) :-
    findall(W-O, blocked_by(W, O), Edges),
    vertices_edges_to_ugraph([], Edges, Graph).
```

```
?- wait_for_graph(G).
G = [101-[103], 103-[104], 104-[103]].

?- wait_for_graph(_G),
   vertices(_G, Vs), edges(_G, Es).
Vs = [101, 103, 104],
Es = [101-103, 103-104, 104-103].
```

Three vertices and three edges: the whole of what `!locks` and the three blocked stack traces say about synchronization in this dump, in one term. The **Wait Chain (Critical Sections)** (page 384) of Chapter 4 followed the edges from thread 101 one goal at a time; the set of threads a thread waits on, directly or through other threads, is reachability from it, and the set of threads a thread holds up is reachability in the transposed graph:

```
% reached_from(V, G, Others): Others are the
% vertices reachable from V in G, other than V;
% [] when V is not a vertex of G.
reached_from(V, G, Others) :-
    (   reachable(V, G, Reached)
    ->  subtract(Reached, [V], Others)
    ;   Others = []
    ).

% waits_on(TID, Others): Others are the threads
% TID waits for, directly or through a chain.
waits_on(TID, Others) :-
    wait_for_graph(G),
    reached_from(TID, G, Others).

% holds_up(TID, Others): Others are the threads
% that wait for TID, directly or through a chain.
holds_up(TID, Others) :-
    wait_for_graph(G),
    transpose_ugraph(G, T),
    reached_from(TID, T, Others).

?- waits_on(101, Ts).
Ts = [103, 104].

?- waits_on(103, Ts).
Ts = [104].

?- waits_on(102, Ts).
Ts = [].

?- holds_up(103, Ts).
Ts = [101, 104].

?- holds_up(104, Ts).
Ts = [101, 103].
```

Thread 101 waits, through 103, on 104 as well; thread 102 waits on nothing, since it is not a vertex of the graph at all. `reachable/3` fails for a vertex that is not in the graph, and the if-then-else of `reached_from/3` turns that failure into the empty list, so that `waits_on/2` is defined for every thread. The transposed question is the one an analyst asks of a lock owner: thread 103 holds up 101 and

104, and so does 104. The `subtract/3` of Chapter 5 removes the start vertex that `reachable/3` includes, with one consequence to be stated: `waits_on(103, Ts)` does not report that 103 waits, through 104, on itself. The chain's return to its start is the cycle, and the cycle is a separate question.

A **Deadlock (Critical Sections)** (page 388) is a cycle in the wait-for graph: a set of threads each of which waits, directly or through others of the set, for itself. In the transitive closure a vertex on a cycle reaches itself, and the deadlocked threads are exactly those. The closure is printed one vertex per line by the failure-driven loop of Chapter 4, since the term is wider than the page:

```
% deadlocked(TID): TID waits, through a chain of
% owners, for a critical section it owns itself.
deadlocked(TID) :-
    wait_for_graph(G),
    transitive_closure(G, Closure),
    member(TID-Reached, Closure),
    memberchk(TID, Reached).

% deadlock_present: the wait-for graph has a
% cycle.
deadlock_present :-
    wait_for_graph(G),
    \+ top_sort(G, _).

?- wait_for_graph(_G), transitive_closure(_G, _C),
   member(T-Reached, _C),
   format("~d reaches ~w~n", [T, Reached]), fail.
101 reaches [103,104]
103 reaches [103,104]
104 reaches [103,104]
false.

?- deadlocked(T).
T = 103 ;
T = 104.

?- deadlocked(101).
false.

?- deadlock_present.
true.
```

Threads 103 and 104 reach themselves and thread 101 does not: it is blocked by the deadlock without being part of it, which is the distinction between a **Wait Chain (Critical Sections)** (page 384) that ends in a deadlock and the **Deadlock (Critical Sections)** (page 388) itself, and the one Chapter 5's visited list could not make. `deadlock_present/0` is the same fact asked as a yes-or-no question: a topological order exists exactly when there is no cycle, so its failure is the presence of one. The recursive `blocked_through/2` of Chapter 4 did not terminate on this graph; the two predicates here cannot fail to, because `library(ugraphs)` computes over sets of vertices. What they do not give is the cycle as a sequence, 103 then 104 then 103, and an analyst reporting a deadlock wants that sequence with the critical section at each step; Chapter 11 obtains it with tabling, and Chapter 20 writes the recognizers in their final form with both paths and evidence.

The CPU table is the second use. Chapter 9 printed it with `forall/2` over `order_by/2` and recomputed the total for every row, and noted that a report over thousands of threads would compute it once. `foldl/4` is the way to pass a value computed once down a list of rows, and to carry a running sum through them:

```
% cpu_rows(Rows): Rows is the list of Ms-TID pairs
% of the dump, largest CPU time first.
cpu_rows(Rows) :-
    findall(Ms-TID, cpu_time(TID, Ms), Pairs),
    sort(1, @>=, Pairs, Rows).

% add_ms(Row, S0, S): S is S0 plus the CPU time
% of Row.
add_ms(Ms-_, S0, S) :-
    S is S0 + Ms.

% print_cpu_row(Total, Row, Cum0, Cum): print one
% row with its share of Total and the cumulative
% share Cum of the rows so far.
print_cpu_row(Total, Ms-TID, Cum0, Cum) :-
    Cum is Cum0 + Ms,
    Share is 100 * Ms / Total,
    CumShare is 100 * Cum / Total,
    format(
        "~d~t~5|~t~D~7+ ms~t~1f~9+%~t~1f~7+%~n",
        [TID, Ms, Share, CumShare]).

% print_cpu_shares: the CPU table with a
% cumulative share column, largest consumer first.
print_cpu_shares :-
    cpu_rows(Rows),
    foldl(add_ms, Rows, 0, Total),
    foldl(print_cpu_row(Total), Rows, 0, _).

?- print_cpu_shares.
102    9,680 ms  68.5%  68.5%
107    4,100 ms  29.0%  97.5%
100      160 ms   1.1%  98.6%
109       60 ms   0.4%  99.0%
103       40 ms   0.3%  99.3%
104       35 ms   0.2%  99.6%
105       25 ms   0.2%  99.8%
101       20 ms   0.1%  99.9%
106       15 ms   0.1% 100.0%
108        0 ms   0.0% 100.0%
true.
```

Two folds over the same sorted rows. The first totals the times, and the second prints, with the total fixed in the closure `print_cpu_row(Total)` and the running sum of the rows printed so far as the accumulator; the final accumulator is the total again and is discarded. The cumulative column is what the fold adds to Chapter 9's table, and it states the **Spiking Thread** (page 335) finding in the form an analyst reports it: two threads account for 97.5 percent of the CPU time of the process, and the remaining eight for 2.5. The arithmetic is unchanged from Chapter 6; `~1f` rounds for display and the comparison in a recognizer uses the unrounded shares. The same two-fold shape serves any report

with a column that depends on the whole, a share, a rank, a running total, and with `maplist/3` and `include/3` in place of `findall/3` it serves a list of records already in hand, as the `include(busy, ...)` of Section 10.7 did.

The module histogram of Section 10.6 and the thread records of Section 10.7 are the remaining constructs of this chapter applied to the dump, and the three libraries divide the representation of its facts between them. The facts themselves stay as clauses, where indexing finds them; an association list holds a mapping computed from them, such as the counts; a dict holds a record assembled from several relations about one thread; and a ugraph holds a relation about to be asked for its closure. Chapter 15 returns to the first of these choices when it designs the schema, and Chapter 30 measures the others.

Chapter 11. The Dynamic Database, Global State, and Tabling

The knowledge base of every chapter so far was fixed when it was consulted. The facts of `dump1.pl` were read once and never changed, every predicate was defined by the clauses in a file, and a query could only derive from what was there. Three things an analysis does need the program to change while it runs. The analyst learns facts during the analysis, from a vendor's web page, from a colleague, from a second dump, and wants them in the knowledge base beside the facts of the artifact; a program needs a counter or a setting that outlives the goal that set it; and a recursive definition that is correct under the declarative reading, such as the transitive closure of the wait-for relation that Chapter 4 wrote and could not run on cyclic data, needs the system to remember which calls it has already made. This chapter covers the three mechanisms: the **dynamic database**, the set of predicates whose clauses may be added and removed at run time with `assertz/1` and `retract/1`; **global variables**, named locations that hold a term across goals; and **tabling**, under which SWI-Prolog records the answers of a predicate as it computes them and reuses them, so that left recursion and cyclic data terminate and expensive predicates are computed once.

The diagnostic uses are the **Wait Chain (Critical Sections)** (page 384) and **Deadlock (Critical Sections)** (page 388) of `dump1.pl`, written as the transitive closure of Chapter 4 with a single declaration added and now terminating, with the chain as a sequence of threads obtained by answer subsumption; and the analyst's annotations, facts about threads, modules, and critical sections that are not in the dump, kept in a dynamic predicate as the analysis proceeds and queried alongside the dump's own facts.

11.1 Static and Dynamic Predicates

A predicate defined by clauses in a consulted file is **static**: SWI-Prolog compiles its clauses and refuses to change them from a running program, and an attempt to do so raises a permission error. A **dynamic predicate** is one declared with the directive `dynamic/1`, and its clauses may be added and removed at run time. The declaration also settles a question Chapter 1 left open: a call to a predicate that has no clauses is an existence error, because an unknown procedure is almost always a typing mistake, but a dynamic predicate with no clauses is an ordinary predicate that happens to be empty, and a call to it fails.

```
:- [dump1].

% marked(TID): the analyst has marked TID for
% attention.
:- dynamic marked/1.

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- marked(T).
false.
```

```
?- catch(assertz(thread(110, 4120)),
         error(permission_error(Op, Kind, P), _),
         true).
Op = modify,
Kind = static_procedure,
P = thread/2.
```

The first query fails rather than raising an error, and the second shows what protects the facts of the dump: `thread/2` was consulted from a file, is static, and cannot be added to. The error names the operation, the kind of object, and the predicate indicator, in the shape Chapter 7 gave for ISO error terms. The distinction is deliberate and this book keeps it. The facts of an artifact are static, read once from a file the parser of Chapter 12 produces, so that nothing in an analysis can alter the evidence; what the analyst adds goes into dynamic predicates declared for the purpose.

11.2 Adding and Removing Clauses

`assertz/1` adds a clause at the end of a predicate's clause list and `asserta/1` at the beginning. The argument is the clause as a term: a fact, or a rule written `(Head :- Body)` with the extra parentheses that the operator priority of `:-` requires inside an argument. A predicate that does not exist is created as a dynamic predicate by the first assertion, so the declaration of Section 11.1 is strictly needed only to make a call before the first assertion fail instead of raising an error; it is good practice to write it anyway, since it documents that the predicate is meant to change.

```
?- assertz(marked(102)), assertz(marked(107)),
   asserta(marked(109)).
true.

?- marked(T).
T = 109 ;
T = 102 ;
T = 107.

?- assertz((hot(T) :- marked(T),
                      thread_state(T, running))).
true.

?- hot(T).
T = 102 ;
T = 107.
```

The order of the answers is the order of the clauses: 109 was put first by `asserta/1`, and the other two follow in the order they were added. The rule `hot/1` was created by its assertion and works like any rule; it is asked in a separate query because SWI-Prolog's toplevel checks a query for unknown procedures before running it, and `hot/1` did not exist when the query that created it was read. The clause asserted is a **copy** of the term: variables of the clause are renamed, as `findall/3` renames its template, and a binding made later to the `T` of the query does not reach the stored clause.

`retract/1` removes the first clause that unifies with its argument and succeeds; on backtracking it removes the next, so that `retract(marked(T))` with `T` unbound removes the clauses one by one,

binding `T` to each, and `retract/1` with a fully instantiated argument is semideterministic. It fails when no clause unifies. A rule is retracted by `retract((Head :- Body))`, with `Body` either a term to be unified with the body or a variable that is bound to it. `retractall/1` removes every clause whose head unifies with its argument, always succeeds, and declares the predicate dynamic if it did not exist, which makes `retractall(p(_))` the idiom for emptying a predicate or ensuring it exists.

```
?- retract(marked(109)).
true.

?- retract(marked(999)).
false.

?- retract((hot(T) :- Body)).
Body = (marked(T), thread_state(T, running)).

?- retract(marked(T)).
T = 102 ;
T = 107.

?- marked(T).
false.

?- assertz(marked(102)), retractall(marked(_)),
   marked(T).
false.
```

`abolish/1` removes a predicate altogether, its clauses and its declaration with them, so that a later call raises an existence error rather than failing. It is the wrong predicate for emptying a dynamic predicate, which is `retractall/1`; its use is to remove a predicate that should not exist at all, which an analysis program rarely needs. The one caution about the whole family is older than SWI-Prolog and applies in full: assertion is for knowledge that arrives, not for computation. A program that asserts an intermediate result in one goal and reads it back in another has given itself a hidden argument that no clause head shows, that backtracking does not undo, and that a second call of the same program finds already set. The results of a computation travel in arguments; what goes into the dynamic database is what the program has been told. Section 11.6 shows that the one legitimate use of assertion for computation, remembering answers already found, is done for the program by tabling.

11.3 The Logical Update View

A goal that runs against a dynamic predicate while clauses are being added to or removed from it raises the question of which clauses it sees. ISO Prolog and SWI-Prolog answer with the **logical update view**: a call sees the clauses that existed when it was made, and clauses added or removed while it runs do not affect it. The goal runs, in effect, against a snapshot; the next call sees the changed predicate.

```
% item(N): a small dynamic relation for the
% examples of this section.
:- dynamic item/1.
item(1).
item(2).

?- item(X), Y is X + 10, assertz(item(Y)), fail.
false.

?- item(X).
X = 1 ;
X = 2 ;
X = 11 ;
X = 12.

?- item(X), retract(item(X)), fail.
false.

?- item(X).
false.
```

The first query is a failure-driven loop that adds a clause for each clause it visits. Without the logical update view it would visit the clauses it added, add clauses for those, and never terminate; under it, `item(X)` was called when two clauses existed, visits exactly those two, and the loop ends. The third query removes every clause while iterating over the predicate, which is likewise safe. SWI-Prolog implements the view by stamping each clause with the generation in which it was added and removed, and a call sees the clauses alive at its own generation; the cost of this is small and the alternative, under which the result of such a loop depends on where in the list new clauses land, is not worth having. A clause retracted while a call is iterating over the predicate remains visible to that call, which still visits it, and is invisible to every later call; that is what makes the third query well defined.

11.4 Inspecting the Program

The clauses of a predicate are terms, and `clause/2` retrieves them: `clause(Head, Body)` unifies `Head` with the head of a clause and `Body` with its body, and enumerates the clauses on backtracking. ISO restricts it to dynamic predicates; SWI-Prolog allows it on static predicates unless the flag `protect_static_code` is set, so the clauses of the dump are available to it.

```
?- clause(frame(109, 0, M, F, O), Body).
M = dbclient,
F = 'Buffer::Copy',
O = 26,
Body = true.

?- clause(thread_state(T, running), true).
T = 102 ;
T = 107.
```

A fact is a clause whose body is `true`, as Chapter 4 said, and `clause/2` shows that SWI-Prolog stores it so. The second query reads the clauses of a static predicate with a body pattern, which is the same as calling the predicate for facts and different for rules, where `clause/2` yields the body unrun. Chapter 29 builds a meta-interpreter on this predicate, a program that runs other programs by retrieving their clauses and resolving them itself.

`predicate_property/2` relates a goal to properties of its predicate: `dynamic`, `static`, `defined`, `built_in`, `tabled`, `number_of_clauses(N)`, `imported_from(Module)` for library predicates, and others. `current_predicate/1` succeeds when a predicate with the given indicator exists and, with the arity unbound, enumerates the arities a name is defined with.

```
?- predicate_property(marked(_), dynamic).
true.

?- predicate_property(frame(_, _, _, _, _),
   dynamic).
false.

?- predicate_property(frame(_, _, _, _, _),
   number_of_clauses(N)).
N = 78.

?- predicate_property(marked(_),
   number_of_clauses(N)).
N = 0.

?- current_predicate(frame/A).
A = 5 ;
false.

?- current_predicate(marked/1).
true.

?- current_predicate(hot/1).
true.

?- predicate_property(hot(_),
   number_of_clauses(N)).
N = 0.
```

Both `marked/1`, emptied by the `retractall/1` of Section 11.2, and `hot/1`, whose only clause was retracted, still exist as dynamic predicates with no clauses: a predicate created by assertion is as dynamic as one declared, and removing its clauses does not remove it. The predicates of this section are how a program asks about itself, which Chapter 13 uses to check that a module exports what it should and Chapter 17 to list the recognizers a knowledge base contains. `listing/1` of Chapter 1 prints the clauses of a dynamic predicate with its declaration, and so shows at a glance what the analyst has added:

```
?- listing(item/1).
:- dynamic item/1.
```

```
true.

?- assertz(item(5)), listing(item/1).
:- dynamic item/1.

item(5).

true.
```

11.5 Global Variables

A **global variable** in SWI-Prolog is a named location, its name an atom, that holds a term and is read and written by goals without being passed as an argument. There are two families. `b_setval/2` and `b_getval/2` are **backtrackable**: a value set is undone when the program backtracks past the goal that set it, as a binding is. `nb_setval/2` and `nb_getval/2` are **non-backtrackable**: the value survives backtracking, and the term stored is a copy, so that later bindings of its variables do not reach it. Reading a variable that was never set is an existence error.

```
?- b_setval(v, 1),
   ( b_setval(v, 2), fail ; true ),
   b_getval(v, X).
X = 1.

?- nb_setval(w, 1),
   ( nb_setval(w, 2), fail ; true ),
   nb_getval(w, X).
X = 2.

?- nb_getval(w, X).
X = 2.

?- catch(nb_getval(unset, X), error(E, _), true).
E = existence_error(variable, unset).

?- nb_setval(t, f(X)), X = 1, nb_getval(t, T).
X = 1,
T = f(_).

?- b_setval(t, f(X)), X = 1, b_getval(t, T).
X = 1,
T = f(1).
```

In the first query the assignment of 2 was undone by the failure of the left branch of the disjunction and the value read is 1; in the second it survived. The third query shows that a non-backtrackable value also survives the end of the query that set it, which is what makes the family a store of settings for a session. The last two show the copy: the term stored with `nb_setval/2` has its own variable, and binding `X` afterward leaves it unbound, where `b_setval/2` stores the term itself.

`flag/3` is a counter with the same lifetime: `flag(Name, Old, New)` unifies `Old` with the current value of the flag, 0 if it was never set, evaluates `New` as an arithmetic expression, and stores the result. It

is the idiom for counting in a failure-driven loop, where an accumulator argument has nowhere to live, and for generating identifiers:

```
?- flag(hits, _, 0),
   (   thread(T, _),
       thread_state(T, waiting(_)),
       flag(hits, N, N + 1),
       fail
   ;   flag(hits, Hits, Hits)
   ).
Hits = 7.

?- flag(next_id, Id, Id + 1).
Id = 0.

?- flag(next_id, Id, Id + 1).
Id = 1.
```

The first query counts the waiting threads with a counter that the failure of each branch does not reset; `aggregate_all(count, ...)` of Chapter 9 is the better way to count solutions, and the loop is shown for the mechanism. The second is a source of fresh integers, which Section 11.7 uses to number annotations.

Global variables are a side channel. A value that one goal sets and a distant goal reads is an argument that the clause heads between them do not show, and a program that leans on them is read with the same difficulty as a procedural program with global state, because that is what it is. Their uses are the ones shown: a counter in a loop that has no accumulator, a session setting such as a threshold that every recognizer should see, and the identifier generator. Values that are the results of an analysis travel in arguments. Facts the analyst has established go into the dynamic database, where `listing/1` can show them and `retract/1` can withdraw them, which a global variable cannot. What neither keeps is anything past the end of the session: dynamic clauses and global values live in the running process and are gone when it halts. `library(persistency)` declares dynamic predicates whose assertions and retractions are appended to a journal file and replayed when the file is attached again, which is how an analyst's annotations survive from one session to the next; Chapter 13, which covers files, is the place for it.

11.6 Tabling

Tabling changes how SWI-Prolog runs a predicate. A predicate declared with the directive `table/1`, written `:- table name/arity.`, is run under **SLG resolution** instead of the SLD resolution of Chapter 4. When a tabled predicate is called, SWI-Prolog looks for a **table** for a variant of the call, a call equal to it up to the renaming of variables. If there is none, it creates one and runs the clauses, adding each answer to the table as it is found; a recursive call that is a variant of a call whose table is still being filled does not run the clauses again but consumes the answers already in that table, and is resumed as more arrive. When no clause can add an answer the table is **complete**, and every later variant call reads it without running anything. Two things follow. A call whose set of answers is finite terminates, however the clauses recurse, because an answer already in the table is not added again

and a call already in progress is not re-entered; and a predicate is computed once per distinct call, which is memoization.

The graph of Chapter 4 that defeated `reaches/2` is a cycle, and the same definition under tabling terminates:

```
% edge(X, Y): a small directed graph with a cycle
% a, b, c, a and a tail to d.
edge(a, b).
edge(b, c).
edge(c, a).
edge(c, d).

% reaches(X, Y): Y is reachable from X by one or
% more edges.  The definition of Chapter 4.
:- table reaches/2.
reaches(X, Y) :-
    edge(X, Y).
reaches(X, Y) :-
    edge(X, Z),
    reaches(Z, Y).

?- setof(Y, reaches(a, Y), Ys).
Ys = [a, b, c, d].

?- setof(X, reaches(X, a), Xs).
Xs = [a, b, c].

?- reaches(d, Y).
false.

?- reaches(a, a).
true.
```

Four answers, each once, and the query ends. The queries collect them with `setof/3` for a reason that is itself a property of tabling: the answers of a tabled call are read from its table in the order the table's internal structure yields them, which is implementation-defined, differs between runs of the same program, and must not be relied on. A program that needs an order sorts, and a transcript that is to be reproducible collects. The last query, `reaches(a, a)`, is the cycle test, and it is the one that did not terminate in Chapter 4.

Left recursion, which Chapter 4 showed fails on every input under SLD resolution because the infinite branch is the first explored, is harmless under tabling: the left-recursive call is a variant of the call in progress and consumes its table instead of descending.

```
% path(X, Y): as reaches/2, written with the
% recursive goal first.
:- table path/2.
path(X, Y) :-
    path(X, Z),
    edge(Z, Y).
path(X, Y) :-
    edge(X, Y).

?- setof(Y, path(a, Y), Ys).
Ys = [a, b, c, d].
```

This is the form in which transitive closure is usually written in the database literature, and it is the natural form, since it says that a path is a path extended by an edge. Tabling lets a Prolog program say it. `listing/1` shows the declaration with the predicate:

```
?- listing(path/2).
:- table path/2 as variant.

path(X, Y) :-
    path(X, Z),
    edge(Z, Y).
path(X, Y) :-
    edge(X, Y).

true.

?- predicate_property(path(_, _), tabled).
true.
```

Tables are kept until they are abolished. A table built over facts that later change is stale, and `abolish_all_tables/0` discards every table so that the next call rebuilds it from the current clauses; a program that asserts into a relation a tabled predicate depends on must call it, or declare the relation and the tabled predicate `incremental` so that SWI-Prolog does the invalidation itself, which is beyond this chapter. The facts of a dump do not change during an analysis, and the tables over them are valid for the session.

Answer subsumption, also called mode-directed tabling, keeps for each call not every answer but the best by some measure. The declaration `:- table dist(_, _, min).` says that the third argument is to be minimized: of two answers that agree on the first two arguments, only the one with the smaller third is kept, and an answer that is not smaller than the stored one is not an answer at all, which makes a definition terminate whose answers would otherwise grow without bound.

```
% dist(X, Y, D): D is the length of the shortest
% path from X to Y.
:- table dist(_, _, min).
dist(X, Y, 1) :-
    edge(X, Y).
dist(X, Y, D) :-
    dist(X, Z, D0),
    edge(Z, Y),
    D is D0 + 1.
```

```
?- setof(Y-D, dist(a, Y, D), Ds).
Ds = [a-3, b-1, c-2, d-3].

?- dist(a, a, D).
D = 3.
```

Without the `min` the second clause would produce distances 3, 6, 9, and so on around the cycle without end; with it, the distance 6 to `a` is not smaller than the 3 already stored and the computation stops. Other modes include `max`, `first`, `last`, `sum`, and `lattice(Pred)`, where `Pred` is a predicate of three arguments that combines two answers into the one to keep. The lattice mode is what makes a shortest path, rather than its length, computable on a cyclic graph:

```
% shorter(P1, P2, P): P is the shorter of the two
% lists P1 and P2, P1 when they are equally long.
shorter(P1, P2, P) :-
    length(P1, L1),
    length(P2, L2),
    (   L1 =< L2
    ->  P = P1
    ;   P = P2
    ).

% route(X, Y, Path): Path is a shortest list of
% vertices from X to Y along edges.
:- table route(_, _, lattice(shorter/3)).
route(X, Y, [X, Y]) :-
    edge(X, Y).
route(X, Y, Path) :-
    route(X, Z, Path0),
    edge(Z, Y),
    append(Path0, [Y], Path).

?- route(a, d, Path).
Path = [a, b, c, d].

?- route(a, a, Path).
Path = [a, b, c, a].

?- route(c, c, Path).
Path = [c, a, b, c].
```

Each extension of a path around the cycle produces a longer list, `shorter/3` keeps the one already stored, and the table completes with one path per pair of vertices. Section 11.7 uses exactly this to obtain the deadlock of the dump as a sequence of threads, which Chapter 10's `library(ugraphs)` could not give.

The second use of tabling is memoization of an expensive predicate. The usual demonstration is the Fibonacci function, whose direct definition recomputes each value exponentially many times:

```
% fib(N, F): F is the Nth Fibonacci number.
fib(N, F) :-
    (   N < 2
    ->  F = N
    ;   N1 is N - 1,
        N2 is N - 2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1 + F2
    ).

% tfib(N, F): as fib/2, tabled.
:- table tfib/2.
tfib(N, F) :-
    (   N < 2
    ->  F = N
    ;   N1 is N - 1,
        N2 is N - 2,
        tfib(N1, F1),
        tfib(N2, F2),
        F is F1 + F2
    ).

?- time(fib(25, F)).
% 728,355 inferences, <...> CPU in <...>
F = 75025.

?- time(tfib(25, F)).
% <...> inferences, <...> CPU in <...>
F = 75025.

?- time(tfib(25, F)).
% <...> inferences, <...> CPU in <...>
F = 75025.

?- tfib(200, F).
F = 280571172992510140037611932413038677189525.
```

`time/1` runs a goal and reports the number of inferences, logical steps, it took, with the CPU and wall-clock time; the figures that vary between runs are elided. The untabled definition takes seven hundred thousand inferences for `fib(25)`; the tabled one takes about a thousand the first time, because each `tfib(N)` is computed once and read from its table thereafter, and a handful the second time, when the table for the call is complete. The 200th number, which the untabled definition could not compute in the lifetime of the reader, is an unbounded integer of Chapter 2. In an analysis the expensive predicates are not arithmetic but the ones that walk every frame of every thread or every message of a trace, and tabling the ones that are called many times with the same arguments, a stack trace signature per thread, a per-component message count, is the cheapest optimization available; Chapter 30 measures it.

Two cautions. A tabled predicate’s body runs once per call variant and its answers are then served from the table, so a body with side effects, output or assertion, performs them once rather than once per call, and a tabled predicate should be pure. And tabling trades space for time: every table is kept until abolished, and tabling a predicate called with many distinct arguments, such as one that

takes a message text, fills memory with tables that are read once. The predicates to table are the recursive ones over cyclic or shared structure and the expensive ones with few distinct calls.

11.7 Diagnostic Use

The `blocked_by/2` of Chapter 4 and its transitive closure `blocked_through/2`, which did not terminate on the deadlock in `dump1.pl`, come back with one line added:

```
% blocked_by(Waiter, Owner): Waiter waits for a
% critical section that Owner owns.
blocked_by(Waiter, Owner) :-
    waits_for(Waiter, critical_section, Addr),
    critical_section(Addr, Owner, _, _),
    Owner \== 0.

% blocked_through(TID, Other): TID is blocked by
% Other directly or through a chain of owners.
% The definition of Chapter 4, tabled.
:- table blocked_through/2.
blocked_through(TID, Other) :-
    blocked_by(TID, Other).
blocked_through(TID, Other) :-
    blocked_by(TID, Owner),
    blocked_through(Owner, Other).

?- setof(X, blocked_through(101, X), Xs).
Xs = [103, 104].

?- setof(X, blocked_through(103, X), Xs).
Xs = [103, 104].

?- blocked_through(102, X).
false.

?- setof(T, blocked_through(T, T), Ts).
Ts = [103, 104].
```

The query that Chapter 4 showed producing 103, 104, 103, 104 without end now produces each thread once and stops; the answers are collected with `setof/3`, as in Section 11.6, because their order out of the table is not defined. The set of threads a thread is blocked through is the **Wait Chain (Critical Sections)** (page 384) of that thread as a set, the same set `reachable/3` gave in Chapter 10, and the last query is the **Deadlock (Critical Sections)** (page 388) test of Chapter 4 asked without a fixed length: a thread blocked through a chain by itself is on a cycle. The visited list of Chapter 5 could not ask it, because a search that refuses to revisit its start cannot find its way back to it; the table can, because the answer `blocked_through(103, 103)` is an answer like any other, found once and stored.

A set of threads is not a chain. An analyst reporting a wait chain gives it as a sequence, the waiter, the critical section it waits for, that section's owner, what the owner waits for, and so on to the end,

and reporting a deadlock gives the cycle in the same form. The lattice mode of Section 11.6 produces the sequence:

```
% wait_chain(From, To, Chain): Chain is a shortest
% list of threads from From to To in which each
% thread is blocked by the next.
:- table wait_chain(_, _, lattice(shorter/3)).
wait_chain(From, To, [From, To]) :-
    blocked_by(From, To).
wait_chain(From, To, Chain) :-
    wait_chain(From, Mid, Chain0),
    blocked_by(Mid, To),
    append(Chain0, [To], Chain).

% deadlock_cycle(Cycle): Cycle is a wait chain
% from a thread back to itself.
deadlock_cycle(Cycle) :-
    wait_chain(TID, TID, Cycle).

?- setof(To-Chain, wait_chain(101, To, Chain),
   Chains).
Chains = [103-[101, 103], 104-[101, 103, 104]].

?- setof(Cycle, deadlock_cycle(Cycle), Cycles).
Cycles = [[103, 104, 103], [104, 103, 104]].
```

Thread 101 is blocked by 103 and, through 103, by 104; the chain to 104 is the whole of the wait chain from 101, and it ends where the cycle begins. The deadlock is reported twice, once from each of its threads, because the two rotations of a cycle are different lists and the lattice keeps the shortest chain per pair of endpoints, not per cycle; Chapter 20 chooses a canonical rotation. The chain is printed with the critical section at each step, which the list does not hold and `waits_for/3` supplies, using the `nextto/3` of Chapter 10 to walk consecutive pairs:

```
% print_chain(Chain): one line per step of the
% chain, with the critical section waited for.
print_chain(Chain) :-
    forall(nextto(Waiter, Owner, Chain),
           print_step(Waiter, Owner)).

% print_step(Waiter, Owner): one line of a chain.
print_step(Waiter, Owner) :-
    waits_for(Waiter, critical_section, Addr),
    format("~d waits for ~16r, owner ~d~n",
           [Waiter, Addr, Owner]).

?- wait_chain(101, 104, _Chain),
   print_chain(_Chain).
101 waits for 7ff61d13a0c0, owner 103
103 waits for 7ff61d13a140, owner 104
true.
```

```
?- once(( deadlock_cycle(_Cycle),
          _Cycle = [103 | _] )),
   print_chain(_Cycle).
103 waits for 7ff61d13a140, owner 104
104 waits for 7ff61d13a0c0, owner 103
true.
```

The first printout is what an analyst writes down from `!locks` and three stack traces, and it is now derived. The second selects the rotation of the cycle that begins with thread 103 by unifying after the call rather than in it: an argument under answer subsumption must be unbound when the tabled predicate is called, and SWI-Prolog raises an error when it is not. The `once/1` discards the choice point that the other rotation would otherwise leave or not leave, depending on the order the table yields them. What the predicates do not say is as before: the facts record that 101 waits for the section 103 owns, and nothing about how long or whether 103 would have released it had 104 not held the other; the recognizer finds the structure, and the analyst judges it. Chapter 20 treats both patterns with the extensions, mutexes and thread objects, that the dump schema allows.

Annotations are the second use. During an analysis the analyst establishes facts the dump does not contain, that a module's vendor is a known cache library, that a thread is a worker of a particular pool, that a critical section guards a particular table, and the conclusions so far, that two threads are deadlocked and a third is held up by them. These belong in the knowledge base, beside the dump's facts and queryable with them, and in a dynamic predicate of their own, since the dump's predicates are static and must stay so:

```
% annotation(Id, Subject, Note): the analyst's
% note number Id about Subject, a term such as
% thread(TID) or module(Name).
:- dynamic annotation/3.

% annotate(Subject, Note): record Note about
% Subject under the next free identifier.
annotate(Subject, Note) :-
    flag(annotation_id, Id0, Id0 + 1),
    Id is Id0 + 1,
    assertz(annotation(Id, Subject, Note)).

% withdraw(Id): remove annotation Id.
withdraw(Id) :-
    retract(annotation(Id, _, _)).

?- annotate(thread(103), "owns A, waits for B").
true.

?- annotate(thread(104), "owns B, waits for A").
true.

?- annotate(module(cachelib), "CacheWorks").
true.

?- annotate(thread(101), "held up by deadlock").
true.
```

```
?- listing(annotation/3).
:- dynamic annotation/3.

annotation(1, thread(103), "owns A, waits for B").
annotation(2, thread(104), "owns B, waits for A").
annotation(3, module(cachelib), "CacheWorks").
annotation(4, thread(101), "held up by deadlock").

true.
```

The subject is a term, so that annotations about threads, modules, and critical sections are distinguished by their principal functor and found by unification, and the identifier comes from `flag/3`, so that a note can be withdrawn by number. The annotations join the dump's facts in queries:

```
?- annotation(_, thread(T), Note),
   thread_state(T, State),
   format("~d ~w: ~s~n", [T, State, Note]), fail.
103 waiting(user_request): owns A, waits for B
104 waiting(user_request): owns B, waits for A
101 waiting(user_request): held up by deadlock
false.

?- deadlock_cycle(_Cycle), member(T, _Cycle),
   \+ annotation(_, thread(T), _).
false.

?- withdraw(3).
true.

?- annotation(Id, module(M), _).
false.

?- annotate(module(cachelib), "no symbols"),
   annotation(Id, module(M), Note).
Id = 5,
M = cachelib,
Note = "no symbols".
```

The second query is a check an analyst can run before writing a report: every thread of the deadlock has been annotated, so the query for one that has not fails. The last two queries withdraw a note and replace it, and show that a withdrawn number is not reused, since `flag/3` only counts up. Facts learned this way have a different standing from the facts of the dump, and the schema keeps them apart by predicate: `module_info/3` says what the debugger reported, and `annotation/3` says what the analyst believes. Chapter 15 makes the distinction explicit with provenance facts that record where each fact came from, and Chapter 17 treats the analyst's conclusions as a third kind, derived from the first and recorded in the manner of the second. The annotations are lost when the session ends unless written to a file; `library(persistency)` does that for a dynamic predicate, and Chapter 13 shows the plain alternative of writing the clauses out with `portray_clause/1` and consulting them back.

Chapter 12. Definite Clause Grammars

The facts of `dump1.pl` and `trace1.pl` did not start as facts. The debugger printed a stack frame as one line of text, `00 00000000`0014f5a8 00007ffb`b0b2a0b0 `kernelbase!SleepEx+0x9e`, and the logging library of the application printed a message as another, and someone had to take those lines apart into the arguments of `frame/5` and `msg/6`. Chapter 8 did this for the symbol part of a frame with `sub_atom/5`, searching for the delimiters by position, and remarked that text with a structure deeper than a sequence of fields needs a grammar. This chapter covers the grammar notation of Prolog, the **definite clause grammar** or DCG: rules written with `-->` that describe a sequence of character codes and are translated into ordinary clauses, run with `phrase/2` and `phrase/3`, and read in both directions, as parsers that take text apart and as generators that put it together. It then covers the two libraries that make grammars over text short, `library(dcg/basics)` for blanks, numbers, and runs of characters, and `library(dcg/high_order)` for repetition, and `phrase_from_file/2`, which runs a grammar over a file without reading the file into memory first.

The diagnostic use is the front of the pipeline that every later chapter assumes: from a debugger's abridged `~*k` output to `frame/5` facts, and from a log file to `msg/6` facts, with the grammars for the two line formats this book's artifacts use, and with the first pattern of each catalog recognized in the text itself, **No Component Symbols** (page 431) in a frame line without a function name and **Error Message** (page 565) in a message that begins with the word.

12.1 Grammar Rules

A **grammar rule** has the form `Head --> Body`. The head names a **nonterminal**, with arguments if it has any, and the body says what sequence of codes the nonterminal stands for: a **terminal** is a list of codes that must appear in the input exactly, a nonterminal in the body stands for whatever its own rules describe, and a comma is concatenation. In SWI-Prolog a double-quoted text in a grammar body is a terminal, the list of its character codes, whatever the `double_quotes` flag of Chapter 2 says; outside grammar bodies it is a string object.

```
:- use_module(library(dcg/basics)).
:- use_module(library(dcg/high_order)).
:- use_module(library(pure_input)).

% method(M): the name of an HTTP method.
method(get) --> "GET".
method(post) --> "POST".

% health_check(M): a request line for the health
% path, with method M.
health_check(M) --> method(M), " /health".
```

A grammar is run with `phrase/2`: `phrase(Body, List)` succeeds when `Body` describes exactly the list of codes `List`. The input is a list of codes, which Chapter 2 distinguished from a string, so a string is converted with `string_codes/2` first; the backquoted notation `` `abc` `` of SWI-Prolog 7 and later writes a code list directly.

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- string_codes("GET /health", _Cs),
   phrase(health_check(M), _Cs).
M = get.

?- string_codes("GET /report/17", _Cs),
   phrase(health_check(M), _Cs).
false.

?- X = `abc`.
X = [97, 98, 99].

?- catch(phrase(health_check(M), "GET /health"),
         error(E, _), true).
E = type_error(list, "GET /health").
```

The grammar is a relation between a term and a text, and like every relation in this book it runs in both directions. With the argument bound and the list unbound, `phrase/2` generates the text, and with both unbound it enumerates the language of the grammar:

```
?- phrase(health_check(post), _Cs),
   string_codes(S, _Cs).
S = "POST /health".

?- phrase(health_check(M), _Cs),
   string_codes(S, _Cs).
M = get,
S = "GET /health" ;
M = post,
S = "POST /health".
```

`phrase/3` is the general form: `phrase(Body, List, Rest)` succeeds when `Body` describes a prefix of `List` and `Rest` is what follows it. `phrase/2` is `phrase/3` with `Rest` fixed to `[]`, the requirement that the whole input be consumed, and `phrase/3` is what a parser uses when it reads one construct and hands the remainder to the next.

```
?- string_codes("GET /health HTTP/1.1", _Cs),
   phrase(health_check(M), _Cs, _Rest),
   string_codes(R, _Rest).
M = get,
R = " HTTP/1.1".
```

12.2 The Translation

A grammar rule is a clause in a notation that hides two arguments. SWI-Prolog translates `Head --> Body` at load time into a clause for a predicate with two more arguments than the head: the list of codes before the nonterminal and the list after it, the difference list of Chapter 5 with its two ends as separate arguments. `listing/1` shows the result:

```
?- listing(method//1).
method(get, [71, 69, 84|A], A).
method(post, [80, 79, 83, 84|A], A).

true.

?- listing(health_check//1).
health_check(M, A, B) :-
    method(M, A, C),
    C=[32, 47, 104, 101, 97, 108, 116, 104|B].

true.
```

The indicator of a nonterminal is written `name//arity` with the arity of the grammar rule, and the predicate it becomes has two more arguments. A terminal becomes a unification of the input with a list whose tail is the remaining input, placed in the head when it begins the body and written as a goal otherwise; a nonterminal becomes a goal with the two lists threaded through, the output of one being the input of the next. `phrase(health_check(M), Cs)` is `health_check(M, Cs, [])`. There is no magic in the notation: everything a grammar does is done by unification on lists, and everything Chapter 4 said about the search applies to the clauses the grammar becomes.

Ordinary goals are placed in a grammar body in braces, `{Goal}`, and are run without touching the input. A hand-written grammar for a natural number shows them, and shows what the search does with a grammar that admits several parses:

```
% digit_code(D): one code that is a decimal digit.
digit_code(D) --> [D], { code_type(D, digit) }.

% digit_codes(Ds): one or more digit codes.
digit_codes([D | Ds]) -->
    digit_code(D),
    digit_codes(Ds).
digit_codes([D]) -->
    digit_code(D).

% nat(N): a natural number written in decimal.
nat(N) -->
    digit_codes(Ds),
    { number_codes(N, Ds) }.

?- listing(nat//1).
nat(N, A, B) :-
    digit_codes(Ds, A, C),
    number_codes(N, Ds),
    B=C.

true.

?- string_codes("123", _Cs), phrase(nat(N), _Cs).
N = 123 ;
false.
```

```
?- string_codes("123abc", _Cs),
   phrase(nat(N), _Cs, _R), string_codes(R, _R).
N = 123,
R = "abc" ;
N = 12,
R = "3abc" ;
N = 1,
R = "23abc".
```

The terminal `[D]` with a variable matches any one code, and the brace goal tests it. The translation of `nat//1` ends in `B=C` rather than passing `B` to the last goal: the goal in braces might fail, and the output list is unified only after it, which keeps the translated clause steadfast in the sense of Chapter 7. The two clauses of `digit_codes//1` both apply to a digit followed by more digits, the first taking the longer run and the second stopping after one, so the grammar has three parses of `123abc` and `phrase/3` produces them on backtracking, longest first because the recursive clause comes first. `phrase/2` on `123` accepts only the parse that consumes everything and leaves the choice point that the `;` shows.

A parser for a fixed format wants one parse, the longest, and the control constructs of Chapter 7 are available in grammar bodies for that purpose: `!` commits to the clause and to the input consumed so far, and if-then-else `( C -> T ; E )` chooses on whether a sub-grammar matches. The deterministic run of digits is written with the latter:

```
% digit_codes_det(Ds): the longest run of one or
% more digit codes, deterministically.
digit_codes_det([D | Ds]) -->
    digit_code(D),
    (   digit_codes_det(Ds)
    ->  []
    ;   { Ds = [] }
    ).

% nat_det(N): as nat/1, deterministically.
nat_det(N) -->
    digit_codes_det(Ds),
    { number_codes(N, Ds) }.

?- string_codes("123abc", _Cs),
   phrase(nat_det(N), _Cs, _R),
   string_codes(R, _R).
N = 123,
R = "abc".

?- string_codes("abc", _Cs),
   phrase(nat_det(N), _Cs, _R).
false.

?- string_codes("42 ms", _Cs),
   phrase((nat_det(N), " ms"), _Cs).
N = 42.
```

The empty terminal `[]` consumes nothing and is the grammar's `true`. The last query runs a body built in the query itself, a conjunction of a nonterminal and a terminal, which `phrase/2` accepts as it

accepts a nonterminal; the body of a grammar rule and the first argument of `phrase/2` are the same kind of term.

Two further body constructs are used in parsers. A rule with a **pushback** list, `Head, PB --> Body`, consumes what `Body` describes and then puts the codes of `PB` back at the front of the remaining input, which is how a grammar looks ahead without consuming; `call//N` calls a closure as a nonterminal with the two list arguments added, so that a grammar can be passed one as a parameter, as `maplist/2` is passed a goal.

```
% at_digit: the next code is a digit, which is not
% consumed.
at_digit, [C] --> [C], { code_type(C, digit) }.

?- listing(at_digit//0).
at_digit([A|B], C) :-
    code_type(A, digit),
    D=B,
    C=[A|D].

true.

?- string_codes("42", _Cs),
   phrase((at_digit, nat_det(N)), _Cs).
N = 42.

?- string_codes("x42", _Cs),
   phrase((at_digit, nat_det(N)), _Cs).
false.

?- string_codes("42", _Cs),
   phrase(call(nat_det, N), _Cs).
N = 42.
```

The translation of `at_digit//0` shows the pushback as a unification of the output list with the code put back in front of the rest. Section 12.4 shows `call//N` at work inside the sequence nonterminals of the library.

12.3 The Basics Library

`library(dcg/basics)` provides the nonterminals every grammar over text needs. They work on lists of codes, classify codes with `code_type/2`, and are written to be deterministic where a format parser wants them to be, taking the longest match:

Nonterminal	Describes
`blank//0`, `blanks//0`	one, any number of layout codes
`white//0`, `whites//0`	the same, excluding newlines
`nonblank//1`, `nonblanks//1`	one, a run of non-layout codes
`digit//1`, `digits//1`	one, any number of digit codes
`integer//1`	an optionally signed integer

Nonterminal	Describes
`number//1`	an integer or a float
`xinteger//1`	hexadecimal digits, as an integer
`string//1`	any codes, shortest first
`string_without//2`	codes up to one of the given set
`remainder//1`	all remaining codes
`eol//0`	a newline or the end of input
`eos//0`	the end of input
`atom//1`	the codes of an atom, for output

```
?- string_codes("  42", _Cs),
   phrase((blanks, integer(N)), _Cs).
N = 42.

?- string_codes("-7", _Cs),
   phrase(integer(X), _Cs).
X = -7.

?- string_codes("0101", _Cs),
   phrase(integer(X), _Cs).
X = 101.

?- string_codes("1.234", _Cs),
   phrase(number(X), _Cs).
X = 1.234.

?- string_codes("1f8000", _Cs),
   phrase(xinteger(X), _Cs).
X = 2064384.

?- string_codes("x", _Cs),
   phrase(digits(D), _Cs, _R),
   string_codes(R, _R).
D = [],
R = "x".
```

`integer//1` reads an optional sign and one or more digits, so a zero-padded field such as a thread identifier printed as `0101` is read as the integer 101; `number//1` reads a float when there is a fraction or exponent and an integer otherwise; `xinteger//1` reads hexadecimal digits without a `0x` prefix, which the debugger's address columns do not carry. `digits//1` accepts an empty run, which is why its result on `x` is `[]` with the input untouched, where `integer//1` fails on no digits.

`blanks//0` and `whites//0` differ in one code: a newline is a blank and not a white, so a grammar for one line of a file skips its indentation with `whites` and does not run into the next line. `nonblanks//1` is the run up to the next layout code, which is a word in the simplest sense; `string_without//2` is the run up to the first occurrence of any code in its first argument, which may be given as a string, and is the usual way to read a field delimited by a known character; `string//1` describes any sequence of codes and, being nondeterministic from shortest to longest, is used only when what follows it fixes where it ends. `remainder//1` takes everything that is left, `eos//0` succeeds only at the end of the input, and `eol//0` consumes a newline or succeeds at the end.

```
?- string_codes("a1 b2", _Cs),
   phrase((nonblanks(A), white, nonblanks(B)),
          _Cs),
   string_codes(SA, A), string_codes(SB, B).
A = [97, 49],
B = [98, 50],
SA = "a1",
SB = "b2".

?- string_codes("abc,def", _Cs),
   phrase(string_without(",", L), _Cs, _R),
   string_codes(S, L), string_codes(R, _R).
L = [97, 98, 99],
S = "abc",
R = ",def".

?- string_codes("abc def", _Cs),
   phrase((string(L), " def"), _Cs),
   string_codes(S, L).
L = [97, 98, 99],
S = "abc" ;
false.

?- string_codes("\n x", _Cs),
   phrase((whites, "x"), _Cs).
false.

?- string_codes("\n x", _Cs),
   phrase((blanks, "x"), _Cs).
true.

?- string_codes("abc", _Cs),
   phrase((string_without("\n", L), eol), _Cs),
   string_codes(S, L).
L = [97, 98, 99],
S = "abc".
```

The nonterminals that yield codes yield a list of integers, as the first query shows, and the grammar that uses them converts with `string_codes/2` or `atom_codes/2` in a brace goal, choosing the representation Chapter 8 recommended for the field: an atom for a name that will be compared, a string for text that will be searched. `string_without//2` does not consume the delimiter, which is left for the grammar to match explicitly, as the second query shows with the comma still at the head of the rest.

12.4 Sequences

Repetition is written in `library(dcg/high_order)` with the `sequence` nonterminals, which take a nonterminal for the element as a closure, as `maplist/2` takes a goal, and call it with `call//N` once per element. `sequence//2` describes zero or more elements, `sequence//3` elements separated by a separator grammar, and `sequence//5` the same between an opening and a closing grammar:

```
?- string_codes("1,2,3", _Cs),
   phrase(sequence(integer, ",", L), _Cs).
L = [1, 2, 3].

?- string_codes("[1,2,3]", _Cs),
   phrase(sequence("[", integer, ",", "]", L),
          _Cs).
L = [1, 2, 3].

?- phrase(sequence(integer, ",", [1, 2, 3]), _Cs),
   string_codes(S, _Cs).
S = "1,2,3".

?- string_codes("12ab", _Cs),
   phrase(sequence(digit, L), _Cs, _R),
   string_codes(R, _R).
L = [49, 50],
R = "ab" ;
L = [49],
R = "2ab" ;
L = [],
R = "12ab".
```

The third query generates: the list is given and the text is produced, since `integer//1` and the terminal separator both work in that direction. The last shows that `sequence//2` is nondeterministic, taking the most elements first and fewer on backtracking, which is right for a grammar that is followed by something that decides and wasteful for a format parser; a run of a known kind of code is better read with `digits//1` or `nonblanks//1`, which are deterministic. The element and separator grammars must each consume at least one code, since a sequence of elements that may be empty has infinitely many parses of any input, and `sequence(nonblanks, whites, Ws)` does not terminate for that reason.

12.5 A Frame Line

A line of the debugger's `k` output for a 64-bit process has four fields separated by spaces: the frame number in two hexadecimal digits, the child stack pointer and the return address as sixteen hexadecimal digits each with a backquote after the eighth, and the symbol in the form Chapter 8 decoded, `module!function+0xoffset` or `module+0xoffset` when the debugger had no symbol. The grammar has one nonterminal per field. The address nonterminal reads the two halves and combines them with the shift of Chapter 6; the symbol nonterminal reads the module name up to the first `!` or `+`, then decides on the `!` whether a function name follows:

```
% addr64(A): a 64-bit address as the debugger
% prints it, two groups of hex digits around `.
addr64(A) -->
    xinteger(Hi), "`", xinteger(Lo),
    { A is Hi << 32 \/ Lo }.
```

```
% symbol(M, F, O): the symbol M!F+0xO, or M+0xO
% with F = 'Unknown'.
symbol(M, F, O) -->
    string_without("!+", MC),
    { atom_codes(M, MC) },
    (   "!"
    ->  string_without("+", FC),
        { atom_codes(F, FC) }
    ;   { F = 'Unknown' }
    ),
    "+0x",
    xinteger(O).

% frame_line(TID, Frame): one line of k output for
% thread TID, as a frame/5 term.
frame_line(TID, frame(TID, N, M, F, O)) -->
    xinteger(N), blank,
    addr64(_), blank,
    addr64(_), blank,
    symbol(M, F, O).


?- string_codes("00007ffb`b0b2a0b0", _Cs),
   phrase(addr64(A), _Cs), format("~16r~n", [A]).
7ffbb0b2a0b0
A = 140718978015408.

?- string_codes("dbclient!Buffer::Copy+0x1a",
   _Cs), phrase(symbol(M, F, O), _Cs).
M = dbclient,
F = 'Buffer::Copy',
O = 26.

?- string_codes("cachelib+0x2a10", _Cs),
   phrase(symbol(M, F, O), _Cs).
M = cachelib,
F = 'Unknown',
O = 10768.

?- string_codes("ucrtbase!thread_start<...>+0x42",
   _Cs), phrase(symbol(M, F, O), _Cs).
M = ucrtbase,
F = 'thread_start<...>',
O = 66.
```

The three symbols are the three cases Chapter 8's `symbol_parts/4` handled with two `sub_atom/5` searches and an if-then-else; the grammar says the same in the order the text is read, and the if-then-else on the `!` terminal is what makes the two forms one nonterminal. The function name is read up to the `+`, so the template brackets and the scope operator pass through as before, and a name containing `+` would be cut short as before; that is a property of the format, not of the parser.

The whole line is a conjunction of fields with `blank` between them. The thread identifier is not on the line; the `~*k` output names the thread in a header before its frames, and Section 12.8 reads it from there, so `frame_line//2` takes it as a parameter. The sample lines are kept as facts, written across two source lines with the escape `\c`, which in SWI-Prolog skips the layout that follows it inside a quoted text and is not part of ISO Prolog:

```
% k_line(N, Line): the Nth line of an abridged k
% output, as a string.
k_line(0,
       "00 00000000`0014f5a8 00007ffb`b0b2a0b0 \c
        kernelbase!SleepEx+0x9e").
k_line(1,
       "01 00000000`0014f5b0 00007ff6`1d10a3a0 \c
        appserver!MainLoop+0x1b2").
k_line(2,
       "02 00000000`0014f620 00007ffb`7a002a10 \c
        cachelib+0x2a10").

?- k_line(0, _L), string_codes(_L, _Cs),
   phrase(frame_line(100, F), _Cs).
F = frame(100, 0, kernelbase, 'SleepEx', 158).

?- forall(k_line(_, L),
          ( string_codes(L, Cs),
            phrase(frame_line(100, F), Cs),
            print(F), nl
          )).
frame(100,0,kernelbase,'SleepEx',158)
frame(100,1,appserver,'MainLoop',434)
frame(100,2,cachelib,'Unknown',10768)
true.
```

Each line becomes a `frame/5` term of the shape the schema prescribes, with the offset in decimal as the toplevel prints integers. The two addresses are read and dropped, `addr64(_)`, because the schema does not keep them; Chapter 15 discusses what is lost by that choice and how a `frame/6` with the return address would be used.

The reverse direction is a separate grammar, because `symbol//3` reads the module name with `string_without//2` before the brace goal that converts it, and run backward it would have to generate codes before knowing the atom. A generating grammar puts the brace goals first and uses `atom//1` of the basics library, which emits the codes of an atom, and `xinteger//1`, which emits hexadecimal digits when its argument is bound. The frame number is formatted with the zero-padded column stop of Chapter 6 into a code list through the `codes(Cs)` sink of `format/3`:

```
% symbol_text(M, F, O): the debugger's text for a
% symbol, for output.
symbol_text(M, F, O) -->
    atom(M),
    (   { F == 'Unknown' }
    ->  []
    ;   "!", atom(F)
    ),
    "+0x", xinteger(O).

% frame_text(Frame): a frame as a numbered line
% without its address columns, for output.
frame_text(frame(_, N, M, F, O)) -->
    { format(codes(NC), "~`0t~16r~2|", [N]) },
    NC, " ",
    symbol_text(M, F, O).
```

```
?- phrase(symbol_text(cachelib, 'Unknown', 10768),
   _Cs), string_codes(S, _Cs).
S = "cachelib+0x2a10".

?- k_line(1, _L), string_codes(_L, _Cs),
   phrase(frame_line(100, F), _Cs),
   phrase(frame_text(F), _Out),
   string_codes(S, _Out).
F = frame(100, 1, appserver, 'MainLoop', 434),
S = "01 appserver!MainLoop+0x1b2".
```

A code list held in a variable, `NC`, is a terminal like a quoted one. The round trip from text to term to text is the test that Chapter 8 ran on all 78 frames, and the generating grammar is the beginning of the report writers of Chapter 23, which produce text from terms by the same means.

12.6 A Log Line

The log format this book's traces come from is `2026-10-03 12:00:01.234 [4120:0101] appserver: Request 1 received`: a date, a time with milliseconds, the process and thread identifiers in brackets separated by a colon, the component name followed by a colon and a space, and the message text to the end of the line. The target is the `msg/6` term of the schema, whose first argument is the ordinal number of the message in the trace, which the line does not carry and the grammar takes as a parameter, and whose second is the time in seconds. The grammar converts the time of day to seconds since midnight and reads the date without using it; Chapter 16 rebases the time to the start of the trace and discusses what to do with the date.

```
% date(date(Y, M, D)): a date as YYYY-MM-DD.
date(date(Y, M, D)) -->
    integer(Y), "-", integer(M), "-", integer(D).

% time_of_day(T): HH:MM:SS.mmm as seconds since
% midnight.
time_of_day(T) -->
    integer(H), ":", integer(Mi), ":", number(S),
    { T is H * 3600 + Mi * 60 + S }.

% log_line(N, Msg): one line of the log, as the
% msg/6 term with ordinal number N.
log_line(N, Msg) -->
    date(_), " ",
    time_of_day(Time), " [",
    integer(PID), ":", integer(TID), "] ",
    string_without(":", CC), ": ",
    string_without("\n", TC),
    { atom_codes(Comp, CC),
      string_codes(Text, TC),
      Msg = msg(N, Time, PID, TID, Comp, Text)
    }.

% sample_msg(Line): one log line as a string.
sample_msg("2026-10-03 12:00:01.234 [4120:0101] \c
            appserver: Request 1 received").
```

```
?- string_codes("12:00:01.234", _Cs),
   phrase(time_of_day(T), _Cs).
T = 43201.234.

?- string_codes("2026-10-03", _Cs),
   phrase(date(D), _Cs).
D = date(2026, 10, 3).

?- sample_msg(_S), string_codes(_S, _Cs),
   phrase(log_line(1, msg(N, T, P, TID, C, Text)),
          _Cs).
N = 1,
T = 43201.234,
P = 4120,
TID = 101,
C = appserver,
Text = "Request 1 received".
```

Each field is read by the library nonterminal that fits it. `integer//1` reads the zero-padded thread identifier as 101, the number of a worker thread of the dump; `number//1` reads the seconds with their fraction as a float, so that the time is a float as the schema requires; the component is the run up to the first colon, since a component name contains none and the text that follows may contain several; and the text is everything up to the end of the line, converted to a string because Part IV searches it. The two literals `"-"` and `":"` between integers deserve a remark: `integer//1` accepts a leading sign, and a grammar that read `integer(Y), integer(M)` without the separator would read `-10` as a negative month. The separators are matched explicitly, and the grammar reads what the format says and nothing more.

12.7 Grammars over Files

A grammar runs over a list, and a file is not one. `phrase_from_file/2` from `library(pure_input)` runs a grammar over the contents of a file as if the file had been read into a code list, without doing so: the list is a **lazy list**, whose tail is produced from the file as the grammar reaches it and whose consumed prefix can be reclaimed, so that a log of any size is parsed in memory proportional to what the grammar has not yet committed to. The grammar must describe the whole file, as with `phrase/2`, and `eos//0` is how it recognizes the end.

A grammar for a file of log lines reads lines until the end, numbering them as it goes. The if-then-else on `eos` makes it deterministic, and the ordinal number is an accumulator of Chapter 5 passed down the recursion:

```
% log_lines(N, Msgs): the msg/6 terms of the lines
% from here to the end of input, numbered from N.
log_lines(N, Msgs) -->
    (   eos
    ->  { Msgs = [] }
    ;   log_line(N, Msg), "\n",
        { Msgs = [Msg | Msgs1],
          N1 is N + 1
        },
        log_lines(N1, Msgs1)
    ).
```

The file is written by Prolog itself, since this chapter cannot assume one exists: `open/3` opens a file for writing and binds its third argument to a stream, `format/3` with a stream as its first argument prints to it, and `close/1` closes it; `setup_call_cleanup/3` of Chapter 7 guarantees the close. Chapter 13 covers files and streams in general; here they serve to produce a six-line log.

```
% sample_log_line(Line): a line of the sample log.
sample_log_line(
    "2026-10-03 12:00:01.234 [4120:0100] \c
     appserver: Service starting").
sample_log_line(
    "2026-10-03 12:00:01.235 [4120:0101] \c
     appserver: Worker 101 ready").
sample_log_line(
    "2026-10-03 12:00:02.000 [4120:0101] \c
     dbclient: Query start: 17").
sample_log_line(
    "2026-10-03 12:00:02.044 [4120:0101] \c
     dbclient: Query end: 44 ms").
sample_log_line(
    "2026-10-03 12:00:07.002 [4120:0100] \c
     appserver: Error: no worker").
sample_log_line(
    "2026-10-03 12:00:09.301 [4120:0101] \c
     appserver: Request 5 done").

% write_sample_log(File): write the sample log to
% File, one line per sample_log_line/1 fact.
write_sample_log(File) :-
    setup_call_cleanup(
        open(File, write, Out),
        forall(sample_log_line(Line),
               format(Out, "~s~n", [Line])),
        close(Out)).
```

The file written is the following, each line a message of the format of Section 12.6:

```
2026-10-03 12:00:01.234 [4120:0100] appserver: Service starting
2026-10-03 12:00:01.235 [4120:0101] appserver: Worker 101 ready
2026-10-03 12:00:02.000 [4120:0101] dbclient: Query start: 17
2026-10-03 12:00:02.044 [4120:0101] dbclient: Query end: 44 ms
2026-10-03 12:00:07.002 [4120:0100] appserver: Error: no worker
2026-10-03 12:00:09.301 [4120:0101] appserver: Request 5 done
```

```
?- write_sample_log('sample.log').
true.

?- phrase_from_file(log_lines(1, _Msgs),
                    'sample.log'),
   length(_Msgs, N).
N = 6.

?- phrase_from_file(log_lines(1, _Msgs),
                    'sample.log'),
   nth1(5, _Msgs, msg(N, T, P, TID, C, Text)).
N = 5,
T = 43207.002,
P = 4120,
TID = 100,
C = appserver,
Text = "Error: no worker".
```

The parsed terms are facts in all but name, and the dynamic database of Chapter 11 makes them facts: a dynamic `msg/6` is declared and the list is asserted with `maplist/2` of Chapter 10, after which the parsed trace is queried exactly as `trace1.pl` was.

```
% msg(N, Time, PID, TID, Component, Text): the
% schema relation, filled from a parsed log.
:- dynamic msg/6.

% load_log(File): parse File and assert its
% messages as msg/6 facts.
load_log(File) :-
    phrase_from_file(log_lines(1, Msgs), File),
    maplist(assertz, Msgs).

?- load_log('sample.log').
true.

?- msg(N, _, _, 101, C, Text).
N = 2,
C = appserver,
Text = "Worker 101 ready" ;
N = 3,
C = dbclient,
Text = "Query start: 17" ;
N = 4,
C = dbclient,
Text = "Query end: 44 ms" ;
N = 6,
C = appserver,
Text = "Request 5 done".

?- aggregate_all(count, msg(_, _, _, _, _, _), N).
N = 6.
```

This is the pipeline of Part II in miniature: a text artifact, a grammar for its line format, `phrase_from_file/2`, and a knowledge base of facts. The alternative to asserting is to write the terms to a file with `portray_clause/1` and consult it, which Chapter 13 shows and Chapter 15

recommends for artifacts that will be analyzed more than once, since a consulted file is static, indexed, and reloadable, and the parse is done once.

12.8 Diagnostic Use

The `~*k` command of the debugger prints the stack trace of every thread, each introduced by a header line that names the thread and followed by a column header and the frame lines of Section 12.5. An abridged sample, with the symbols shortened to fit the page:

```
.  0  Id: 1018.64 Suspend: 1 Teb: 00000000`0025d000 Unfrozen
 # Child-SP          RetAddr           Call Site
00 00000000`0014f5a8 00007ffb`b0b2a0b0 kernelbase!SleepEx+0x9e
01 00000000`0014f5b0 00007ff6`1d10a3a0 appserver!MainLoop+0x1b2
   1  Id: 1018.65 Suspend: 1 Teb: 00000000`00260000 Unfrozen
 # Child-SP          RetAddr           Call Site
00 00000000`0024f8c8 00007ffb`b0b2a0b0 kernelbase!SleepEx+0x9e
01 00000000`0024f8d0 00007ffb`7a002a10 cachelib+0x2a10
```

The header line begins with a one-character mark, a period for the current thread and a space otherwise, then the thread's ordinal in the debugger's numbering, then `Id:` followed by the process and thread identifiers in hexadecimal separated by a period, and then fields the schema does not use. The process identifier `1018` is 4120 and the thread identifiers `64` and `65` are 100 and 101, the first two threads of `dump1.pl`. The grammar for the whole output is three nonterminals: the header, which yields the thread identifier; the column header, which is skipped; and the frame lines, read with the `frame_line//2` of Section 12.5 until a line that is not one, which is the next header or the end.

```
% thread_header(TID): the line that introduces a
% thread in ~*k output; TID is read from Id:.
thread_header(TID) -->
    [_], blanks, integer(_), blanks,
    "Id: ", xinteger(_), ".", xinteger(TID),
    string_without("\n", _), "\n".

% column_header: the # Child-SP ... line.
column_header -->
    blanks, "#", string_without("\n", _), "\n".

% frame_lines(TID, Frames): the frame lines that
% follow, each a frame/5 term of thread TID.
frame_lines(TID, Frames) -->
    (   frame_line(TID, F), "\n"
    ->  { Frames = [F | Fs] },
        frame_lines(TID, Fs)
    ;   { Frames = [] }
    ).
```

```
% thread_block(Frames): one thread's header and
% frames.
thread_block(Frames) -->
    thread_header(TID),
    column_header,
    frame_lines(TID, Frames).

% k_output(Frames): the frames of every thread in
% the output, in order.
k_output(Frames) -->
    (   eos
    ->  { Frames = [] }
    ;   thread_block(Fs),
        k_output(Rest),
        { append(Fs, Rest, Frames) }
    ).
```

The mark is read with the anonymous terminal `[_]`, one code of any value, and the ordinal and the fields after the identifiers are read and dropped. `frame_lines//2` is deterministic by the same if-then-else as the digit run of Section 12.2, and a line that does not parse as a frame ends the block without being consumed, so that `k_output//1` finds it as the next header. The sample is written to a file as the log was:

```
% sample_k_line(Line): a line of the abridged ~*k
% output.
sample_k_line(
    ".  0  Id: 1018.64 Suspend: 1 \c
     Teb: 00000000`0025d000 Unfrozen").
sample_k_line(
    " # Child-SP          RetAddr           \c
     Call Site").
sample_k_line(
    "00 00000000`0014f5a8 00007ffb`b0b2a0b0 \c
     kernelbase!SleepEx+0x9e").
sample_k_line(
    "01 00000000`0014f5b0 00007ff6`1d10a3a0 \c
     appserver!MainLoop+0x1b2").
sample_k_line(
    "   1  Id: 1018.65 Suspend: 1 \c
     Teb: 00000000`00260000 Unfrozen").
sample_k_line(
    " # Child-SP          RetAddr           \c
     Call Site").
sample_k_line(
    "00 00000000`0024f8c8 00007ffb`b0b2a0b0 \c
     kernelbase!SleepEx+0x9e").
sample_k_line(
    "01 00000000`0024f8d0 00007ffb`7a002a10 \c
     cachelib+0x2a10").
```

```
% write_sample_k(File): write the sample output.
write_sample_k(File) :-
    setup_call_cleanup(
        open(File, write, Out),
        forall(sample_k_line(Line),
               format(Out, "~s~n", [Line])),
        close(Out)).

% frame(TID, N, Module, Function, Offset): the
% schema relation, filled from parsed k output.
:- dynamic frame/5.

% load_k(File): parse the ~*k output in File and
% assert its frames.
load_k(File) :-
    phrase_from_file(k_output(Frames), File),
    maplist(assertz, Frames).

% print_frames(TID): the stack trace of TID as
% numbered lines, from the parsed frames.
print_frames(TID) :-
    forall(frame(TID, N, M, F, O),
           ( Frame = frame(TID, N, M, F, O),
             phrase(frame_text(Frame), Codes),
             format("~s~n", [Codes])
           )).

?- write_sample_k('k.txt').
true.

?- phrase_from_file(k_output(_Fs), 'k.txt'),
   length(_Fs, N).
N = 4.

?- load_k('k.txt').
true.

?- frame(T, N, M, F, O).
T = 100,
N = 0,
M = kernelbase,
F = 'SleepEx',
O = 158 ;
T = 100,
N = 1,
M = appserver,
F = 'MainLoop',
O = 434 ;
T = 101,
N = 0,
M = kernelbase,
F = 'SleepEx',
O = 158 ;
T = 101,
N = 1,
M = cachelib,
F = 'Unknown',
O = 10768.
```

Four frames of two threads, in the shape of `dump1.pl`, from text that the debugger could have printed. Every predicate of the earlier chapters now applies to them, and two patterns are already visible in the parse itself. The frame of thread 101 in `cachelib` has no function name, because its symbol had no `!`, and `symbol//3` recorded `'Unknown'`: this is the **No Component Symbols** (page 431) pattern of Chapter 8 recognized at the point where the text enters the knowledge base, before any `module_info/3` fact exists to confirm it. The parser is where the analyst first learns that a module has no symbols, and a parser that counts the frames it had to mark unknown, per module, produces the pattern's instances as a by-product of loading.

```
?- frame(T, N, M, 'Unknown', _).
T = 101,
N = 1,
M = cachelib.

?- print_frames(100).
00 kernelbase!SleepEx+0x9e
01 appserver!MainLoop+0x1b2
true.
```

The second query prints the **Stack Trace** (page 329) of thread 100 with the generating grammar of Section 12.5, which is the `k` output with its address columns removed, regenerated from the facts the parser produced from it.

The log side has its own pattern in the text. An **Error Message** (page 565) is a message whose text announces an error, which in this log format is a text beginning with the word `Error` and a colon; it is the first thing an analyst searches a log for, and the search is a grammar run over the text of each message with `phrase/3`, since only the prefix matters:

```
% error_prefix: the text begins with "Error:".
error_prefix --> "Error:".

% error_message(N): message N is an Error Message:
% its text begins with the error prefix.
error_message(N) :-
    msg(N, _, _, _, _, Text),
    string_codes(Text, Codes),
    phrase(error_prefix, Codes, _).

?- error_message(N), msg(N, T, _, TID, C, Text).
N = 5,
T = 43207.002,
TID = 100,
C = appserver,
Text = "Error: no worker" ;
false.
```

The recognizer is a grammar used as a test, and `phrase/3` with the rest unbound asks only whether the text starts as the grammar says. Chapter 27 treats **Error Message** (page 565) with the other message patterns and extends the prefix to the forms other logging libraries use; the point here is that a pattern over text is a grammar, and the same notation that loaded the log recognizes patterns

in it. The grammars of this chapter are the parsers that Chapters 15 and 16 assume when they describe the knowledge bases of dumps and traces, and the schema those chapters set out is the one the grammars here produce.

Chapter 13. Programs, Modules, Files, and Tooling

The programs of the previous twelve chapters were loaded into one namespace, run from the toplevel, and fed from two files that were themselves Prolog source. A program that is meant to be used, by its author a month later or by a colleague, needs more: a boundary between the predicates it offers and the helpers it keeps to itself, a way to run from the command line on an artifact named as an argument, a way to read artifacts that are not Prolog source, and tests that say whether a change broke it. This chapter covers the SWI-Prolog machinery for all four. Modules give the boundary; `library(main)` and `initialization/2` make a file into a script; `open/3`, `read_term/2`, `library(readutil)`, `library(csv)`, `library(http/json)`, and `library(pcre)` read what debuggers, tracing tools, and other programs write; `library(debug)` and `library(plunit)` make the result observable and checkable. A last section looks at what SWI-Prolog does with a knowledge base of facts, which is why lookups of `frame/5` by thread and frame number cost the same for a dump of ten threads and a dump of ten thousand.

The diagnostic use is the packaging of what the earlier chapters wrote. The stack trace predicate of Chapters 5 and 7, the **Spiking Thread** (page 335) recognizer of Chapter 6, and the wait relation of Chapter 4 become a module `dumpkb` with a declared interface; a test suite states what they must say about `dump1.pl`, so that a change to a predicate that alters a pattern recognition is caught; and a script reads a software trace from a CSV file and prints its **Time Delta** (page 511) anomalies, the first tool of this book that runs outside the toplevel.

13.1 Modules

A **module** is a named namespace for predicates. Every predicate lives in exactly one module, and a goal is resolved in the module of the clause that contains it. Until now every program block was loaded into the module `user`, the default for files without a module declaration and for queries typed at the toplevel; the built-in predicates live in `system`, from which every other module inherits. The libraries loaded so far, `library(lists)`, `library(apply)`, `library(aggregate)`, are modules too, and the predicates they define reached `user` either by an explicit `use_module/1` or by **autoloading**, the mechanism by which SWI-Prolog loads a library the first time one of its exported predicates is called; the flag `autoload` controls it, and a program meant to be portable or to start quickly names its libraries.

A file becomes a module by beginning with the directive `:- module(Name, Exports).`, where `Exports` is a list of predicate indicators. Predicates in the list are **exported**; the others are local to the module. The following file is the dump knowledge base of this chapter. It loads the shared data into itself and exports five relations over it, two from earlier chapters and three that recognize or prepare patterns:

```
:- module(dumpkb,
          [ stack_trace/2,
            top_frame/2,
            thread_modules/2,
            spiking_thread/1,
            blocked_on/2
          ]).

:- [dump1].

%!  stack_trace(+TID, -Frames) is det.
%
%   Frames is the list of frame/5 terms of thread
%   TID in frame order, [] if TID has no frames.
stack_trace(TID, Frames) :-
    frames_from(TID, 0, Frames).

frames_from(TID, N, Frames) :-
    (   frame(TID, N, M, F, O)
    ->  Frames = [frame(TID, N, M, F, O) | Rest],
        succ(N, N1),
        frames_from(TID, N1, Rest)
    ;   Frames = []
    ).

%!  top_frame(?TID, ?Function) is nondet.
%
%   Function is the function of frame 0 of TID.
top_frame(TID, Function) :-
    frame(TID, 0, _, Function, _).

%!  thread_modules(+TID, -Modules) is det.
%
%   Modules is the set of modules on the stack
%   trace of TID, in standard order.
thread_modules(TID, Modules) :-
    stack_trace(TID, Frames),
    maplist(frame_module, Frames, List),
    sort(List, Modules).

frame_module(frame(_, _, M, _, _), M).

%!  spiking_thread(?TID) is nondet.
%
%   TID used at least a quarter of the CPU time
%   of all threads of the dump.
spiking_thread(TID) :-
    aggregate_all(sum(U + K),
                  thread_time(_, U, K), Total),
    thread_time(TID, User, Kernel),
    (User + Kernel) * 4 >= Total.
```

```
%!  blocked_on(?TID, ?Owner) is nondet.
%
%   TID waits for a critical section that thread
%   Owner holds.
blocked_on(TID, Owner) :-
    waits_for(TID, critical_section, Addr),
    critical_section(Addr, Owner, _, _),
    Owner =\= 0.
```

The directive `:- [dump1].` inside a module file loads a file that has no module declaration of its own, and such a file is loaded **into the module that loads it**: the 165 clauses of `dump1.pl` become predicates of `dumpkb`, not of `user`. SWI-Prolog allows a non-module file to be loaded into one module only; a second module that tried to consult `dump1.pl` would be refused with a permission error, which is the reason the data is placed in the module that owns the predicates over it.

The comments beginning with `%!` are **PlDoc** documentation: a header line with the predicate indicator in mode notation and a determinism annotation (`det`, `semidet`, `nondet`, `multi`), then a blank comment line, then prose. `library(pldoc)` can render them as HTML and `help/1` shows them in the toplevel; their first use is that the header states the contract, the modes under which the predicate is meant to be called and how many times it may succeed, which Chapter 7 defined and which a reader otherwise has to work out from the clauses.

A module file is loaded with `use_module/1`, which consults the file if it is not loaded yet and imports its exports into the module that issued the directive, here `user`:

```
:- use_module(dumpkb).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- stack_trace(109, _Fs), length(_Fs, N).
N = 7.

?- top_frame(109, F).
F = 'Buffer::Copy'.

?- spiking_thread(T).
T = 102 ;
T = 107 ;
false.

?- blocked_on(T, O).
T = 101,
O = 103 ;
T = 103,
O = 104 ;
T = 104,
O = 103.
```

The flag `toplevel_print_anon` is set as in Chapter 5, so that bindings of variables whose names begin with an underscore are not printed. The exported predicates are called from `user` as if they

were defined there. The facts of the dump are not; `frame/5` is a predicate of `dumpkb` and not in the export list, so a goal `frame(109, 0, M, F, O)` in `user` refers to a predicate `user:frame/5` that does not exist:

```
?- G = frame(109, 0, M, F, O),
   catch(G, error(E, _), true).
G = frame(109, 0, M, F, O),
E = existence_error(procedure, frame/5).
```

A goal is resolved in another module by **module qualification**, `Module:Goal`. The qualification reaches every predicate of the module, exported or not; SWI-Prolog's module system is a namespace mechanism, not an access control, and `dumpkb:frame(...)` is the way to inspect the data behind the interface when the interface is not enough:

```
?- dumpkb:frame(109, 0, M, F, O).
M = dbclient,
F = 'Buffer::Copy',
O = 26.

?- dumpkb:frames_from(105, 2, _Fs),
   length(_Fs, N),
   last(_Fs, frame(_, K, M, F, _)).
N = 2,
K = 3,
M = ntdll,
F = 'RtlUserThreadStart'.
```

The toplevel prints the offsets in decimal, as Chapter 2 said it would; the module boundary changes nothing about terms. Two reflection predicates answer questions about modules and their predicates. `module_property/2` gives a module's export list and source file; `predicate_property/2` with `imported_from(M)` says where a predicate visible in the current module came from, and with `exported` whether it is part of an interface:

```
?- module_property(dumpkb, exports(_Es)),
   length(_Es, N).
N = 5.

?- predicate_property(stack_trace(_, _),
   imported_from(M)).
M = dumpkb.

?- predicate_property(append(_, _, _),
   imported_from(M)).
M = lists.

?- predicate_property(dumpkb:frames_from(_, _, _),
   exported).
false.
```

`use_module/2` imports a subset: `:- use_module(dumpkb, [stack_trace/2]).` makes only that predicate visible, and `:- use_module(dumpkb, []).` loads the module without importing anything, for use with qualification only. A local definition of a predicate that is also imported overrides the

import, with a warning at load time that the import was weak; importing the same name from two modules is a permission error, and the first import stays. `reexport/1,2` imports a module and adds its exports to the importing module's own export list, which is how a module that collects the recognizers of several pattern families can present them as one interface:

```
:- module(patterns, []).
:- reexport(dumpkb).
:- reexport(tracekb, [time_delta/3]).
```

The argument of `use_module/1` is a **file specification**: a plain atom is a path relative to the directory of the file being loaded, or to the working directory at the toplevel, with the extension `.pl` supplied; the term `library(Name)` is resolved through the predicate `file_search_path/2`, whose clauses for `library` name the directories SWI-Prolog searches. `absolute_file_name/3` performs the resolution on request:

```
?- absolute_file_name(library(lists), F,
   [file_type(prolog), access(read)]).
F = '/usr/lib/swi-prolog/library/lists.pl'.
```

The path is that of this installation, the directory the flag `home` names; on Windows it is under the SWI-Prolog program directory, and on any other installation the answer differs in that prefix and in nothing else. A program adds its own search path by asserting a clause of `file_search_path/2`, which is a dynamic, multifile predicate, so that `use_module(diagnostics(dumpkb))` finds the module wherever the installation put it.

13.1.1 Meta-Predicates

A predicate that takes a goal or a closure as an argument, as `maplist/3`, `findall/3`, and `forall/2` do, raises a question modules must answer: in which module is the goal resolved? The default answer, the module of the clause containing the call, is wrong for a library predicate called from another module with a closure that names the caller's predicate. The directive `meta_predicate/1` declares which arguments are **module-sensitive**: `0` marks an argument called as a goal, an integer `N` a closure to which `N` arguments are added before the call, `:` an argument that is module-sensitive but not called, and `+`, `-`, `?` ordinary arguments. SWI-Prolog then qualifies a module-sensitive argument with the caller's module before the predicate is entered. The following module shows the effect:

```
:- module(closures,
          [ show_goal/1,
            show_plain/1,
            frames_where/3
          ]).

:- meta_predicate show_goal(0).
:- meta_predicate frames_where(+, 1, -).
```

```
show_goal(G) :- writeln(G).
show_plain(G) :- writeln(G).

%!  frames_where(+TID, :Cond, -Frames) is det.
%
%   Frames are the frames of TID for which the
%   closure Cond succeeds.
frames_where(TID, Cond, Frames) :-
    dumpkb:stack_trace(TID, All),
    include(Cond, All, Frames).

:- use_module(closures).

% in_module(M, Frame): Frame is in module M.
in_module(M, frame(_, _, M, _, _)).

?- show_goal(true).
user:true
true.

?- show_plain(true).
true
true.

?- frames_where(109, in_module(appserver), _Fs),
   maplist([frame(_, N, _, F, _), N-F]>>true,
           _Fs, Pairs).
Pairs = [2-'Session::Fetch', 3-'Worker::Run'].

?- predicate_property(frames_where(_, _, _),
   meta_predicate(Spec)).
Spec = frames_where(+, 1, -).
```

`show_goal/1` received `user:true`, its argument qualified with the module of the call, and `show_plain/1` received the bare atom. In `frames_where/3`, `include/3` receives `user:in_module(appserver)` and resolves it in `user`, where the predicate is defined. Without the declaration the call would succeed here too, because `user` is the **default import module** of every ordinary module: a predicate not found in `closures` is looked up in `user`, then in `system`. The declaration matters when the caller is itself a module other than `user`, which is the normal case for a predicate in a library, and the rule is simple: every argument that is called, in the predicate itself or in anything it calls, is declared. The mode `:` in the PlDoc header is the same information for the reader.

13.2 Programs and Scripts

A Prolog program runs in one of three ways. Consulted into a toplevel, it is queried interactively, which is how every transcript in this book was made. Loaded with `swipl -g Goal -t halt file.pl`, as Chapter 1 showed, it runs `Goal` after loading and exits. The third way is a **script**: a file that declares its own entry point, so that `swipl file.pl args` runs it, and that on Unix-like systems can be made executable with a first line `#!/usr/bin/env swipl`, which SWI-Prolog skips when it loads the file.

The entry point is declared with `initialization/2`. The directive `:- initialization(Goal, When).` registers `Goal` to run at the moment `When` names: `now`, immediately; `after_load`, when the file containing the directive has been loaded, which is also what the one-argument `initialization/1` does; `program`, after the files named on the command line have been loaded and the `-g` goals have run; and `main`, which runs `Goal` after loading, in place of the toplevel, and halts with exit status 0 when the goal succeeds, 1 when it fails, and 2 when it raises an exception, after printing the exception. The same file consulted into a running toplevel registers the goal without running it, since the toplevel is already in charge, which is what makes a script also usable as a module of predicates to query by hand.

`library(main)` supplies the conventional `main/0`: it fetches the command-line arguments that follow the script name, as the flag `argv` holds them, and calls the user's `main/1` with them as a list of atoms, after arranging that an interrupt terminates the program cleanly. `argv_options/3` from the same library separates options from positional arguments: `--threshold=0.7` becomes the term `threshold(0.7)`, with numeric values converted, `--verbose` becomes `verbose(true)`, and the remaining arguments are returned in order. The following script is the one the Diagnostic Use section runs. It reads a software trace from a CSV file whose columns are message number, time, thread, component, and text, and prints every pair of consecutive messages whose **Time Delta** (page 511) reaches a threshold:

```
#!/usr/bin/env swipl
% deltas.pl: print the Time Delta anomalies of a
% software trace stored as CSV.

:- use_module(library(main)).
:- use_module(library(option)).
:- use_module(library(csv)).

% stop the collector thread, so that the exit of
% a short run cannot race it (Section 31.1)
:- set_prolog_gc_thread(false).
:- initialization(main, main).

main(Argv) :-
    argv_options(Argv, [File], Options),
    option(threshold(Min), Options, 1.0),
    csv_read_file(File, Rows, [functor(msg)]),
    report_deltas(Rows, Min).

% report_deltas(Rows, Min): print each pair of
% adjacent rows whose time difference is >= Min.
report_deltas([], _).
report_deltas([Row | Rows], Min) :-
    report_deltas(Rows, Row, Min).
```

```
report_deltas([], _, _).
report_deltas([Row | Rows], Prev, Min) :-
    Prev = msg(N1, T1, _, _, _),
    Row = msg(N2, T2, Th, C, Tx),
    Delta is T2 - T1,
    (   Delta >= Min
    ->  format("~w -> ~w: ~3f s (~w, ~w)~n",
               [N1, N2, Delta, Th, C]),
        format("    ~w~n", [Tx])
    ;   true
    ),
    report_deltas(Rows, Row, Min).
```

`option/3` from `library(option)` looks up `threshold(Min)` in the option list and binds `Min` to the default `1.0` when it is absent. `report_deltas/2` hands the first row to `report_deltas/3`, which carries the previous row alongside the rest; the first argument of each is either `[]` or a non-empty list, so first-argument indexing selects the clause and the predicate leaves no choice point, as Chapter 7 required of a loop. From a shell, with the CSV file that Section 13.4 writes:

```
$ swipl deltas.pl trace1.csv
66 -> 67: 4.691 s (109, appserver)
    Request 10 received: GET /session/9/data
$ swipl deltas.pl --threshold=0.7 trace1.csv
10 -> 11: 0.786 s (100, appserver)
    Request 1 received: GET /report/17
38 -> 39: 0.998 s (100, appserver)
    Request 6 received: GET /health
40 -> 41: 0.999 s (100, appserver)
    Request 7 received: GET /health
43 -> 44: 0.998 s (100, appserver)
    Request 8 received: GET /health
45 -> 46: 0.999 s (100, appserver)
    Request 9 received: GET /health
66 -> 67: 4.691 s (109, appserver)
    Request 10 received: GET /session/9/data
$ echo $?
0
```

With the default threshold the one anomaly is the gap between messages 66 and 67, the **Discontinuity** (page 513) of Chapter 25; at 0.7 seconds the one-second cadence of the health requests shows as well. `halt/1` ends a program with an explicit exit status, and `main/0` arranges the statuses above when `main/1` ends by success, failure, or exception; a script that wants to distinguish kinds of failure calls `halt(Status)` itself. Run without a file argument, `argv_options/3` leaves no positional argument, `[File]` does not unify with `[]`, `main/1` fails, and the exit status is 1; run on a file that does not exist, `csv_read_file/3` raises an existence error, which is printed, and the status is 2.

13.3 Reading and Writing Files

A **stream** is Prolog's handle on an open file or device. `open/3` opens a file in mode `read`, `write`, or `append` and binds a stream term; `open/4` takes options, among them `encoding(utf8)` and

`type(binary)`; `close/1` releases it. The predicates that read and write take the stream as an extra first argument: `read_term/3` is `read_term/2` on a stream, `format/3` is `format/2` on a stream, and `current_output/1` and `current_input/1` give the streams the two-argument versions use.

`read_term/2,3` reads the next term from the input, up to and including the period that ends it, and returns the atom `end_of_file` at the end of the file. It is the predicate `consult/1` is built on, and it is how a program inspects a file of facts without loading it:

```
?- open('dump1.pl', read, _S),
   read_term(_S, T1, []),
   read_term(_S, T2, []),
   close(_S).
T1 = (:-discontiguous frame/5),
T2 = process(4120, 'AppServer.exe').
```

The first term of `dump1.pl` is its directive, read as an ordinary term with principal functor `:-`, which `consult/1` would execute and a program reading the file decides about. A loop over a whole file is a recursion that stops at `end_of_file`, and the file is closed by `setup_call_cleanup/3` of Chapter 7 whether the loop ends normally, fails, or raises:

```
% file_functors(File, Pairs): Pairs lists, in
% standard order, the Name/Arity of each clause
% head in File with its number of clauses.
file_functors(File, Pairs) :-
    setup_call_cleanup(
        open(File, read, S),
        read_functors(S, Fs),
        close(S)),
    msort(Fs, Sorted),
    clumped(Sorted, Pairs).

read_functors(S, Fs) :-
    read_term(S, T, []),
    (   T == end_of_file
    ->  Fs = []
    ;   T = (:- _)
    ->  read_functors(S, Fs)
    ;   functor(T, Name, Arity),
        Fs = [Name/Arity | Rest],
        read_functors(S, Rest)
    ).

?- file_functors('dump1.pl', _Pairs),
   memberchk(frame/5-N, _Pairs),
   length(_Pairs, Kinds).
N = 78,
Kinds = 15.
```

`clumped/2` from `library(lists)` turns a sorted list into `Element-Count` pairs. The predicate is a census of a fact file: 15 kinds of fact, 78 of them frames. `library(readutil)` packages the common whole-file cases. `read_file_to_terms/3` reads every term into a list, `read_file_to_string/3` the whole text into a string, and `read_line_to_string/2` and `read_line_to_codes/2` read one line,

without its line terminator, as a string or a code list, returning `end_of_file` or `-1` respectively at the end. Writing is symmetric. `format/3` writes to a stream; `with_output_to/2` of Chapter 8 redirects the output of a goal into a string or atom; and `portray_clause/2`, Section 13.6, writes a term as a clause a later `consult/1` reads back:

```
?- open('notes.txt', write, _S),
   format(_S, "thread ~w~n", [109]),
   format(_S, "code ~16r~n", [0xc0000005]),
   close(_S).
true.

?- open('notes.txt', read, _S),
   read_line_to_string(_S, A),
   read_line_to_string(_S, B),
   read_line_to_string(_S, C),
   close(_S).
A = "thread 109",
B = "code c0000005",
C = end_of_file.

?- read_file_to_string('notes.txt', Str, []).
Str = "thread 109\ncode c0000005\n".
```

The line-by-line read is the shape of every log reader: open, read a line, parse it with the string predicates of Chapter 8 or the grammars of Chapter 12, recurse, stop at `end_of_file`. For a file too large to hold as a list of lines, `phrase_from_file/2` of Chapter 12 parses it lazily; Chapter 30 returns to the trade-off.

13.4 CSV, JSON, and Regular Expressions

Trace and log analysis tools export tables, and the lowest common format is CSV. `library(csv)` reads and writes it. `csv_write_file/2` takes a list of `row/N` terms, one per line, quoting fields that contain separators, and `csv_read_file/2` reads a file into such a list, converting fields that look like numbers into numbers and the rest into atoms. The shared trace is loaded and written out as a five-column file, with the constant process identifier dropped:

```
:- [trace1].

?- findall(row(N, T, Th, C, Tx),
   msg(N, T, _, Th, C, Tx), _Rows),
   csv_write_file('trace1.csv', _Rows).
true.
```

```
?- read_file_to_string('trace1.csv', _Str, []),
   split_string(_Str, "\n", "\r", _Lines),
   nth1(7, _Lines, _L7),
   nth1(14, _Lines, _L14),
   format("~s~n~s~n", [_L7, _L14]).
7,0.215,101,appserver,Worker 101 ready
14,1.048,101,dbclient,"Query end: 1 row, 44 ms"
true.

?- csv_read_file('trace1.csv', _Rows,
   [functor(msg)]),
   length(_Rows, N),
   nth1(14, _Rows, msg(_, T, Th, C, Text)).
N = 71,
T = 1.048,
Th = 101,
C = dbclient,
Text = 'Query end: 1 row, 44 ms'.
```

The fourteenth message text was quoted on output because it contains a comma; the seventh was not, and the line terminator is the carriage return and line feed pair that the CSV specification prescribes, which `split_string/4` strips here as padding. The file is read back with `csv_read_file/3`, whose `functor(msg)` option names the principal functor of the row terms; `arity(N)` checks the column count, `separator(0'\t)` reads tab-separated files, `skip_header(0'#)` skips comment lines, and `convert(false)` leaves every field an atom. The text comes back as an atom, not as the string it was written from; CSV has no notion of type, and a program that wants strings converts with `atom_string/2` after reading or defines its message terms with atoms throughout. Chapter 16 adopts this file as the second way to load a trace, next to the log parser of Chapter 12.

JSON is the format of structured tool output, and `library(http/json)` maps it onto SWI-Prolog dicts, Chapter 10: an object becomes a dict, an array a list, strings become strings, numbers numbers, and `true`, `false`, `null` the atoms of the same name. `json_write_dict/2,3` writes a dict to a stream and `json_read_dict/2,3` reads one; `atom_json_dict/3` converts in either direction between text and a dict. A record of the exception in `dump1.pl` is written to a file and read back:

```
:- use_module(library(http/json)).

?- _D = _{thread:109, code:0xc0000005,
          access:"read"},
   open('exception.json', write, _S),
   json_write_dict(_S, _D),
   close(_S),
   read_file_to_string('exception.json', _T, []),
   format("~s~n", [_T]).
{"access":"read", "code":3221225477, "thread":109}
true.
```

```
?- open('exception.json', read, _S),
   json_read_dict(_S, D),
   close(_S),
   get_dict(code, D, _Code),
   format("~16r~n", [_Code]).
c0000005
D = _{access:"read", code:3221225477, thread:109}.

?- atom_json_dict('{"tid": 101, "waits": [103]}',
   D, []).
D = _{tid:101, waits:[103]}.
```

Dict keys are written in standard order, and the hexadecimal literals are written as the decimal integers they are. The round trip loses nothing that JSON can carry; what it cannot carry is a compound term, so a frame is exported as an object with named fields, not as `frame(...)`, and rebuilt on reading.

Message texts are the part of a trace that is neither a number nor a field, and matching them by shape is what regular expressions do. `library(pcre)` binds the PCRE library; it loads in this installation, and whether it is present in another depends on how that SWI-Prolog was built, so a program that needs it should say so in its documentation. `re_match/2` succeeds when a pattern matches somewhere in a string; `re_matchsub/3` additionally returns the captured groups as a dict, under their numbers or their names when the pattern uses `(?<name>...)`; a name suffix `_I` converts the capture to an integer, `_A` to an atom, `_N` to a number, and the default is a string. `re_foldl/6` folds a goal over every match in a string:

```
:- use_module(library(pcre)).

% completion(Text, Request, Ms): Text reports the
% completion of Request after Ms milliseconds.
completion(Text, Request, Ms) :-
    re_matchsub("^Request (?<r_I>\\d+) \c
                 completed: (?<s_I>\\d+), \c
                 (?<ms_I>\\d+) ms$",
                Text, Sub, []),
    get_dict(r, Sub, Request),
    get_dict(ms, Sub, Ms).

?- re_match("^Error:", "Error: queue depth 3").
true.

?- re_match("^Error:", "No error here").
false.

?- msg(N, _, _, _, _, Tx), completion(Tx, R, Ms).
N = 17,
Tx = "Request 1 completed: 200, 309 ms",
R = 1,
Ms = 309 ;
N = 58,
Tx = "Request 5 completed: 200, 4300 ms",
R = 5,
Ms = 4300 ;
```

```
N = 60,
Tx = "Request 6 completed: 200, 3302 ms",
R = 6,
Ms = 3302 ;
N = 62,
Tx = "Request 7 completed: 200, 2305 ms",
R = 7,
Ms = 2305 ;
N = 64,
Tx = "Request 8 completed: 200, 1307 ms",
R = 8,
Ms = 1307 ;
N = 66,
Tx = "Request 9 completed: 200, 309 ms",
R = 9,
Ms = 309 ;
false.

?- re_foldl([_, C0, C]>>(C is C0 + 1), "\\d+",
   "Request 5 completed: 200, 4300 ms", 0, N, []).
N = 3.
```

The pattern is a Prolog string, so the backslash of `\d` is doubled, and the string is continued across three source lines with `\c`, the escape that makes the reader skip the line break and the indentation that follows it. `re_foldl/6` calls its closure with the match dict, here ignored, and an accumulator, counting three numbers in the text. The completion times of requests 5 to 9 fall by about a second each, which Chapter 25 treats as **Delay Dynamics** (page 522); the predicate that extracts them is three lines, and the same three lines written with `sub_string/5` would be ten. Regular expressions are the right tool for a message whose shape is fixed and a poor one for a message whose shape has nested structure, which is where the grammars of Chapter 12 take over.

13.5 Debugging, Assertions, and Unit Tests

The tracer of Chapter 4 shows everything, which is too much for a predicate that is almost right. `library(debug)` offers print statements that cost nothing when switched off. `debug/3` takes a **topic**, an atom or compound term, a format string, and arguments, and prints the message, prefixed with `%`, when the topic is enabled; `debug/1` enables a topic, `nodebug/1` disables it, `debugging/1` tests it, and `list_debug_topics/0` lists them. A disabled `debug/3` goal costs one lookup of its topic, so the statements may stay in the program:

```
:- use_module(library(debug)).

% deep_thread(TID, Depth): the stack trace of TID
% has Depth frames, more than nine.
deep_thread(TID, Depth) :-
    stack_trace(TID, Frames),
    length(Frames, Depth),
    debug(depth, "thread ~w: ~w frames",
          [TID, Depth]),
    Depth > 9.
```

```
?- deep_thread(107, D).
D = 14.

?- debug(depth).
true.

?- deep_thread(107, D).
% thread 107: 14 frames
D = 14.

?- deep_thread(105, D).
% thread 105: 4 frames
false.

?- nodebug(depth).
true.
```

The second run of the failing query shows the value that the test rejected, which is the question a print statement is for. Debug messages go to the standard error stream, so they do not mix with a program's output when that output is redirected to a file.

`assertion/1`, from the same library, states a condition the author believes holds at that point of the program. When the goal succeeds the predicate succeeds, deterministically and without bindings; when it fails, SWI-Prolog prints the goal with a backtrace and, in an interactive session, enters the debugger, or, in a non-interactive one, raises `assertion_error`. An assertion is not an input check, which is `must_be/2` of Chapter 7 and raises a typed error the caller can handle; it is a claim about the program's own consistency, and its failure means the program, not its input, is wrong:

```
?- stack_trace(109, _Fs), length(_Fs, N),
   assertion(N > 0).
N = 7.
```

Run non-interactively, a failing assertion prints the goal and its context, abridged:

```
ERROR: Assertion failed: user:(0>0)
  [13] prolog_stack:backtrace(10) at ...
  [12] prolog_debug:assertion_failed(fail,user:(0>0)) at ...
  ...
```

Tests are assertions kept outside the program. `library(plunit)` defines a **unit test** as a clause of `test/1` or `test/2` between the directives `:- begin_tests(Unit).` and `:- end_tests(Unit).`; the clauses are compiled into a module of their own, whose name is derived from the unit's, and which imports from the module in which the directives appear. A test passes when its body succeeds once, and the second argument of `test/2` refines this: `true(Cond)` additionally calls `Cond` after the body, `all(X == List)` collects every solution for `X` and compares the list, `fail` requires the body to fail, `throws(E)` requires it to raise an exception unifying with `E`, and `nondet` records that the body is allowed to leave a choice point, which is otherwise reported as a warning, because an unintended choice point is usually a missing cut or a non-steadfast clause. The unit below tests the two older exports of `dumpkb` against what the dump is known to contain:

```
:- use_module(library(plunit)).
:- set_prolog_flag(verbose, normal).

:- begin_tests(stacks).

test(faulting_frame, true(F == 'Buffer::Copy')) :-
    top_frame(109, F).

test(depth_of_107, true(N == 14)) :-
    stack_trace(107, Frames),
    length(Frames, N).

test(no_such_thread, true(Frames == [])) :-
    stack_trace(999, Frames).

test(pool_threads, all(T == [105, 106])) :-
    top_frame(T, 'NtWaitForWorkViaWorkerFactory').

test(deep_thread_fails, fail) :-
    deep_thread(105, _).

test(top_frame_is_nondet, nondet) :-
    top_frame(_, 'NtWaitForAlertByThreadId').

:- end_tests(stacks).

?- run_tests.
% PL-Unit: stacks ...... passed <...> sec
% All 6 tests passed
true.

?- run_tests(stacks:depth_of_107).
% PL-Unit: stacks:depth_of_107 . passed <...> sec
% test passed
true.
```

The flag `verbose` is set to `normal` because the toplevel of these transcripts was started with `swipl -q`, which sets it to `silent` and suppresses the informational messages that `run_tests/0` prints; a reader starting `swipl` without `-q` sees the summary without the directive. `run_tests/0` runs every unit, `run_tests/1` a unit or a single test, and the time at the end of the progress line varies from run to run. The last test has three solutions, the three threads blocked in the wait function, and the `nondet` option says that this is intended. A test written to find a top frame `RtlUserThreadStart`, which is the bottom frame of every thread, fails, and its report appears below with the reports of two more tests written to fail:

```
:- begin_tests(broken).

test(wrong_depth, true(N == 7)) :-
    stack_trace(107, Frames),
    length(Frames, N).

test(wrong_type) :-
    _ is foo + 1.
```

```
test(bottom_frame, nondet) :-
    top_frame(_, 'RtlUserThreadStart').

:- end_tests(broken).

?- run_tests(broken).
% PL-Unit: broken
ERROR: <...>
<...>test wrong_depth: wrong answer <...>
ERROR:     Expected: 7
ERROR:     Got:      14
ERROR: <...>
<...>test wrong_type: received error: <...>
ERROR: <...>
<...>test bottom_frame: failed
...
% 3 tests failed
% 0 tests passed
false.
```

Each failure names the test, its file and line, and the kind of failure: a wrong answer with both values, an error the body raised, here the type error of evaluating the atom `foo`, and a plain failure. `run_tests/1` fails when any test failed, which is what a build script needs. The suite of Section 13.7 tests the pattern recognizers the same way, and Chapter 17 makes such suites part of the definition of a recognizer: the tests say which instances a predicate must find in a known artifact and which it must not.

Two more checks run without tests. `check/0` examines the loaded program for predicates that are called but nowhere defined, for goals that cannot succeed because no clause of the called predicate matches them, for `format/2` templates whose directives disagree with their argument lists, and for redefinitions of system predicates; `list_undefined/0` is the first of these alone. Both print warnings naming the clause and the source location. A misspelled `lenght/2` in a clause body is found by `check/0` at load time rather than by an existence error at the one run that reaches the clause, which, in a pattern recognizer run over a dump that happens not to contain the pattern, may be never.

13.6 Timing, Indexing, and Inspection

`time/1` runs a goal and prints the number of **inferences**, the calls of predicates it made, with CPU and wall-clock times, and `statistics/2` gives the same counters as values: `inferences`, `cputime` and `process_cputime` in seconds, `walltime` in milliseconds, and the memory keys of Chapter 30. The transcripts below start a fresh toplevel with the module only, so that the indexes shown reflect only the calls made here:

```
:- use_module(dumpkb).

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

```
?- time(stack_trace(107, _Fs)).
% <...> inferences, <...> CPU in <...>
true.

?- statistics(inferences, _I0),
   stack_trace(107, _Fs),
   statistics(inferences, _I1),
   I is _I1 - _I0.
I = 47.
```

Forty-seven inferences for a stack trace of fourteen frames: three per frame for the lookup, the successor, and the recursive call, and a few for the ends. The figure of interest is not the count but its independence from the size of the dump, and that is a matter of **indexing**. Chapter 7 said that SWI-Prolog selects clauses by their first argument when it is bound at the call, so that `thread(109, P)` does not try ten clauses but one. The full mechanism is **just-in-time indexing**: on the first call of a predicate with a given instantiation pattern, SWI-Prolog decides whether a hash index on some argument, or on a combination of arguments, would make the call selective, builds the index if so, and keeps it for later calls with the same pattern. `jiti_list/1` shows the indexes a predicate has acquired:

```
?- jiti_list(dumpkb:frame/5).
Predicate <...> Flags
<...>
dumpkb:frame/5 <...>
true.
```

The one index so far is on the pair of the first two arguments, the pattern `frame(TID, N, _, _, _)` with which `stack_trace/2` has been calling the predicate, since neither the thread alone, with up to fourteen frames each, nor the frame number alone, with up to ten threads each, is as selective as the two together. A call with a different pattern adds a second index:

```
?- dumpkb:frame(T, N, _, 'Worker::Run', _).
T = 101,
N = 6 ;
T = 102,
N = 3 ;
T = 103,
N = 6 ;
T = 107,
N = 10 ;
T = 109,
N = 3.

?- jiti_list(dumpkb:frame/5).
Predicate <...> Flags
<...>
dumpkb:frame/5 <...>
<...>4 <...>
true.
```

The new line is an index on argument 4, the function name, and from now on the question “which threads have `Worker::Run` on their stack” costs a hash lookup and five clause tries, not seventy-

eight. The columns of the table give the number of hash buckets and an estimated speedup against a linear scan; the figures depend on the data and are left out above. Two consequences for the design of a knowledge base follow. The argument most often bound at calls should come first, since the first argument is indexed without a decision, which is why the schema puts the thread identifier first in `frame/5` and the message number first in `msg/6`; and an index is built per instantiation pattern, so a predicate called in many ways acquires several indexes, each costing memory proportional to the number of clauses, which for a trace of a million messages is a consideration that Chapter 30 takes up.

`listing/1` prints the clauses of a predicate as SWI-Prolog has stored them, in the layout of this book; `portray_clause/1` prints one term the same way, and `portray_clause/2` prints it to a stream, which is the way to write a knowledge base to a file that `consult/1` reads back:

```
?- listing(dumpkb:top_frame).
top_frame(TID, Function) :-
    frame(TID, 0, _, Function, _).

true.

?- portray_clause(frame(109, 0, dbclient,
   'Buffer::Copy', 0x1a)).
frame(109, 0, dbclient, 'Buffer::Copy', 26).
true.
```

The variable names of the source are shown because SWI-Prolog records them when it loads a file in this edition; a term printed with `portray_clause/1` has no source, and its variables are named `A`, `B`, and so on. A dump written out with `portray_clause/2` has its addresses in decimal, which `consult/1` reads back without loss; a reader who wants hexadecimal in the generated file uses `format/3` with `~16r` instead, at the cost of writing the clause layout by hand.

An exception that escapes to the toplevel is printed with a backtrace, as the error transcripts of Chapter 7 showed, but an exception caught by `catch/3` carries only the error term, and by the time the handler runs the stack that raised it is gone. `catch_with_backtrace/3` is `catch/3` with the backtrace recorded in the context argument of the error term, from where `print_message/2` prints it; `library(prolog_stack)` is the library behind both, and its `backtrace/1` prints the current stack on demand, the goal to put inside a handler when the question is not what went wrong but how the program got there. A script's `main/1` that catches everything and prints the error term alone throws the answer away; the script of Section 13.2 catches nothing and lets `main/0` print the exception with its context.

13.7 Diagnostic Use

The module `dumpkb` of Section 13.1 is the dump knowledge base this chapter set out to build: the facts of `dump1.pl` behind an interface of five relations, three of them the recognizers of two patterns and the preparation of a third. `stack_trace/2` builds the **Stack Trace** (page 329) of a thread as Chapter 5 defined it. `spiking_thread/1` recognizes **Spiking Thread** (page 335), a thread whose share

of the process's CPU time is out of proportion to the rest, with the share threshold of one quarter that Chapter 6 used, written here in integer arithmetic; and `blocked_on/2` relates a thread to the owner of the critical section it waits for, which is Chapter 4's `blocked_by/2`, the one link of a **Wait Chain (Critical Sections)** (page 384) that Chapter 11 closes transitively and Chapter 20 treats in full. The test suite states what these must say about `dump1.pl`:

```
:- use_module(library(plunit)).

:- begin_tests(patterns).

test(spiking_threads, all(T == [102, 107])) :-
    spiking_thread(T).

test(main_thread_not_spiking, fail) :-
    spiking_thread(100).

test(deadlock_pair, true(O == 104)) :-
    blocked_on(103, O).

test(blocked_threads, all(T-O == [101-103,
                                  103-104,
                                  104-103])) :-
    blocked_on(T, O).

test(faulting_modules, true(Ms == [appserver,
                                   dbclient,
                                   kernel32,
                                   ntdll,
                                   ucrtbase])) :-
    thread_modules(109, Ms).

test(idle_pool_thread, true(Ms == [kernel32,
                                   ntdll])) :-
    thread_modules(105, Ms).

:- end_tests(patterns).

?- set_prolog_flag(verbose, normal).
true.

?- run_tests(patterns).
% PL-Unit: patterns ...... passed <...> sec
% All 6 tests passed
true.
```

The tests are the catalog's descriptions made operational for one artifact: there are exactly two spiking threads and the main thread is not one; thread 103 is blocked by 104; the three blocked threads and their owners are a list whose last two elements, `103-104` and `104-103`, are the **Deadlock (Critical Sections)** (page 388) of Chapter 11 in miniature. A later change to `spiking_thread/1`, say a lower threshold that admits thread 100, fails the first two tests, and the failure report names the thread the recognizer now wrongly admits. The tests also fix the interface: `thread_modules/2` returns the modules in standard order of atoms, and a change to `msort/2`, which would keep duplicates, or to the order, breaks the fifth test.

The second artifact is the software trace, read by the script of Section 13.2 from the CSV file of Section 13.4. **Time Delta** (page 511) is the difference between the timestamps of two messages, usually adjacent ones, and the pattern is the reading of such differences as evidence: a delta much larger than its neighbors marks a wait, a stall, or an interval in which the traced component was not writing, and a delta of the same length repeated is a timer. The script computes the delta of every adjacent pair and reports those at or above a threshold. Its logic can be exercised in the toplevel, which is the use of a script that is also a loadable program:

```
:- consult(deltas).

?- csv_read_file('trace1.csv', _Rows,
   [functor(msg)]),
   report_deltas(_Rows, 2.0).
66 -> 67: 4.691 s (109, appserver)
    Request 10 received: GET /session/9/data
true.
```

The one delta above two seconds is the 4.691 seconds between the completion of request 9 at 9.309 and the arrival of request 10 at 14.000, during which nothing was written: a **Discontinuity** (page 513), Chapter 25, and the interval in which the deadlocked workers, the spiking thread, and the suspended housekeeping thread of the dump were all silent. The output of the script from the shell, shown in Section 13.2, is the same text; the script, the CSV reader, the module, and the tests are four files, and the book’s remaining chapters assume this way of packaging: a module per knowledge base or pattern family, a test unit per module, and a script per analysis that is run more than once.

Chapter 14. Constraints and Coroutining

Every goal met so far either succeeded, failed, or raised an error at the moment it was called, with whatever bindings existed then. A test such as `X \== Y` on two unbound variables is true at once and stays true in the search even after the variables are bound to the same value; a comparison such as `T2 - T1 > 500` on unbound timestamps raises an instantiation error. The order of goals in a clause body therefore carries meaning that the declarative reading does not express, and Chapter 7 gave the rules for keeping tests after the goals that bind their variables. This chapter introduces goals that do not have to be placed: a **constraint** is a goal that, called with unbound variables, suspends and becomes part of the state, and that is woken whenever a variable it mentions is bound, when it either fails, prunes the possibilities of the remaining variables, or succeeds. `dif/2`, `freeze/2`, and `when/2` are the simplest constraints; `library(clpfd)`, constraint logic programming over finite domains of integers, is a complete arithmetic in this style, in which `X #> 5` on an unbound `X` is a statement about `X`, not an error.

The diagnostic use is reasoning from partial knowledge, which is most of diagnostics. A software trace rarely records every event, and the ones it records are stamped by clocks that disagree; a dump shows where threads are and not how they got there. Constraints state what is known, the order a program imposes on its own events, the increments a counter can take, the window a clock reading could belong to, and let propagation deduce what follows: the orders a trace could have had (**Event Sequence Order** (page 603)), whether an observed order could have happened at all (**Impossible Trace** (page 614)), the values an unlogged counter must have passed through (**Counter Value** (page 568)), the interval in which two threads became deadlocked, and the alignment of a trace split across two clocks (Split Trace). A constraint solver deduces exactly what follows from the constraints and no more, and when nothing follows it says so by leaving a variable with a domain; that honesty about what is undetermined is the reason to use one.

14.1 Coroutining

The three predicates of this section belong to **coroutining**: the suspension of a goal until a variable it depends on is bound. The shared data is loaded as in the earlier chapters, with the toplevel flag of Chapter 5 set.

```
:- [dump1].
:- [trace1].

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

`dif/2` states that two terms are different. If they can never unify it succeeds; if they are identical it fails; and if they could become identical under some binding it suspends and is re-examined whenever one of the variables involved is bound. The toplevel prints a suspended constraint after the bindings, as a **residual goal**, the part of the answer that is not a binding:

```
?- dif(X, a).
dif(X, a).

?- dif(X, Y), X = a.
X = a,
dif(a, Y).

?- dif(X, Y), X = a, Y = a.
false.

?- dif(f(X, b), f(a, Y)).
dif(f(X, Y), f(a, b)).
```

The last answer shows the solver's reformulation: the two compound terms differ exactly when `X` and `Y` differ from `a` and `b` as a pair, and that is what remains. Compare `\==/2` of Chapter 3, which tests the two terms at the moment of the call: `X \== Y, X = a, Y = a` succeeds, because the test was made while the variables were distinct, and the later bindings are not re-examined. The difference is the one Chapter 7 had to legislate for with goal order: a `dif/2` goal may be written first, before the goals that bind its variables, and is correct wherever it stands. The question "two threads with the same top frame" is the case. With `dif/2` in front, the count of ordered pairs is eight; with `\==` in front, the ten pairs of a thread with itself pass the test, made while both variables were unbound, and are counted too:

```
?- aggregate_all(count, ( dif(T1, T2),
     frame(T1, 0, M, F, _),
     frame(T2, 0, M, F, _) ), N).
N = 8.

?- aggregate_all(count, ( T1 \== T2,
     frame(T1, 0, M, F, _),
     frame(T2, 0, M, F, _) ), N).
N = 18.

?- dif(T1, T2),
   frame(T1, 0, ntdll, F, _),
   frame(T2, 0, ntdll, F, _),
   F == 'NtWaitForWorkViaWorkerFactory'.
T1 = 105,
T2 = 106,
F = 'NtWaitForWorkViaWorkerFactory' ;
T1 = 106,
T2 = 105,
F = 'NtWaitForWorkViaWorkerFactory' ;
false.
```

The `dif/2` goal is woken when `T2` is bound and fails the pair at once when it repeats `T1`; the eight pairs are the six orderings of the three threads blocked in `NtWaitForAlertByThreadId` and the two of the idle pool threads. The `==/2` test at the end of the third query is still placed after the goals that bind `F`, since it is not a constraint.

`freeze/2` attaches a goal to a variable: `freeze(X, Goal)` runs `Goal` as soon as `X` is bound to a non-variable term, and at once if it already is. `frozen/2` reports the goals attached to a variable, qualified with their module:

```
?- freeze(X, writeln(X)), X = 5.
5
X = 5.

?- freeze(X, writeln(X)).
freeze(X, writeln(X)).

?- freeze(X, Y is X * 2), frozen(X, G).
G = freeze(X, user:(Y is X*2)),
freeze(X, Y is X*2).

?- frozen(X, G).
G = true.
```

`when/2` generalizes the trigger: `when(Cond, Goal)` runs `Goal` when `Cond` becomes true, where `Cond` is built from `nonvar(X)`, `ground(T)`, `?=(X, Y)`, which is true when `X` and `Y` are either identical or cannot unify, and conjunctions and disjunctions of these. A delta that is computed as soon as both timestamps are known, in whichever order they are found, is the diagnostic shape:

```
?- when(ground(T1 - T2), D is T2 - T1), T1 = 1.0.
T1 = 1.0,
when(ground(1.0-T2), D is T2-1.0).

?- when(ground(T1 - T2), D is T2 - T1),
   T2 = 1.5, T1 = 1.0.
T1 = 1.0,
T2 = 1.5,
D = 0.5.
```

All three predicates are built on **attributed variables**, variables that carry a module-keyed attribute and a hook, `attr_unify_hook/2`, that SWI-Prolog calls when the variable is unified; `put_attr/3` and `get_attr/3` manipulate the attributes directly, and the constraint libraries, including the one of the next section, are written with them. A program that needs a constraint the libraries do not have writes its own hook; this book does not.

14.2 Constraints over Integers

`library(clpfd)` is **constraint logic programming over finite domains**: arithmetic over the integers in which the relations are constraints. It is loaded explicitly; its operators are not part of the base language.

```
:- use_module(library(clpfd)).
```

The equality is `#=/2`. Where `is/2` evaluates its right side and needs it ground, `#=` relates two expressions and solves for whichever side has the unknown, or suspends when it cannot:

```
?- X #= 3 + 4.
X = 7.

?- 7 #= X + 4.
X = 3.

?- X + Y #= 10.
X+Y#=10.

?- X #= Y * Y, X #= 16.
X = 16,
Y in -4\/4.
```

The last two answers are residual goals in the syntax of the library: an equation that cannot be solved yet, and a **domain**, the set of integers a variable may still take, written with `..` for an interval and `\/` for a union. `in/2` gives a variable a domain and `ins/2` gives one to every variable of a list; `inf` and `sup` are the unbounded ends. The comparison constraints are `#</2`, `#>/2`, `#=</2`, `#>=/2`, and `#\=/2`. Posting a constraint narrows the domains of the variables it mentions, and this narrowing is **propagation**: the solver removes from each domain the values that no solution of the constraint can take, as far as it can see, and does so again whenever a domain changes:

```
?- X in 1..10, X #> 5, X #< 8.
X in 6..7.

?- X #\= 3, X in 1..5.
X in 1..2\/4..5.

?- [X, Y] ins 0..5, X + Y #= 9.
X in 4..5,
X+Y#=9,
Y in 4..5.

?- X in 1..1000000, X * X #= 144.
X = 12.
```

The third answer shows the limit of propagation. From `X + Y #= 9` with both in `0..5` the solver deduces that each is at least 4, and stops: the two remaining values of each are consistent with some value of the other, and the constraint stays as a residual goal, because propagation is **bounds-consistent** for arithmetic, reasoning about the ends of intervals and not about each pair of values. Propagation is sound, in that it never removes a value that belongs to a solution, but not complete, in that an answer with residual goals is not a guarantee that a solution exists:

```
?- X #= Y + 1, Y #= Z + 1, Z #= X + 1.
Z+1#=Y,
Y+1#=X,
X+1#=Z.
```

The three equations have no integer solution, and without domains the solver has nothing to narrow and reports the constraints as given. The complement of propagation is **search**. `label/1` takes a list of variables with finite domains and enumerates their values systematically, posting each choice and propagating, backtracking when a domain becomes empty; every answer it produces is a solution,

and when it fails there is none. `labeling/2` takes options: `ff`, first-fail, labels the variable with the smallest domain first; `min(Expr)` and `max(Expr)` order the solutions by the value of an expression, so that the first solution is an optimum:

```
?- X in 1..10, X #> 5, X #< 8, label([X]).
X = 6 ;
X = 7.

?- [X, Y] ins 0..5, X + Y #= 9, label([X, Y]).
X = 4,
Y = 5 ;
X = 5,
Y = 4.

?- [X, Y] ins 1..3, X #< Y,
   labeling([max(Y - X)], [X, Y]).
X = 1,
Y = 3 ;
X = 1,
Y = 2 ;
X = 2,
Y = 3 ;
false.
```

The last answer is a point of method. Labeling with `max(Y - X)` returns every solution in decreasing order of the objective, and the first answer is the maximum; a program that wants only the optimum wraps the goal in `once/1`. The combination of propagation and labeling is the way every model of this chapter is used: post the constraints, read the domains to see what follows without search, and label when a concrete scenario is wanted or when the question is whether any exists.

The reflection predicates `fd_dom/2`, `fd_inf/2`, `fd_sup/2`, and `fd_size/2` read a variable's domain, its bounds, and its size, for a program that acts on what propagation found rather than printing it:

```
?- X in 1..10, X #> 5,
   fd_dom(X, D), fd_size(X, N).
D = 6..10,
N = 5,
X in 6..10.
```

Integers in CLP(FD) are the unbounded integers of SWI-Prolog, so a domain may be `inf..sup`, but labeling requires finite domains, and a model whose variables are times in milliseconds gives them bounds from the artifact, the first and last timestamps, before it labels. Constraints over the rational or real numbers are `library(clpq)` and `library(clpr)`, which this book does not use; a time in seconds is converted to an integer number of milliseconds before it enters a model.

14.3 Global Constraints and Reification

A **global constraint** relates many variables at once and propagates more than the conjunction of pairwise constraints it stands for. `all_different/1` and `all_distinct/1` both state that the

elements of a list are pairwise distinct; the first propagates only when a variable is bound, the second reasons about the domains as a whole, at a higher cost per propagation:

```
?- X in 1..2, Y in 1..2, Z in 1..3,
   all_different([X, Y, Z]).
X in 1..2,
all_different([X, Y, Z]),
Y in 1..2,
Z in 1..3.

?- X in 1..2, Y in 1..2, Z in 1..3,
   all_distinct([X, Y, Z]).
Z = 3,
X in 1..2,
all_distinct([X, Y, 3]),
Y in 1..2.
```

`all_distinct/1` sees that `X` and `Y` between them use up both values of `1..2` and binds `Z`; `all_different/1` waits. For models of the size in this chapter, a few dozen variables, `all_distinct/1` is the one to use. `sum/3` constrains the sum of a list, `sum(List, #=, 6)`; `tuples_in/2` constrains a list of variable tuples to take their values from a table of allowed tuples, which is how a finite relation given as data, a table of permitted state transitions for example, enters a model; and `zcompare/3` relates an order `<`, `=`, `>` to two integers as a constraint, the counterpart of `compare/3` of Chapter 2. In the second query below the table lists the permitted transitions between three states numbered 0, 1, 2, and two consecutive transitions are constrained to it; fixing the first state determines the second and leaves two possibilities for the third:

```
?- [A, B, C] ins 1..3, sum([A, B, C], #=, 6),
   all_distinct([A, B, C]), A #< B, B #< C.
A = 1,
B = 2,
C = 3.

?- tuples_in([[S0, S1], [S1, S2]],
   [[0, 1], [1, 0], [1, 2]]), S0 = 0.
S0 = 0,
S1 = 1,
S2 in 0\/2.
```

Reification turns the truth of a constraint into a 0/1 integer that other constraints can use. `B #<==> C` states that `B` is 1 if and only if the constraint `C` holds, `#==>/2` is implication, and `#\/`, `#/\`, `#\` are the connectives; the reified constraints propagate in both directions:

```
?- B #<==> (X #> 5), X = 7.
B = 1,
X = 7.

?- B #<==> (X #> 5), X in 1..4.
B = 0,
X in 1..4.
```

```
?- X in 0..10, B #<==> (X #> 5), B = 1.
B = 1,
X in 6..10.

?- X #> 5 #==> Y #= 1, X = 6.
X = 6,
Y = 1.
```

Reification is how a model counts: the number of variables of a list that satisfy a condition is the sum of their reified conditions, and a constraint on that sum constrains the list. The following predicate states that at most `Max` of the values of a list exceed a threshold, which is the shape of a statement like “at most two workers were busy at any time”:

```
% at_most(Max, Threshold, Values): at most Max of
% Values exceed Threshold.
at_most(Max, Threshold, Values) :-
    maplist(exceeds_flag(Threshold), Values,
            Flags),
    sum(Flags, #=<, Max).

exceeds_flag(Threshold, Value, Flag) :-
    Flag #<==> (Value #> Threshold).


?- Vs = [A, B, C], Vs ins 0..9, at_most(1, 5, Vs),
   A = 7.
Vs = [7, B, C],
A = 7,
B in 0..5,
C in 0..5.
```

Once `A` exceeds the threshold the budget is spent, and the solver bounds the other two without being asked. A reified constraint whose truth cannot yet be decided stays as a residual goal with `B in 0..1`; the solver never guesses.

The library also provides `library(clpb)`, constraints over Boolean variables with a complete decision procedure, which is the tool for questions that are purely about presence and absence, such as the Boolean tables of Chapter 28; this chapter stays with integers, since the diagnostic questions below are about times and counts.

14.4 Reconstructing an Event Order

A software trace is read as a sequence, and the catalog’s **Event Sequence Order** (page 603) pattern is the attention paid to that sequence: which message came before which, and whether the recorded order is the order in which the events happened, since timestamps, buffering, and several writers can put messages in an order other than the one the program went through. The constraint formulation starts from what the program imposes on its own events. A report request in the application of `trace1.pl` goes through seven: it is received, dispatched to a worker, the worker starts and ends a database query, starts and ends building the report, and the request completes. Some of the order is forced by the program, dispatch after receipt, each end after its start, both ends

before completion, and the following predicate states exactly that and nothing more, as constraints on seven time variables:

```
% request_order(Events): Events pairs the seven
% lifecycle events of a report request with their
% times; the constraints are those the program
% imposes on every request.
request_order([ received-R, dispatched-D,
                query_start-QS, query_end-QE,
                build_start-BS, build_end-BE,
                completed-C ]) :-
    R #< D,
    D #< QS, QS #< QE,
    D #< BS, BS #< BE,
    QE #< C, BE #< C.
```

The times may be positions, 1 to 7, as well as milliseconds; the constraints do not care. With positions, all distinct, propagation alone fixes three of the seven and bounds the rest:

```
?- request_order(_Es), pairs_values(_Es, Ps),
   Ps ins 1..7, all_distinct(Ps).
Ps = [1, 2, _A, _B, _C, _D, 7],
_A in 3..5,
_A#=<_B+ -1,
all_distinct([1, 2, _A, _B, _C, _D, 7]),
_B in 4..6,
_C in 3..5,
_C#=<_D+ -1,
_D in 4..6.
```

The receipt is first, the dispatch second, the completion last; the query and the build are each an ordered pair somewhere in positions 3 to 6, and the solver writes `_A #=< _B + -1` for the strict inequality. Which orders remain is a question for labeling, and the number of them is the number of sequences the trace could show for one request if the program were exactly this model:

```
% consistent_orders(Count): Count is the number of
% total orders of the seven events that satisfy
% request_order/1.
consistent_orders(Count) :-
    request_order(Es),
    pairs_values(Es, Ps),
    Ps ins 1..7,
    all_distinct(Ps),
    findall(Ps, label(Ps), Orders),
    length(Orders, Count).

?- consistent_orders(N).
N = 6.
```

Six orders: the six interleavings of two ordered pairs into four positions. The program of `trace1.pl` is more constrained than the model, because the report is built from the rows the query returns, so the build cannot start before the query ends. One constraint more leaves one order:

```
% report_lifecycle(Events): request_order/1 with
% the knowledge that the build consumes the
% query's result.
report_lifecycle(Events) :-
    request_order(Events),
    memberchk(query_end-QE, Events),
    memberchk(build_start-BS, Events),
    QE #< BS.

?- report_lifecycle(_Es), pairs_values(_Es, Ps),
   Ps ins 1..7, all_distinct(Ps).
Ps = [1, 2, 3, 4, 5, 6, 7].
```

No labeling was needed: propagation alone found the order, which is the order of messages 11 to 17 in the trace. The point of the exercise is the direction of the inference. The constraints came from knowledge of the program, the order came out, and the trace confirms it; when the trace disagrees, either the knowledge or the trace is wrong, and the next predicates say which question to ask.

An **Impossible Trace** (page 614) is a trace whose recorded sequence could not have been produced by the program as the analyst understands it: a message that must follow another precedes it, a completion without a start, an event of a lifecycle out of its place. With the model in hand the check is the model applied to observed times. `agrees/2` unifies each observed event's time with the model's variable for it; a binding that violates a constraint fails inside `memberchk/2`, and the whole check fails:

```
% agrees(Model, Observed): every Name-Time pair of
% Observed is consistent with the Name-Var pairs
% of Model.
agrees(Model, Observed) :-
    maplist(observed_in(Model), Observed).

observed_in(Model, Event) :-
    memberchk(Event, Model).

% feasible(Observed): the observed report request
% events could have been produced by the program.
feasible(Observed) :-
    report_lifecycle(Model),
    agrees(Model, Observed).
```

The observed events of a request are collected from `trace1.pl` by text, with the times converted to integer milliseconds. The receipt and dispatch messages name the request; the query and build messages are those on the worker's thread between its dispatch and its completion, or to the end of the trace when it never completes:

```
% ms(Seconds, Ms): Ms is Seconds as integer ms.
ms(Seconds, Ms) :-
    Ms is round(Seconds * 1000).

% request_events(R, Events): the lifecycle events
% of request R found in the trace, as Name-Ms.
request_events(R, [received-Tr, dispatched-Td
```

```
                    | Rest]) :-
    once(request_msg(R, "received", _, Tr)),
    once(request_msg(R, "dispatched", W, Td)),
    (   once(request_msg(R, "completed", W, Tc))
    ->  true
    ;   Tc = inf
    ),
    findall(Name-T,
            worker_event(W, Td, Tc, Name, T),
            Rest).

request_msg(R, Verb, TID, Ms) :-
    format(string(Prefix), "Request ~w ~s",
           [R, Verb]),
    msg(_, Time, _, TID, appserver, Text),
    string_concat(Prefix, _, Text),
    ms(Time, Ms).

worker_event(W, Td, Tc, Name, Ms) :-
    msg(_, Time, _, W, _, Text),
    ms(Time, Ms),
    Ms > Td,
    Ms =< Tc,
    event_prefix(Prefix, Name),
    string_concat(Prefix, _, Text).

event_prefix("Query start", query_start).
event_prefix("Query end", query_end).
event_prefix("Build start", build_start).
event_prefix("Build end", build_end).
event_prefix("Request", completed).

?- request_events(1, _Es),
   forall(member(Name-T, _Es),
          format("~w ~w~n", [T, Name])).
1002 received
1003 dispatched
1004 query_start
1048 query_end
1049 build_start
1310 build_end
1311 completed
true.

?- request_events(1, _Es), feasible(_Es).
true.

?- request_events(5, _Es), feasible(_Es).
true.
```

Requests 1 and 5, the two report requests that complete, are feasible. The check does more than compare adjacent pairs: it posts the whole model and then the observations, so that an inconsistency between any two events is found, however far apart they are in the trace. A trace in which two messages were written in the wrong order, here the query's end before its start, is rejected:

```
?- request_events(1, [A, B, QS-T1, QE-T2 | Rest]),
   feasible([A, B, QS-T2, QE-T1 | Rest]).
false.
```

Two further uses of the same model follow from leaving a variable unbound. If the trace lacks a message, the model bounds the time at which the event it records must have happened. Without message 14, the end of the query of request 1, the model still knows the end came after the start and before the build:

```
?- request_events(1, _Es),
   selectchk(query_end-_, _Es, _Rest),
   report_lifecycle(_Model),
   agrees(_Model, _Rest),
   memberchk(query_end-QE, _Model).
QE in 1005..1048.
```

And if the trace ends before a request does, as for request 2, whose build never ends in `trace1.pl`, the model states what the missing events would have to satisfy, and no more:

```
?- request_events(2, _Es), length(_Es, N),
   report_lifecycle(_Model),
   agrees(_Model, _Es),
   memberchk(build_end-BE, _Model),
   memberchk(completed-C, _Model).
N = 5,
BE in 1552..sup,
BE#=<C+ -1,
C in 1551..sup.
```

Five events were found, the last of them the start of the build at 1.551 seconds. The build ended, if it ended, after that, and the request completed, if it completed, after the build ended; the solver keeps the second bound as a constraint between the two unknowns rather than tightening the domain of `C`, which it is entitled to do, since the domains it prints are always sound and not always the tightest. The dump, if it is of the same run, says it did neither: thread 102 is still inside `Report::Build`, as Chapter 28 discusses under **Small DA+TA** (page 622). The model cannot know that from the trace; it reports the lower bounds and leaves the upper ones at `sup`, which is the correct description of what a trace that ends at 14.010 seconds says about events after it. A model applied to the wrong kind of request fails too. Requests 6 to 9 are health checks with no query and no build, and `feasible/1` rejects them, because their completion comes one millisecond after dispatch and the model demands four events in between. The rejection is of the conjunction of model and trace, and here the model is at fault; a feasibility check refutes the analyst’s assumptions as readily as the artifact, and which one gave way is a judgment the solver does not make.

14.5 Deducing a Counter

A **Counter Value** (page 568) in the catalog is a number carried by messages, such as a queue depth, a sequence number, or a resource count, whose successive values are read as the trajectory of the state they measure. The trace of `trace1.pl` has such a counter. Messages 43, 45, and 47 report the

request queue's depth as 3, 4, and 5; messages 38, 40, and 42 record that requests 5, 6, and 7 were queued; requests 8 and 9 are received, in messages 44 and 46, without any message saying they were queued; and messages 53 to 65 dispatch requests 5 to 9 to the one worker that became free. The counter was logged three times. The constraint model recovers its value at every step, and decides whether requests 8 and 9 were queued, from the arithmetic of a queue: a queued request adds one, a dispatched request that was queued removes one, the depth is never negative, and a logged depth is what it says.

The events are recognized by their text, with the regular expressions of Chapter 13:

```
:- use_module(library(pcre)).

% queue_event(N, Event): message N of the trace
% is an event of the request queue.
queue_event(N, Event) :-
    msg(N, _, _, _, appserver, Text),
    queue_text(Text, Event).

queue_text(Text, received(R)) :-
    re_matchsub("^Request (?<r_I>\\d+) received",
                Text, Sub, []),
    get_dict(r, Sub, R).
queue_text(Text, queued(R)) :-
    re_matchsub("request (?<r_I>\\d+) queued$",
                Text, Sub, []),
    get_dict(r, Sub, R).
queue_text(Text, dispatched(R)) :-
    re_matchsub("^Request (?<r_I>\\d+) dispatch",
                Text, Sub, []),
    get_dict(r, Sub, R).
queue_text(Text, depth(D)) :-
    re_matchsub("queue depth (?<d_I>\\d+)",
                Text, Sub, []),
    get_dict(d, Sub, D).

% queue_events(From, To, Events): the queue events
% of messages From to To, in order.
queue_events(From, To, Events) :-
    findall(E, ( between(From, To, N),
                 queue_event(N, E) ),
            Events).

?- forall(( between(37, 47, N),
            queue_event(N, E) ),
          format("~w ~w~n", [N, E])).
37 received(5)
38 queued(5)
39 received(6)
40 queued(6)
41 received(7)
42 queued(7)
43 depth(3)
44 received(8)
45 depth(4)
46 received(9)
```

```
47 depth(5)
true.
```

The model walks the events with the depth as a constrained variable. Each received request gets an increment, a variable in `0..1` that is 1 when the request was queued and 0 when it went straight to a worker; the increment is fixed at 1 when a `queued` message for the request exists, and is otherwise left to the constraints. A dispatch subtracts the request's own increment, so that a request that was never queued does not empty the queue when it is dispatched:

```
% depth_model(Events, D0, Final, Incs): D0 is the
% queue depth before Events and Final after them;
% Incs pairs each received request with its
% increment, 1 if it was queued.
depth_model(Events, D0, Final, Incs) :-
    D0 #>= 0,
    findall(R, member(queued(R), Events), Queued),
    depth_steps(Events, Queued, D0, Final,
                [], Incs).

depth_steps([], _, D, D, Incs, Incs).
depth_steps([E | Es], Qd, D0, D, Incs0, Incs) :-
    depth_step(E, Qd, D0, D1, Incs0, Incs1),
    depth_steps(Es, Qd, D1, D, Incs1, Incs).

depth_step(received(R), Queued, D0, D1,
           Incs, [R-I | Incs]) :-
    (   memberchk(R, Queued)
    ->  I = 1
    ;   I in 0..1
    ),
    D1 #= D0 + I.
depth_step(queued(_), _, D, D, Incs, Incs).
depth_step(depth(V), _, D, D, Incs, Incs) :-
    D #= V.
depth_step(dispatched(R), _, D0, D1, Incs,
           Incs) :-
    memberchk(R-I, Incs),
    D1 #= D0 - I,
    D1 #>= 0.


?- queue_events(37, 66, _Es),
   depth_model(_Es, D0, Final, Incs).
D0 = Final, Final = 0,
Incs = [9-1, 8-1, 7-1, 6-1, 5-1].
```

Everything is determined, and no labeling was needed. The depth before message 37 was 0: three queued requests made a logged depth of 3, so nothing was in the queue before them. Requests 8 and 9 were queued, with increments of 1, because the logged depths went from 3 to 4 to 5 while they were the only requests received; the messages that would have said "request 8 queued" and "request 9 queued" are absent from the trace, a **Missing Message** (page 575) of Chapter 27, and the counter proves what they would have said. And the depth after message 66 is 0: five queued, five dispatched, the queue was drained by the time the worker wrote its last completion, which is consistent with the Housekeeping thread's report in message 52 that worker 101 was idle just before

it began. The region was chosen as messages 37 to 66, the span from the first queued request to the last dispatch, because the model of a dispatch as a dequeue holds only for a request that waited; the dispatches of requests 1 to 4, to idle workers, are outside it, and a model run over them would correctly leave their increments in `0..1`, undetermined by any logged depth.

The deduction depends on the model being right about the program: one counter, one increment per queued request, one decrement per dispatch. A program that batched its dispatches or logged the depth before incrementing it would need a different model, and a logged depth the model cannot reproduce would make `depth_model/4` fail, which, as in the previous section, is the solver reporting that the trace and the analyst's picture of the program disagree.

14.6 Thread Timelines

The dump of `dump1.pl` shows threads 103 and 104 deadlocked on two critical sections, the **Deadlock (Critical Sections)** (page 388) that Chapter 11 recognized as a cycle in the wait-for graph, and the trace shows the two threads' last messages at 2.003 and 2.103 seconds and the Housekeeping thread reporting their waits at 9.003. Neither artifact records when the locks were taken and when the waits began; the constraint model bounds those times from what the two artifacts do say. Four events matter: thread 103 acquires section A and later requests section B; thread 104 acquires section B and later requests section A. The dump's `critical_section/4` and `waits_for/3` facts say that each thread's request found the section held by the other, so each request came after the other thread's acquisition. The trace bounds the acquisitions from below by the dispatch of each worker, messages 24 and 28, and the requests from above by the watchdog's report:

```
% deadlock_timeline(A1, R1, A2, R2): the times in
% ms at which 103 acquired section A, 103
% requested section B, 104 acquired B, and 104
% requested A.
deadlock_timeline(A1, R1, A2, R2) :-
    A1 #< R1,
    A2 #< R2,
    A2 #< R1,
    A1 #< R2,
    A1 #>= 2001,
    A2 #>= 2101,
    R1 #=< 9003,
    R2 #=< 9003.

?- deadlock_timeline(A1, R1, A2, R2).
A1 in 2001..9002,
A1#=<R2+ -1,
A1#=<R1+ -1,
R1 in 2102..9003,
A2#=<R1+ -1,
A2 in 2101..9002,
A2#=<R2+ -1,
R2 in 2102..9003.
```

The deadlock is complete when the later of the two requests is made, and labeling for the smallest and the largest value of that instant gives the earliest and the latest scenario the constraints allow:

```
?- deadlock_timeline(A1, R1, A2, R2),
   Deadlock #= max(R1, R2),
   once(labeling([min(Deadlock)],
                 [A1, R1, A2, R2, Deadlock])).
A1 = 2001,
R1 = R2, R2 = Deadlock, Deadlock = 2102,
A2 = 2101.

?- deadlock_timeline(A1, R1, A2, R2),
   Deadlock #= max(R1, R2),
   once(labeling([max(Deadlock)],
                 [A1, R1, A2, R2, Deadlock])).
A1 = 2001,
R1 = 2102,
A2 = 2101,
R2 = Deadlock, Deadlock = 9003.
```

The deadlock closed no earlier than 2.102 seconds, one millisecond after worker 104 was dispatched, and no later than the watchdog's report; the earliest scenario has both requests at the same millisecond. The width of the interval is the honest result: between 2.103 and 9.003 seconds neither thread wrote a message, and nothing in either artifact narrows the gap. The watchdog's two messages, "worker 103 wait 7001 ms" and "worker 104 wait 6900 ms", seem to narrow it, if a wait is the time a thread has been blocked on its critical section. The model says they cannot be that:

```
?- deadlock_timeline(A1, R1, A2, R2),
   R1 #= 9003 - 7001,
   R2 #= 9003 - 6900.
false.
```

A wait of 7001 ms ending at 9003 began at 2002, before thread 104 was dispatched at 2101, and so before 104 could have acquired the section that 103 is blocked on. The hypothesis contradicts the dump, and the solver rejects it. The two figures match, instead, the times of messages 25 and 30, "Session 9 commit start" and "Session 9 lookup start": the watchdog reports how long each worker has been on its current operation, not how long it has been blocked. That is a reading of the program the analyst could have reached by inspection; the model reached it by refutation, and under the corrected reading the requests are bounded as before. Scheduling constraints proper, `serialized/2` for tasks that may not overlap on one resource and `cumulative/2` for a resource of bounded capacity, are in the same library and model a critical section as a resource of capacity one; the timelines of `trace1.pl` are too short to need them.

14.7 Diagnostic Use

A **Split Trace** is one activity recorded in several traces or logs, each with its own timestamps, that the analyst has to read as one; when the clocks disagree, the merge is wrong by the difference, and the difference is usually unknown. The application log of `trace1.pl` is treated here as one half of

such a split: the messages of the application server, the report engine, and the cache library, stamped by the process's clock. The other half is a log written by the database client with timestamps from a clock of its own, given below as `dblog/3` facts with times in milliseconds. Its content matches the `dbclient` messages of the trace, as a reader can check, but its clock does not.

```
% dblog(Time, TID, Text): the database client's
% own log of the same run; Time in ms of its own
% clock.
dblog(1265, 100, "Connecting to db01:5432").
dblog(1462, 100,
      "Connected to db01:5432, session=71").
dblog(2254, 101, "Query start: SELECT report 17").
dblog(2298, 101, "Query end: 1 row, 44 ms").
dblog(2752, 102, "Query start: SELECT report 18").
dblog(2800, 102, "Query end: 1 row, 48 ms").
dblog(3253, 103,
      "Send start: session=71, bytes=1024").
dblog(3352, 104,
      "Reply received: session=71, bytes=512").
dblog(10257, 101,
      "Query start: SELECT report 19").
dblog(10310, 101, "Query end: 1 row, 53 ms").
dblog(15251, 109, "Receive start: session=71").
dblog(15252, 109,
      "Error: buffer copy failed, code=0x3e6").

% app_msg(N, Ms): message N of the application
% log, at Ms milliseconds of the application's
% clock.
app_msg(N, Ms) :-
    msg(N, Time, _, _, Component, _),
    Component \== dbclient,
    ms(Time, Ms).
```

The skew is deduced from causal links the analyst knows: events of one log that must precede or follow events of the other. Six are stated here. The client cannot connect before the configuration that names the server is loaded, message 2, and the worker pool is started, message 6, only after the connection is up; a query for a report starts after the request for it was received and ends before the request completes:

```
% link(Order, Text, N): the dblog message Text
% happened before or after application message N.
link(after,  "Connecting to db01:5432", 2).
link(before, "Connected to db01:5432, session=71",
     6).
link(after,  "Query start: SELECT report 17", 11).
link(before, "Query end: 1 row, 44 ms", 17).
link(after,  "Query start: SELECT report 19", 37).
link(before, "Query end: 1 row, 53 ms", 58).
```

```
% skew(S): S is the offset in ms of the database
% client's clock: its time is the application's
% time plus S.
skew(S) :-
    findall(Order-(Tdb, Tapp),
            ( link(Order, Text, N),
              dblog(Tdb, _, Text),
              app_msg(N, Tapp) ),
            Links),
    maplist(skew_link(S), Links).

skew_link(S, after-(Tdb, Tapp)) :-
    Tdb - S #> Tapp.
skew_link(S, before-(Tdb, Tapp)) :-
    Tdb - S #< Tapp.

?- skew(S).
S in 1249..1251.
```

Six inequalities on one variable, and the clock of the database client is found to run 1249 to 1251 milliseconds ahead of the application's. Each link contributes a bound; the two tightest are the start of the query for report 17, which came after message 11 at 1002, and the connection, which was up before message 6 at 214. More links would narrow the interval further, and the analyst who knows more about the program adds them as `link/3` facts; links that contradict each other make `skew/1` fail, which is the **Impossible Trace** (page 614) check of Section 14.4 across two logs.

With the skew bounded but not known, the merged order of the two logs is determined for most pairs of messages and not for all. A message of the client log is before a message of the application log under every admissible skew exactly when the opposite order is inconsistent with the constraints, and the test is a negated conjunction of `skew/1` with the opposite order:

```
% relative_order(Text, N, Order): under every
% admissible skew, the dblog message Text is
% before or after application message N;
% undetermined if the skew interval allows both.
relative_order(Text, N, Order) :-
    dblog(Tdb, _, Text),
    app_msg(N, Tapp),
    (   \+ ( skew(S), Tdb - S #>= Tapp )
    ->  Order = before
    ;   \+ ( skew(S), Tdb - S #=< Tapp )
    ->  Order = after
    ;   Order = undetermined
    ).

?- relative_order("Query start: SELECT report 17",
   11, O).
O = after.

?- relative_order("Query start: SELECT report 17",
   12, O).
O = undetermined.
```

```
?- relative_order("Query end: 1 row, 44 ms",
   16, O).
O = before.

?- aggregate_all(count,
   ( dblog(_, _, T), app_msg(N, _),
     relative_order(T, N, undetermined) ), U).
U = 11.
```

The start of the query for report 17 is after the receipt of request 1 but may be before or after its dispatch, message 12 at 1003: the query's time on the application's clock is between 1003 and 1005, and the skew interval of three milliseconds straddles the dispatch. Of the 708 pairs of a client message and an application message, 11 are undetermined in this way, all of them a client message within three milliseconds of an application message, which is where the **Master Trace** of Chapter 28, the merge of a split trace into one sequence, has to say that it does not know. The merge itself is a sort by application time under a chosen skew:

```
% print_merged(S, From, To): print both logs
% merged under skew S, for application times
% From..To.
print_merged(S, From, To) :-
    findall(T-app(Text),
            ( app_msg(N, T),
              msg(N, _, _, _, _, Text) ),
            As),
    findall(T-db(Text),
            ( dblog(Tdb, _, Text),
              T is Tdb - S ),
            Ds),
    append(As, Ds, All),
    keysort(All, Sorted),
    forall(( member(T-Event, Sorted),
             T >= From, T =< To ),
           print_event(T, Event)).

print_event(T, app(Text)) :-
    format("~w app ~w~n", [T, Text]).
print_event(T, db(Text)) :-
    format("~w db  ~w~n", [T, Text]).

?- print_merged(1250, 1000, 1320).
1002 app Request 1 received: GET /report/17
1003 app Request 1 dispatched to worker 101
1004 db  Query start: SELECT report 17
1048 db  Query end: 1 row, 44 ms
1049 app Build start: report 17
1310 app Build end: report 17, 2611 nodes, 261 ms
1311 app Request 1 completed: 200, 309 ms
true.
```

At a skew of 1250 the merge reproduces the order of messages 11 to 17 of `trace1.pl`, which is the order the trace recorded with one clock. At 1251 the query's start would share the millisecond of the dispatch, and at 1249 the query's end would share the millisecond of the build's start; `keysort/2` is stable and keeps the application message first in a tie, so the merge under 1249 shows the build

starting before the query ended, an order the skew does not justify and the lifecycle model rejects. The two ties are exactly the two pairs that `relative_order/3` marks undetermined for this request, and an analyst who reads a merged trace at one of those pairs is reading the skew, not the program; a merge is to be printed with the undetermined pairs marked, or with the skew interval beside it.

The lifecycle model closes the loop. The merged events of request 1, with the client's times corrected by a skew of 1250, are the events `request_events/2` found in the single-clock trace, and `feasible/1` accepts them; corrected by a skew outside the interval `skew/1` found, 1300 for instance, the query would start at 954, before the request was received at 1002, and `feasible/1` rejects the merge:

```
?- feasible([received-1002, dispatched-1003,
             query_start-954, query_end-998,
             build_start-1049, build_end-1310,
             completed-1311]).
false.
```

The two predicates are the same knowledge used twice. `skew/1` asks which clock offsets make the two logs consistent with the program; `feasible/1` asks whether one merged sequence is. Both are refutations: a constraint model never shows that a trace is possible, only that it has not been shown impossible by what the analyst wrote down, and the chapters of Part IV that use these models say so each time.

Part II. Diagnostic Artifacts as Terms

Part II fixes what Parts III and IV depend on: how a memory dump and a software trace become knowledge bases of ground facts, which predicates carry them and why, what the facts can and cannot be taken to say about the system that produced the artifact, and how a diagnostic analysis pattern, which the catalogs describe in prose for an analyst, becomes a predicate that a program can run. The three chapters are definitional, and the conventions they state, the schema, the forms of a recognizer, the evidence term, and the threshold facts, are used without further comment in the parts that follow.

Chapter 15. Memory Dumps as Knowledge Bases

Part I used `dump1.pl` as given. This chapter says where such a file comes from, what each of its predicates means, and which decisions about representation it embodies, so that a knowledge base built from another dump by another analyst has the same shape and the predicates of Parts III and IV apply to it unchanged. The chapter is definitional: it fixes the schema of the memory dump knowledge base, the normalization rules a parser must apply, the representation of unknown and missing values, and the recording of provenance. The constructs are those of Part I, with a short grammar from Chapter 12 and the file handling of Chapter 13.

The chapter also fixes an epistemic convention. A memory dump is data: a copy of memory and registers written at one instant. Nearly everything an analyst reads about it is derived from that data by a debugger using information from outside the dump, above all symbol files, and a knowledge base built from debugger output records those derivations, not the dump. The predicates of the schema are therefore statements of the form “the debugger reported”, and the chapter says for each what was reported, by which command, and what the report depends on.

15.1 What a Dump Contains and What a Debugger Derives

A **process memory dump** written by the operating system, by a debugger, or by a capture tool is, on Windows, a file in the minidump format, and what it contains depends on the options its writer was given. Every such dump lists the modules loaded in the process with their base addresses, sizes, and version timestamps; lists the threads with their identifiers and the register context of each at the moment of the dump; and holds an exception record when the dump was written because of an exception. Everything else is an option of the **capture**. The memory the dump holds ranges from the thread stacks alone to what Microsoft’s documentation of the format calls all accessible memory of the process: memory the writer could read at that moment, which excludes pages it could not, and which a further option lets the writer skip rather than fail on. Filtering options reduce the stacks to the values a stack walk needs, and other options exclude memory not essential to the dump’s purpose. Optional streams, among them the handle data, the list of unloaded modules, and per-thread information such as creation time and processor times, are present only when requested. A debugger’s `.dump /ma` asks for full memory with the optional streams; other tools and other option sets write less. The dump contains no stack traces, no function names, and no notion of a wait, a critical section, or a heap.

The facts of the schema therefore depend on the capture twice: on which commands were run, and on what the dump gave the commands to read. `!handle` reports nothing from a dump without the handle data stream; `!heap -s` reports nothing from a dump without the heap manager’s memory; a stack trace stops at the first page the dump does not hold. An empty relation, a missing thread, or a count that disagrees with another is evidence about the process only once the capture is known to have included the data, and Section 15.5 records the capture in the knowledge base and states the rule.

The derived facts come from the debugger. A **stack trace** is a reconstruction: starting from the registers in the thread's context, the debugger walks the stack memory using unwind information stored in the module images, or, for code without such information, heuristics about frame pointers, and at each step it looks up the return address in the symbol files of the module containing it to produce a name and an offset. With different symbol files the names change; without symbol files frames become offsets from module bases; on a corrupt stack the walker produces frames that were never executed. The `k` command prints the result of this procedure, and `frame/5` records what `k` printed. The same holds for critical sections, which the `!locks` extension finds by walking the loader's list of critical section debug records in the dumped memory, for heaps, which `!heap -s` summarizes from the heap manager's structures, and for the handle table summary, which comes from a handle data stream that not every dump has.

Four consequences shape the schema. First, a fact is as reliable as the command that produced it, and the knowledge base should say which command that was; Section 15.5 does so with `source/3`. Second, the debugger distinguishes what it could not determine from what it found absent, and the knowledge base must preserve that distinction; Section 15.5 gives the rule. Third, a command reports only what the capture gave it, and the knowledge base should record the capture, so that the absence of facts can be read; Section 15.5 does so with `dump_info/2`. Fourth, two analysts running the same commands on the same dump with the same symbols get the same facts, and the pipeline of Section 15.3 is a procedure, not a judgment: judgment begins with the predicates of Part III.

15.2 The Schema

The **schema** of a memory dump knowledge base is the set of predicates below with the argument conventions stated for each. Every knowledge base in this book that represents a process memory dump uses exactly these predicates; a chapter that needs more defines an extension and says so.

Predicate	Arguments	Source
`process/2`	PID, Name	`!analyze -v`
`module/3`	Name, Base, Size	`lm`
`module_info/3`	Name, Key, Value	`lm`, `lmv`
`thread/2`	TID, PID	`~*kn`
`thread_time/3`	TID, UserMs, KernelMs	`!runaway 7`
`thread_state/2`	TID, State	derived
`frame/5`	TID, N, Module, Function, Offset	`~*kn`
`exception_record/3`	TID, Code, Address	`.exr -1`
`exception_info/3`	TID, Key, Value	`.exr -1`
`critical_section/4`	Addr, Owner, WaiterCount, RecursionCount	`!locks`
`waits_for/3`	TID, Kind, Addr	`k`, `!cs`
`handle_count/3`	PID, Type, Count	`!handle 0 0`
`heap/4`	Addr, ReservedKb, CommittedKb, FreeKb	`!heap -s`
`last_error/2`	TID, Code	`!teb`

Predicate	Arguments	Source
`hooked/3`	Function, Module, Hooker	analyst

The commands in the Source column are the ones whose output the parser of Section 15.3 reads, spelled as they are typed. `~*kn` is the `k` command applied to every thread by the `~*` prefix, with the `n` option, which adds the column of frame numbers; without the option `k` prints no frame numbers, and the grammar of Section 15.3 requires them. The shared dump is consulted here so that the facts of each relation can be shown as the relation is discussed:

```
:- [dump1].
```

The identifiers come first. `process(PID, Name)` records the process identifier as an integer and the image name as an atom spelled as the debugger shows it, `'AppServer.exe'`, with its quotes. `thread(TID, PID)` records every thread of the dump with the process it belongs to; the second argument is redundant in a process dump, which has one process, and is kept because Section 15.6 extends the schema to dumps with several. Identifiers are integers because they are compared and used as keys; the debugger prints them in hexadecimal, and `0x1018` and `4120` are the same integer.

`module(Name, Base, Size)` records each loaded module by a lowercase atom, its base address, and its size in bytes, both integers, so that the containment test of Chapter 6 is arithmetic. `module_info(Name, Key, Value)` holds what else the debugger reports about a module, one fact per key, with three keys fixed: `version`, an atom such as `'3.2.1.0'`, because versions are compared as text, not as numbers; `symbols`, one of the atoms `pdb`, `export`, and `none`, the three degrees of symbol information the debugger distinguishes; and `company`, an atom from the module's version resource. A key-value relation rather than a module with many arguments is used because the set of attributes a debugger reports varies with the module and with the command, and a missing attribute is then a missing fact rather than a placeholder argument.

`thread_time(TID, UserMs, KernelMs)` records processor time in milliseconds, two integers, as `!runaway` reports it with its flags set for user and kernel time. `thread_state(TID, State)` is the one relation in the table marked derived. A user-mode dump has no field that says why a thread was not running; the state is a classification the parser makes from three recorded things, the suspend count in the thread header, the exception record, and the top frame of the stack trace. `suspended` is a suspend count above zero, `waiting(exception)` is the thread named by the exception record, `waiting(user_request)` is a top frame in a wait system call, and `running` is the rest. The reason inside `waiting/1` borrows the vocabulary of the kernel's wait reasons, which a kernel dump reports directly, and is a compound term so that a clause can match any wait or one wait in particular, as Chapter 2 said.

`frame(TID, N, Module, Function, Offset)` is one fact per frame of every stack trace. `N` counts from 0 at the top; `Module` is the lowercase atom that keys `module/3`; `Function` is a quoted atom spelled exactly as the debugger printed it, scope operators and template brackets included, or the atom `'Unknown'` when the debugger printed no function; `Offset` is the integer after `+0x`. The frame is a fact and not an element of a per-thread list, so that any frame is reached by thread and number

through indexing and so that a stack trace is built, when needed, by the predicates of Chapters 5, 7, and 9. The debugger's two address columns, the child stack pointer and the return address, are not recorded; Section 15.3 shows the parser reading and discarding them.

`exception_record(TID, Code, Address)` records the exception record: the faulting thread, the status code as an integer, and the faulting instruction's address. `exception_info(TID, Key, Value)` holds the record's parameters by key, `access` with value `read`, `write`, or `execute`, and `target` with the address accessed; for an access violation the debugger decodes the two parameters into exactly these. A key-value relation is used for the same reason as for modules: other exception codes carry other parameters.

`critical_section(Addr, Owner, WaiterCount, RecursionCount)` records each critical section the debugger listed, by address, with the owning thread, `0` when the section is unowned, and two counts. `WaiterCount` is the number of threads waiting to enter the section. It is a normalized value, produced by the import step of Section 15.3 from what the debugger displayed, and not a field of the critical section structure. The structure's field is `LockCount`, and its meaning depends on the Windows version. Microsoft's documentation describes it for Windows XP and 2000 as a counter, and its example of a section entered twice by its owner shows `LockCount` 1 and `RecursionCount` 2 with no thread waiting, so that the counter is not the number of waiters; for Windows Server 2003 SP1 and later it describes a bit encoding in which bit 0 set means the section is unlocked, bit 1 set means no waiting thread has been woken, and the remaining bits hold the ones' complement of the number of waiting threads. A parser must therefore know which value it is reading and decode it; Section 15.4 gives the decoding for the later encoding. Which commands display the field as stored and which a decoded count is a property of the debugger and of the version of its extensions: `dt` on the structure displays the field, and whether `!locks` or `!cs` displays the field or a count in a given installation is not fixed by this book. The shared data is synthetic, and its waiter counts are normalized by assumption: the values `2`, `1`, and `0` were written as waiter counts and were not decoded from a displayed field. `RecursionCount` is the owner's recursive acquisition count as the debugger displays it, `1` for a section entered once. Chapter 20 reads `WaiterCount` as a measure of contention and compares it with the threads whose stack traces wait for the section, and in the shared data the two agree. `waits_for(TID, Kind, Addr)` records that a thread was blocked waiting for the synchronization object of the given kind at the given address, with `Kind` the atom `critical_section` in the shared data and open to other kinds in extensions. The two relations are separate because they come from different places and have different directions: the first describes an object and names a thread, the second describes a thread and names an object, and a wait chain is the join of the two, as Chapter 4 showed.

```
?- critical_section(A, _, WaiterCount, _),
   aggregate_all(count,
      waits_for(_, critical_section, A), Waiting).
A = 140695026507968,
WaiterCount = Waiting, Waiting = 2 ;
A = 140695026508096,
WaiterCount = Waiting, Waiting = 1 ;
A = 140695026508224,
WaiterCount = Waiting, Waiting = 0.
```

The toplevel writes `WaiterCount = Waiting, Waiting = 2` when two variables of the query are bound to the same value, which is the agreement in each row. The agreement is a check of the fixture's internal consistency, that its waiter counts and its wait facts were written to match; it does not validate the import convention, which only a real fragment and its decoding can. A knowledge base in which the two numbers differ for some section has either a parser that recorded the raw field, or decoded it under the wrong encoding, which Chapter 20 reports as a count mismatch, or a dump in which a waiting thread's stack trace does not show the wait; the recognizer of Chapter 20 cannot tell these apart, and the provenance of Section 15.5 is where the analyst looks first.

`handle_count(PID, Type, Count)` is the handle table summary by object type; `heap(Addr, ReservedKb, CommittedKb, FreeKb)` is one fact per heap with the three sizes in kilobytes as the heap summary prints them; `last_error(TID, Code)` is the last error value from a thread's environment block, recorded for the threads the analyst inspected. `hooked(Function, Module, Hooker)` is the one relation in the table attributed to the analyst rather than to a command: it records that the entry of `Function` in `Module` was found to transfer control into `Hooker`, a finding made by comparing code in the dump with the module image, and the knowledge base records the finding without the method.

The schema is itself data. Chapter 8 related the field names of `frame/5` to argument positions with `frame_field/2`; the following does the same for every relation, and `field/3` reads any field of any fact by name.

```
% schema(Relation, Fields): Relation is a relation
% of the dump schema; Fields names its arguments
% in order.
schema(process/2, [pid, name]).
schema(module/3, [name, base, size]).
schema(module_info/3, [name, key, value]).
schema(thread/2, [tid, pid]).
schema(thread_time/3, [tid, user_ms, kernel_ms]).
schema(thread_state/2, [tid, state]).
schema(frame/5,
       [tid, number, module, function, offset]).
schema(exception_record/3, [tid, code, address]).
schema(exception_info/3, [tid, key, value]).
schema(critical_section/4,
       [address, owner, waiters, recursion]).
schema(waits_for/3, [tid, kind, address]).
schema(handle_count/3, [pid, type, count]).
schema(heap/4,
       [address, reserved, committed, free]).
schema(last_error/2, [tid, code]).
schema(hooked/3, [function, module, hooker]).
```

```
% field(Fact, Name, Value): Value is the argument
% of Fact that the schema names Name.
field(Fact, Name, Value) :-
    functor(Fact, F, A),
    schema(F/A, Fields),
    nth1(Index, Fields, Name),
    arg(Index, Fact, Value).

% fact_count(Relation, Count): Count facts of the
% schema relation Relation are loaded.
fact_count(F/A, Count) :-
    schema(F/A, _),
    functor(Goal, F, A),
    aggregate_all(count, Goal, Count).
```

`field/3` takes the functor of the fact with `functor/3`, finds the position of the name with `nth1/3`, and reads the argument with `arg/3`; with the name unbound it enumerates the fields. `fact_count/2` builds a most general goal for the relation with `functor/3` in its constructive mode and counts its solutions, which is how a program can ask about a relation it was not written for.

```
?- field(critical_section(1, 2, 3, 4), N, V).
N = address,
V = 1 ;
N = owner,
V = 2 ;
N = waiters,
V = 3 ;
N = recursion,
V = 4.

?- fact_count(frame/5, N).
N = 78.

?- forall(fact_count(R, C),
   format("~w~t~20|~t~d~4+~n", [R, C])).
process/2              1
module/3              10
module_info/3         20
thread/2              10
thread_time/3         10
thread_state/2        10
frame/5               78
exception_record/3     1
exception_info/3       2
critical_section/4     3
waits_for/3            3
handle_count/3         6
heap/4                 3
last_error/2           6
hooked/3               2
true.

?- aggregate_all(sum(C), fact_count(_, C), Total).
Total = 165.
```

The shared dump is 165 facts, 78 of them frames. A real dump of a server process has hundreds of threads and thousands of frames, and the same fifteen relations.

15.3 From Debugger Output to Facts

The **pipeline** from a dump to a knowledge base has four steps: the debugger writes a log; a parser reads the log and produces terms; the terms are written to a file as facts; the file is consulted. The steps are separate so that each can be checked, and so that the log and the facts file both remain as records. The checks this section builds in are four: every line of a command's output parses or the pipeline stops with an error that names the line; the facts written are as many as the lines read; the keys of the facts are distinct; and the written file, read back, is the same collection of facts as a reference knowledge base, in both directions.

The log is produced in the debugger session. `.logopen` with a file name starts copying everything the debugger prints into that file, and `.logclose` stops. Between them the analyst runs the commands of the Source column of Section 15.2: `!analyze -v` for the process name and the faulting thread, `lm` and `lmv` for modules, `~*kn` for the stack traces of all threads with frame numbers, `!runaway 7` for thread times, `.exr -1` for the exception record, `!locks` for critical sections, `!handle 0 0` for the handle summary, `!heap -s` for heaps, and `!teb` on each thread of interest for its last error. The order does not matter to the parser, which recognizes each command's output by its shape, but it helps to keep the same order in every session, so that logs are comparable. The analyst also records how the dump was written, since the log does not say; Section 15.5 gives the facts for it.

The parser is a grammar in the sense of Chapter 12. Each command's output is a sequence of lines of a known shape, and a DCG for each shape turns a line into a term. The output of `k` is the central case. For each thread, `~*kn` prints a header line with the thread's position, its process and thread identifiers in hexadecimal after `Id:`, its suspend count, and its environment block address; then a column header; then one line per frame with the frame number in hexadecimal, the child stack pointer, the return address, and the call site as `module!function+0xoffset`. The frame number column is what the `n` option adds; plain `k` prints the other three columns without it, and a parser given plain `k` output has to count lines itself, which the grammar below does not do. The following is the `kn` output for thread 108 of the shared dump below its thread header, five frames, each frame line wrapped here because it is wider than the page; in `dump1.log` these are lines 212 to 216:

```
 # Child-SP          RetAddr               Call Site
00 0000004a`2b3ff8c8 00007ffb`b0b2a09e
   ntdll!NtDelayExecution+0x14
01 0000004a`2b3ff8d0 00007ff6`1d10503a
   KERNELBASE!SleepEx+0x9e
02 0000004a`2b3ff970 00007ffb`b1c14014
   appserver!Housekeeping::Run+0x3a
03 0000004a`2b3ff9b0 00007ffb`b2a52651
   KERNEL32!BaseThreadInitThunk+0x14
04 0000004a`2b3ff9e0 00000000`00000000
   ntdll!RtlUserThreadStart+0x21
```

The addresses are invented, as everything in the shared data is, but the shape is the debugger's: 64-bit addresses are printed as two 32-bit halves separated by a backquote, and the modules `kernelbase` and `kernel32` appear as `KERNELBASE` and `KERNEL32`, the spelling of their export names, which is the first thing the parser normalizes. The grammar for one frame line, given the thread identifier from the header, is:

```
:- use_module(library(dcg/basics)).

% k_line(TID, Frame)//: one frame line of kn
% output for thread TID, as a frame/5 term.
k_line(TID, frame(TID, N, M, F, O)) -->
    xinteger(N), blanks,
    address(_ChildSP), blanks,
    address(_RetAddr), blanks,
    symbol(M, F, O), eos.

% address(A)//: a 64-bit address, two hexadecimal
% halves separated by a backquote.
address(A) -->
    xinteger(Hi), "`", xinteger(Lo),
    { A is Hi << 32 \/ Lo }.

% symbol(M, F, O)//: module!function+0xoffset, or
% module+0xoffset for a frame without a symbol.
symbol(M, F, O) -->
    string_without("!+", MC),
    { atom_codes(M0, MC), downcase_atom(M0, M) },
    (   "!"
    ->  string_without("+", FC),
        { atom_codes(F, FC) }
    ;   { F = 'Unknown' }
    ),
    "+0x", xinteger(O).
```

`xinteger//1` from `library(dcg/basics)` reads a run of hexadecimal digits, which is what the frame number and each half of an address are. `string_without//2` reads up to, but not including, any of the given characters, so the module name ends at `!` or `+`, and the function name at `+`; the if-then-else in the grammar body takes the `!` branch when there is one and otherwise records the function as `'Unknown'`, the schema's convention for the `module+0xoffset` form the debugger prints when it has no symbol. `eos//0` requires the line to end after the offset. The address halves are combined by shifting; the two addresses are read and then dropped, because the schema does not keep them, and their variables are named with a leading underscore so that the compiler does not report them as singletons.

The lines are too wide to appear as strings in a program block of this book, so the following assembles each from its columns and presents them as the relation `log_line(Log, LineNo, Text)`, which stands for what a real parser gets from the log file with `read_line_to_string/2` and a line counter: the name of the log, the number of the line in it, and the text. The line number is what locates an error.

```
% k_columns(LineNo, N, ChildSP, RetAddr,
% CallSite): the columns of log line LineNo of
% dump1.log, a frame line of thread 108.
k_columns(212, '00', '0000004a`2b3ff8c8',
          '00007ffb`b0b2a09e',
          'ntdll!NtDelayExecution+0x14').
k_columns(213, '01', '0000004a`2b3ff8d0',
          '00007ff6`1d10503a',
          'KERNELBASE!SleepEx+0x9e').
k_columns(214, '02', '0000004a`2b3ff970',
          '00007ffb`b1c14014',
          'appserver!Housekeeping::Run+0x3a').
k_columns(215, '03', '0000004a`2b3ff9b0',
          '00007ffb`b2a52651',
          'KERNEL32!BaseThreadInitThunk+0x14').
k_columns(216, '04', '0000004a`2b3ff9e0',
          '00000000`00000000',
          'ntdll!RtlUserThreadStart+0x21').

% log_line(Log, LineNo, Text): line LineNo of the
% log file Log is Text. dump1.log holds the frame
% lines of thread 108 as the debugger printed
% them; bad.log holds the same lines with the
% offset of line 214 removed, a form the grammar
% does not accept, for the check at the end of
% this section.
log_line('dump1.log', LineNo, Text) :-
    k_columns(LineNo, N, SP, Ret, Sym),
    atomic_list_concat([N, SP, Ret, Sym], ' ',
                       Text).
log_line('bad.log', LineNo, Text) :-
    log_line('dump1.log', LineNo, Text0),
    (   LineNo =:= 214
    ->  atom_concat(Text, '+0x3a', Text0)
    ;   Text = Text0
    ).

% parse_k_line(TID, Text, Frame): Frame is the
% frame/5 term for the frame line Text of TID.
parse_k_line(TID, Text, Frame) :-
    atom_codes(Text, Codes),
    phrase(k_line(TID, Frame), Codes).


?- log_line('dump1.log', _, L),
   parse_k_line(108, L, F),
   portray_clause(F), fail.
frame(108, 0, ntdll, 'NtDelayExecution', 20).
frame(108, 1, kernelbase, 'SleepEx', 158).
frame(108, 2, appserver, 'Housekeeping::Run', 58).
frame(108, 3, kernel32, <...>, 20).
frame(108, 4, ntdll, 'RtlUserThreadStart', 33).
false.

?- forall(log_line('dump1.log', _, L),
   ( parse_k_line(108, L, F), call(F) )).
true.
```

The parsed terms are printed by `portray_clause/1`, which writes a term as a clause, quoted where quoting is needed and terminated by a full stop, in the form `consult/1` reads back; the function name of the fourth frame, `'BaseThreadInitThunk'`, is elided in the transcript for width. The second query calls each parsed term as a goal against the consulted `dump1.pl` and finds all five there: the parser reproduces the shared facts for these frames. The parse is deterministic and leaves no choice point, because every nonterminal in the grammar is deterministic: `xinteger//1` and `string_without//2` take the longest match and the choice in `symbol//3` is an if-then-else.

The second query is not yet a check of the pipeline, because it is silent about a line that does not parse. `forall/2` runs its action for every solution of its generator, and a line that fails `parse_k_line/3` in the generator position is simply not a solution: it is skipped, the remaining lines are checked, and the query succeeds. A writer built the same way would drop the line from the facts file without a word, and a knowledge base failing the `frames_contiguous` invariant of Section 15.7 would be the first sign, if any frame followed the dropped one. The pipeline must instead treat a line it cannot parse as an error of the kind Section 7.6 distinguished from failure: the line is not an argument the grammar is defined on. The third step therefore parses every record before it writes anything, inside a goal that is called once per record and that raises when the record is rejected:

```
% parse_record(Log, TID, LineNo-Text, Frame):
% Frame is the frame/5 term for the frame line
% Text of TID; a line the grammar rejects raises a
% syntax error located at Log:LineNo.
parse_record(Log, TID, LineNo-Text, Frame) :-
    (   parse_k_line(TID, Text, Frame0)
    ->  Frame = Frame0
    ;   format(atom(Where), "~w:~d: ~w",
               [Log, LineNo, Text]),
        throw(error(syntax_error(k_line),
                    context(parse_record/4,
                            Where)))
    ).

% check_keys(Frames): the (TID, N) keys of Frames
% are distinct; a repeated key raises an error.
check_keys(Frames) :-
    findall(T-N,
            member(frame(T, N, _, _, _), Frames),
            Keys),
    msort(Keys, Sorted),
    (   append(_, [Key, Key | _], Sorted)
    ->  domain_error(unique_frame_key, Key)
    ;   true
    ).

% write_frames(Log, TID, File): parse every frame
% line of TID in Log and write the frame/5 facts
% to File, one per line. Nothing is written until
% every line has parsed and the keys are distinct.
write_frames(Log, TID, File) :-
    findall(LineNo-Text,
            log_line(Log, LineNo, Text),
            Records),
    maplist(parse_record(Log, TID),
```

```
            Records, Frames),
    check_keys(Frames),
    setup_call_cleanup(
        open(File, write, S),
        maplist(portray_clause(S), Frames),
        close(S)).
```

`parse_record/4` is the checked action. Its if-then-else parses into a fresh variable and unifies the result with the output argument only on success, so that a caller who passes a bound `Frame` gets failure rather than a syntax error when the frame does not match; on rejection it builds the location as an atom with `format/3` writing to `atom(Where)`, and raises the standard error term of Section 7.6 with the formal `syntax_error(k_line)`, naming the nonterminal that rejected the line, and a context that names the predicate and carries the log, the line number, and the text. `maplist/3` applies it to every record, with `Log` and `TID` supplied by partial application as Chapter 10 showed, and fails or raises if any application does, so no record can be lost: the output list has one frame per input record by construction. `check_keys/1` sorts the keys with `msort/2`, which keeps duplicates where `sort/2` would remove them, and `append(_, [Key, Key | _], Sorted)` finds two equal neighbors if there are any; `domain_error/2` from `library(error)` raises with the offending key. Only then is the file opened, written with `portray_clause/2` under `setup_call_cleanup/3` from Chapter 7, which guarantees it is closed, and the order means that a rejected line leaves no file behind, not even a partial one.

The check that the written file agrees with a reference knowledge base must run in both directions. A check that calls every term of the file as a goal establishes that nothing wrong was written; it does not establish that nothing was left out, and an empty file passes it. The following reads the file back and compares complete collections:

```
% read_facts(File, Facts): Facts are the terms of
% File, in order.
read_facts(File, Facts) :-
    setup_call_cleanup(
        open(File, read, S),
        read_terms(S, Facts),
        close(S)).

read_terms(S, Facts) :-
    read_term(S, Term, []),
    (   Term == end_of_file
    ->  Facts = []
    ;   Facts = [Term | Rest],
        read_terms(S, Rest)
    ).

% frames_agree(Log, TID, File): File holds one
% fact per frame line of TID in Log, and those
% facts are exactly the frame/5 facts of TID in
% the loaded knowledge base, each side counted
% with repetitions.
frames_agree(Log, TID, File) :-
    aggregate_all(count, log_line(Log, _, _), In),
    read_facts(File, Written),
    length(Written, In),
```

```
    findall(frame(TID, N, M, F, O),
            frame(TID, N, M, F, O), Loaded),
    msort(Written, Canonical),
    msort(Loaded, Canonical).

?- write_frames('dump1.log', 108, 'frames108.pl').
true.

?- read_file_to_string('frames108.pl', S, []),
   write(S).
frame(108, 0, ntdll, 'NtDelayExecution', 20).
frame(108, 1, kernelbase, 'SleepEx', 158).
frame(108, 2, appserver, 'Housekeeping::Run', 58).
frame(108, 3, kernel32, <...>, 20).
frame(108, 4, ntdll, 'RtlUserThreadStart', 33).
S = <...>.

?- frames_agree('dump1.log', 108, 'frames108.pl').
true.
```

`read_facts/2` reads the file with `read_term/3` into a list. `frames_agree/3` then checks three things. `length(Written, In)` with `In` already bound to the number of log lines checks the count: as many facts out as lines in. The two `msort/2` calls bring both collections into one canonical order without removing repetitions, and the second unifies its result with the first, which holds only when the two lists are equal element by element: every written fact is loaded and every loaded fact of the thread was written, with the same multiplicity. A repeated fact on one side, a missing fact on the other, or a differing offset each makes the unification fail. The comparison is against the five `frame/5` facts of thread 108 in `dump1.pl`, so the test is that the regenerated file agrees with the loaded one; here it does, and the file is not consulted a second time, which would redefine `frame/5`.

The counterexample that the pipeline must catch is a log with a line the grammar does not accept. The grammar accepts two forms of call site, `module!function+0xoffset` and `module+0xoffset`, and no others. The second clause of `log_line/3` above defines `bad.log` as `dump1.log` with the `+0x3a` removed from line 214, by `atom_concat/3` run backwards with its result and its suffix known, so that the third call site reads `appserver!Housekeeping::Run` alone:

```
?- log_line('bad.log', 214, T),
   sub_atom(T, _, 27, 0, CallSite).
T = <...>,
CallSite = 'appserver!Housekeeping::Run'.

?- catch(
       write_frames('bad.log', 108, 'bad108.pl'),
       error(E, context(_, Where)), true).
E = syntax_error(k_line),
Where = 'bad.log:214: 02 <...>Housekeeping::Run'.

?- exists_file('bad108.pl').
false.

?- write_frames('bad.log', 108, 'bad108.pl').
ERROR: Syntax error: k_line (bad.log:214: 02 <...>
...
```

This is a check of a boundary case: the line that does not parse is the third of five, and the pipeline stops at it. `catch/3` shows the error term: the formal names the grammar, and the context names the log, the line number, and the line's text, which the transcript elides for width. No file was created, since parsing precedes opening. Uncaught, the same error is printed by the toplevel with its message and a backtrace, abridged here. Nothing is silently lost: the pipeline either writes all five frames or writes none and says which line it could not read. Whether `appserver!Housekeeping::Run` without an offset should parse is a separate question about the grammar; a debugger version or an extension that prints call sites in a further form needs a further branch of `symbol//3`, and until it has one the pipeline rejects such lines rather than dropping them. The key check is exercised on constructed terms, since the sample has no repeated frame number:

```
?- catch(check_keys([ frame(108, 0, a, 'A', 1),
                      frame(108, 1, b, 'B', 2),
                      frame(108, 0, c, 'C', 3)
                    ]),
         error(E, _), true).
E = domain_error(unique_frame_key, 108-0).

?- check_keys([ frame(108, 0, a, 'A', 1),
                frame(109, 0, b, 'B', 2) ]).
true.
```

The same frame number twice for one thread is a repeated key, which is what a log with a thread's output pasted twice, or a parser that counted frames itself and restarted its count, would produce; frame 0 of two threads is not.

The fourth step is `consult/1` of the written file, which is how `dump1.pl` was loaded in every chapter of Part I.

`portray_clause/1` writes integers in decimal, so the offsets appear as `20` and `158` where `dump1.pl` has `0x14` and `0x9e`. Both are the same integer to `consult/1`, and the hexadecimal form in the shared file is for the reader. A parser that writes facts for people to read uses `format/2` with `~q` for atoms, which quotes them as `portray_clause/1` does, and `~16r` for the values that the debugger shows in hexadecimal:

```
% write_frame(Frame): write a frame/5 term as a
% fact with a hexadecimal offset.
write_frame(frame(T, N, M, F, O)) :-
    format("frame(~d, ~d, ~q, ~q, 0x~16r).~n",
           [T, N, M, F, O]).

?- frame(109, 0, M, F, O),
   write_frame(frame(109, 0, M, F, O)).
frame(109, 0, dbclient, 'Buffer::Copy', 0x1a).
M = dbclient,
F = 'Buffer::Copy',
O = 26.
```

The printed line is the fact as it stands in `dump1.pl`, and the bindings below it are the same offset as the toplevel prints it.

The other commands' outputs are parsed the same way, one grammar per line shape, and the terms for all of them are written to one file. A log of a few megabytes becomes a facts file of a few megabytes, and consulting it takes well under a second in this edition; Chapter 30 measures larger cases.

15.4 Normalization

A knowledge base is useful only if the same thing is written the same way everywhere in it, because unification is syntactic: `kernelbase` and `'KERNELBASE'` are different atoms, and a query for one does not find the other. The parser is where sameness is imposed, and the **normalization** rules of the schema are the following.

Module names are lowercase atoms without a file extension. The debugger prints a module by the name in its export table in some places and by its file name in others, so the same module can appear as `KERNELBASE`, `KERNELBASE.dll`, and `kernelbase` in one log, and the parser maps all three to `kernelbase`. The rule is applied wherever a module is named: in `module/3`, `module_info/3`, `frame/5`, and `hooked/3`, so that the name is a key that joins them. `file_name_extension/3` splits a file name from its extension, and `downcase_atom/2` lowercases:

```
% module_atom(Text, Module): Module is the schema
% atom for a module the debugger names Text.
module_atom(Text, Module) :-
    file_name_extension(Stem, _, Text),
    downcase_atom(Stem, Module).

?- module_atom('KERNELBASE.dll', M).
M = kernelbase.

?- module_atom('KERNELBASE', M).
M = kernelbase.

?- module_atom('AppServer.exe', M).
M = appserver.

?- forall(frame(_, _, M, _, _), module(M, _, _)).
true.

?- forall(hooked(_, M, H),
   ( module(M, _, _), module(H, _, _) )).
true.
```

The last two queries are the referential integrity of the rule over the shared dump: every module named in a frame or a hook is a loaded module under the same spelling. A knowledge base in which they fail has a parser that normalized inconsistently, and the failing conjunction, in the manner of Chapter 7, names the module.

Function names are atoms spelled exactly as the debugger printed them, with no case change and no splitting, because an analyst searches for `'Buffer::Copy'` as it appeared on the screen and because the scope operators and template brackets carry information Chapter 8 showed how to extract. The one substitution is `'Unknown'` for a frame without a symbol. The debugger's own text for such a frame, `cachelib+0x2a10`, has no function part, and the schema fills the position with one fixed atom so that `frame(_, _, _, 'Unknown', _)` finds every such frame.

Addresses, sizes, offsets, counts, times, and identifiers are integers. The debugger prints addresses in hexadecimal with a backquote, offsets after `+0x`, identifiers in hexadecimal after `Id:` and in decimal elsewhere, and times as hours, minutes, seconds, and milliseconds; every one of them becomes an integer, so that arithmetic and comparison apply and so that the same value from two commands unifies. Exception codes and status codes are integers for the same reason: Chapter 7 classified them by their top bits. Version strings are the exception, kept as atoms, because `'3.2.1.0'` is not a number and because two versions are compared as text.

Thread identifiers in `frame/5` are the decimal integers that `~*kn` prints in hexadecimal in its headers; the parser converts them once, and the schema's thread 108 is the debugger's thread `6c`.

```
?- X is 0x1018, Y is 0x6c.
X = 4120,
Y = 108.
```

The one normalization that decodes rather than respells is the waiter count of `critical_section/4`. Section 15.2 said that the `LockCount` field of the critical section structure is, from Windows Server 2003 SP1 onward, a bit encoding: bit 0 set means the section is unlocked, bit 1 set means no waiting thread has been woken, and the bits above the two hold the ones' complement of the number of waiting threads. The field is a signed 32-bit integer; an unlocked section with no waiters holds `-1`, every bit set. In the two's complement representation that Chapter 6 described, the ones' complement of an integer `X` is `-1 - X`, which SWI-Prolog also writes `\ X`, so the complement of the field holds the waiter count in the bits above the two flags, and a right shift by two drops the flags. With fewer than 2^29 waiters the complemented bits leave the sign bit of the field set, so every value the encoding produces is negative, and a nonnegative value decodes to a negative count: the predicate raises a domain error for it rather than return a number, which catches a field read as an unsigned 32-bit number before sign extension as well as a corrupt one.

```
% raw_fields(+Raw, -State, -Woken, -Waiters): the
% fields packed in Raw, the LockCount field of a
% critical section as Windows Server 2003 SP1 and
% later store it: bit 0 is 1 when the section is
% unlocked, bit 1 is 1 when no waiting thread has
% been woken, and the bits above are the ones'
% complement of the number of waiting threads.
% State is locked or unlocked; Woken is woken or
% none.
raw_fields(Raw, State, Woken, Waiters) :-
    must_be(integer, Raw),
    (   Raw /\ 1 =:= 1
    ->  State = unlocked
```

```
    ;   State = locked
    ),
    (   Raw /\ 2 =:= 2
    ->  Woken = none
    ;   Woken = woken
    ),
    Waiters is (-1 - Raw) >> 2,
    (   Waiters >= 0
    ->  true
    ;   domain_error(lock_count_field, Raw)
    ).

% waiters_from_raw(+Raw, -Waiters): the number of
% waiting threads encoded in Raw; the normalized
% WaiterCount of critical_section/4 for a parser
% that reads the field under this encoding.
waiters_from_raw(Raw, Waiters) :-
    raw_fields(Raw, _, _, Waiters).

% raw_from_fields(+State, +Woken, +Waiters, -Raw):
% the inverse of raw_fields/4, for checking it.
raw_from_fields(State, Woken, Waiters, Raw) :-
    (   State == unlocked -> B0 = 1 ; B0 = 0 ),
    (   Woken == none -> B1 = 1 ; B1 = 0 ),
    Raw is ((\ Waiters) << 2) \/ (B1 << 1) \/ B0.


?- raw_fields(-1, S, K, W).
S = unlocked,
K = none,
W = 0.

?- raw_fields(-2, S, K, W).
S = locked,
K = none,
W = 0.

?- waiters_from_raw(-6, W).
W = 1.

?- waiters_from_raw(-10, W).
W = 2.

?- raw_from_fields(locked, none, 2, Raw),
   format("~2r~n", [Raw]).
-1010
Raw = -10.

?- forall(( between(0, 1000, N),
            member(S-K, [locked-none,
                         locked-woken,
                         unlocked-none]) ),
          ( raw_from_fields(S, K, N, R),
            raw_fields(R, S, K, N) )).
true.

?- catch(waiters_from_raw(0xfffffffa, W),
         error(E, _), true).
E = domain_error(lock_count_field, 4294967290).
```

The value `-1` is the unlocked section with no waiters; `-2` has bit 0 clear and is locked with no waiters; `-6` and `-10` are locked with one and two waiters, and `format/2` with `~2r` shows `-10` in binary as `-1010`, with a sign rather than as a two's complement bit pattern. As a pattern it is an unbroken run of ones ending in `0110`: bit 0 clear is locked, bit 1 set is no waiter woken, and the bits above them, ones down to `01`, are the complement of zeros down to `10`, which is two. The round trip encodes every count up to a thousand under three flag combinations and decodes it back; it is the check that `raw_fields/4` implements the encoding stated in its comment, and the encoding is the documentation's, which the comment restates. The last query is the same field `-6` read as the unsigned `0xfffffffa`, which the decoder rejects. A knowledge base that must also keep the field as displayed adds an extension relation for it rather than changing the meaning of the third argument.

Atoms that classify, such as the symbol kinds `pdb`, `export`, and `none`, the access kinds `read`, `write`, and `execute`, and the wait reasons, are lowercase atoms from a fixed set that the schema enumerates, never free text copied from the log; a parser that meets a value outside the set should fail loudly, with an error in the manner of Section 7.6, rather than record a fourth spelling.

15.5 Unknowns and Provenance

Chapter 7 found three different answers to the question of a module's vendor: a name, the atom `'Unknown'`, and the absence of any fact. They mean three different things, and the schema keeps them apart with one rule: a value the debugger reported as unknown is recorded as an explicit atom; a value the debugger was not asked for, or did not report, is recorded by the absence of a fact; and a value that is genuinely zero or empty is recorded as that value.

The first case is `'Unknown'` as a function name, which says that the debugger looked for a symbol and had none, and `'Unknown'` as a company, which says that the version resource was read and named no vendor. The explicit atom is a positive fact about the debugger's result, and a recognizer can count on it: the **No Component Symbols** (page 431) instances of Chapter 8 were found by matching the atom. The second case is the five modules of `dump1.pl` with no `company` fact at all:

```
?- module(M, _, _), \+ module_info(M, company, _).
M = ntdll ;
M = kernelbase ;
M = kernel32 ;
M = ucrtbase ;
M = msvcp140 ;
false.
```

Nothing is known about their vendors from this knowledge base, and a recognizer that needs a vendor must treat them as the `unrecorded` of Chapter 7, not as unknown. The third case is the owner `0` in `critical_section(0x7ff6_1d13_a1c0, 0, 0, 0)`: the section is unowned, which is a fact about the section, and `0` is the value the debugger displays for it; it is not a missing owner. The atom `none` for symbols is of the first kind, and the schema uses it rather than omitting the fact, because a module whose symbol status was not examined and a module with no symbols must remain

distinguishable. A fourth case arises in stack traces: a return address that lies in no loaded module, which the debugger prints as a bare address. The schema records such a frame as `frame(TID, N, none, 'Unknown', 0)`, with the module atom `none`, the function `'Unknown'`, and a zero offset, and keeps the address itself, when the knowledge base needs it, in an extension relation `frame_address(TID, N, Address)` of the kind Chapter 19 defines. The atom `none` in the module position is thus of the first kind too: the debugger looked and found no module.

The rule has a consequence for negation. Under the closed-world reading of Chapter 7, `\+ last_error(105, _)` succeeds, and it means only that the knowledge base has no last error for thread 105, which is so because `!teb` was run for six threads and not for that one:

```
?- thread(T, _), \+ last_error(T, _).
T = 105 ;
T = 106 ;
T = 107 ;
T = 108 ;
false.
```

To tell the absence of a fact from the absence of a command, the knowledge base records **provenance**: which command produced the facts of each relation, from which log. The convention is `source(Relation, Command, Log)`, with `Relation` a name and arity, `Command` an atom holding the debugger command as typed, or `derived` for a relation the parser computes from others, or `analyst` for a finding entered by hand, and `Log` the file the facts were parsed from.

```
% source(Relation, Command, Log): the facts of
% Relation were parsed from the output of Command
% in the log file Log.
source(process/2, '!analyze -v', 'dump1.log').
source(module/3, lm, 'dump1.log').
source(module_info/3, lmv, 'dump1.log').
source(thread/2, '~*kn', 'dump1.log').
source(thread_time/3, '!runaway 7', 'dump1.log').
source(thread_state/2, derived, 'dump1.log').
source(frame/5, '~*kn', 'dump1.log').
source(exception_record/3, '.exr -1',
       'dump1.log').
source(exception_info/3, '.exr -1', 'dump1.log').
source(critical_section/4, '!locks', 'dump1.log').
source(waits_for/3, '!cs', 'dump1.log').
source(handle_count/3, '!handle 0 0',
       'dump1.log').
source(heap/4, '!heap -s', 'dump1.log').
source(last_error/2, '!teb', 'dump1.log').
source(last_status/2, '!teb', 'dump1.log').
source(hooked/3, analyst, 'dump1.notes').

?- schema(R, _), \+ source(R, _, _).
false.

?- source(R, '~*kn', _).
R = thread/2 ;
```

```
R = frame/5.

?- source(R, C, _), \+ source(R, _, 'dump1.log').
R = hooked/3,
C = analyst.
```

Every relation of the schema has a source, two relations come from the one `~*kn` command, and one relation comes from somewhere other than the log. The commands are atoms and nothing more: they are quoted because a debugger command is not an atom in Prolog's own syntax, and the knowledge base does not interpret them. With `source/3` in place, the negated query about thread 105 can be qualified. A missing `last_error/2` fact means "no error was recorded" only for a thread that `!teb` was run on, and since the convention records the command per relation and not per thread, it says that the command was run, not on which threads; the per-thread record is a separate relation, `covering/2` below.

`source/3` says which command produced a relation. It does not say what the dump could have shown the command, which Section 15.1 described as a property of the capture; nor whether the command succeeded; nor, for a command run per thread, which threads it was run on. The knowledge base records all three in three extension relations, and a rule below combines them with a declared policy for each relation into a three-valued answer.

`dump_info(Key, Value)` records how the dump was written, one fact per attribute of the capture as the analyst recorded it from the debugger's description of the file when the dump was opened: the dump file; the command or tool that wrote it; whether its memory is `full` or `partial`; whether its stacks are `unfiltered` or `filtered` by the option that reduces them to what a stack walk needs; and one fact `dump_info(stream, S)` per stream the file holds, the mandatory streams among them, so that a policy can name a stream without the rule assuming it. Ranges of the address space that the debugger reported it could not read are recorded as `unavailable(Base, Size)`, the extension Chapter 21 defines; the shared dump has two. `ran(Command, Outcome)` records that the parser found the output of `Command` in the log, with `Outcome` the atom `ok` when the output parsed, or `failed(LineNo)` when the debugger printed an error message in place of its output, at that line of the log. `covering(Command, TID)` records, for a command run per thread, that the parser found the command's output for thread `TID`: `~*kn` covered every thread, and `!teb` the six the analyst inspected. The three relations are declared dynamic because the checks at the end of this section alter them for the duration of a goal. A fourth extension relation is declared with them, `last_status(TID, Status)`, for the nonzero `LastStatusValue` that `!teb` prints beside the last error: the parser records one only when it is nonzero, and none of the six inspected threads had one, so the relation is defined and empty, which is the state the checks below need to tell from a relation that is not defined at all.

```
:- dynamic dump_info/2, ran/2, covering/2.
```

```
% last_status(TID, Status): the nonzero
% LastStatusValue that !teb prints for thread TID,
% from the same output as last_error/2. An
% extension of the schema, declared dynamic: the
% six inspected threads had none, so the relation
% is defined and has no facts.
:- dynamic last_status/2.

% dump_info(Key, Value): how the dump was written,
% as the analyst recorded it. An extension of the
% schema; dump1.pl does not contain it.
dump_info(file, 'dump1.dmp').
dump_info(writer, '.dump /ma').
dump_info(memory, full).
dump_info(stacks, unfiltered).
dump_info(stream, thread_list).
dump_info(stream, module_list).
dump_info(stream, exception).
dump_info(stream, thread_info).
dump_info(stream, handle_data).
dump_info(stream, unloaded_modules).

% ran(Command, Outcome): the parser found the
% output of Command in the log; Outcome is ok when
% it parsed, or failed(LineNo) when the debugger
% printed an error in its place at that line.
ran('!analyze -v', ok).
ran(lm, ok).
ran(lmv, ok).
ran('~*kn', ok).
ran('!runaway 7', ok).
ran('.exr -1', ok).
ran('!locks', ok).
ran('!cs', ok).
ran('!handle 0 0', ok).
ran('!heap -s', ok).
ran('!teb', ok).

% covering(Command, TID): the parser found the
% output of the per-thread Command for thread TID.
covering('~*kn', 100).
covering('~*kn', 101).
covering('~*kn', 102).
covering('~*kn', 103).
covering('~*kn', 104).
covering('~*kn', 105).
covering('~*kn', 106).
covering('~*kn', 107).
covering('~*kn', 108).
covering('~*kn', 109).
covering('!teb', 100).
covering('!teb', 101).
covering('!teb', 102).
covering('!teb', 103).
covering('!teb', 104).
covering('!teb', 109).
```

The **capture policy** of a relation states what the capture must hold for the relation's facts to be complete, so that the absence of a fact may be read as evidence. `capture_policy(Relation, Needs)` declares it, one fact per relation of the schema and no relation without one. `Needs` is a list of terms from a closed vocabulary: `stream(S)`, the capture holds stream `S`; `memory(full)`, the memory was written in full; `stacks(unfiltered)`, the stacks were not filtered; `relation(R)`, the facts of relation `R` are themselves complete, for a relation computed from others; and `every_thread(Command)`, the per-thread `Command` covered every thread of `thread/2`. The empty list is a policy, and it is declared rather than defaulted: `process/2` needs nothing of the capture beyond the command having run, and neither do the two exception relations, since the exception record is written with the dump whenever there is one and no capture option governs it.

```
% need(Need): the vocabulary of capture needs.
need(stream(_)).
need(memory(_)).
need(stacks(_)).
need(relation(_)).
need(every_thread(_)).

% capture_policy(Relation, Needs): the facts of
% Relation are complete when the command of
% source/3 ran and succeeded and the capture meets
% every need in Needs. Every relation of the
% schema declares one; an empty list is a policy.
capture_policy(process/2, []).
capture_policy(module/3, [stream(module_list)]).
capture_policy(module_info/3,
               [stream(module_list),
                memory(full)]).
capture_policy(thread/2, [stream(thread_list)]).
capture_policy(thread_time/3,
               [stream(thread_list),
                stream(thread_info)]).
capture_policy(thread_state/2,
               [relation(thread/2),
                relation(exception_record/3),
                relation(frame/5)]).
capture_policy(frame/5,
               [stream(thread_list),
                every_thread('~*kn')]).
capture_policy(exception_record/3, []).
capture_policy(exception_info/3, []).
capture_policy(critical_section/4,
               [memory(full)]).
capture_policy(waits_for/3,
               [memory(full), stacks(unfiltered),
                relation(frame/5)]).
capture_policy(handle_count/3,
               [stream(handle_data)]).
capture_policy(heap/4, [memory(full)]).
capture_policy(last_error/2,
               [stream(thread_list),
                every_thread('!teb')]).
capture_policy(last_status/2,
               [stream(thread_list),
```

```
                every_thread('!teb')]).
capture_policy(hooked/3, []).
```

The policies are requirements of this pipeline, declared so that the rule below has something to check. They are not a description of the smallest dump from which the debugger could extract each relation, and a policy may be stricter than the debugger needs; what matters is that it is written down and checked. Three policies deserve comment. `frame/5` requires the thread list and the stack trace of every thread, and not unfiltered stacks, since the filter keeps what the stack walk needs. `waits_for/3` is the relation most dependent on the capture: the parser identifies a waiting thread by its frames and finds the section it waits for among the values on its stack, which a filtered stack may not keep, and `!cs` reads the section itself from the dumped memory. `hooked/3` has the empty policy and the source `analyst`: no `ran/2` fact records an inspection by hand as a command run to completion, so the rule below does not find the relation complete, and will not until the analyst records the inspection as run, which is the right answer for a finding recorded without its method.

The rule is `coverage(Relation, Status)`, and `Status` is one of three terms. `complete` says that the absence of a fact of `Relation` may be read as evidence: a policy is declared, a capture is recorded, the command of `source/3` is recorded as run with outcome `ok`, and every need of the policy is met. `incomplete(Missing)` says that the records are there but fall short, and `Missing` lists what is lacking: a stream, the memory, a thread the per-thread command did not cover, a relation the derived relation depends on, or the command's failure. `unknown` says that the question cannot be answered from the knowledge base, because the relation has no policy, or no capture is recorded, or the command is not recorded as run; it is never collapsed into `complete`. `thread_coverage(Relation, TID, Status)` asks the same question of the facts about one thread, for which a per-thread command need only have covered that thread.

```
% missing(Need, Scope, What): the recorded capture
% does not meet Need within Scope, which is all or
% thread(TID), and What names what is missing. A
% need outside the vocabulary raises an error.
missing(stream(S), _, stream(S)) :-
    \+ dump_info(stream, S).
missing(memory(M), _, memory(M)) :-
    \+ dump_info(memory, M).
missing(stacks(K), _, stacks(K)) :-
    \+ dump_info(stacks, K).
missing(relation(R), Scope,
        relation(R, Status)) :-
    coverage(R, Scope, Status),
    Status \== complete.
missing(every_thread(C), all, covering(C, T)) :-
    thread(T, _),
    \+ covering(C, T).
missing(every_thread(C), thread(T),
        covering(C, T)) :-
    \+ covering(C, T).
missing(Need, _, _) :-
    \+ need(Need),
    domain_error(capture_need, Need).
```

```
% coverage(Relation, Status): Status is complete
% when the absence of a fact of Relation may be
% read as evidence, incomplete(Missing) when the
% recorded capture or command run falls short in
% the ways Missing lists, and unknown when the
% policy, the capture, or the command's run is
% not recorded.
coverage(Relation, Status) :-
    coverage(Relation, all, Status).

% thread_coverage(Relation, TID, Status): as
% coverage/2, for the facts of Relation about
% thread TID alone.
thread_coverage(Relation, TID, Status) :-
    coverage(Relation, thread(TID), Status).

coverage(Relation, Scope, Status) :-
    (   capture_policy(Relation, Needs),
        dump_info(_, _),
        source(Relation, Command, _),
        (   Command == derived
        ->  Outcome = ok
        ;   ran(Command, Outcome)
        )
    ->  findall(What,
                ( member(Need, Needs),
                  missing(Need, Scope, What) ),
                Unmet),
        (   Outcome == ok
        ->  Missing = Unmet
        ;   Missing = [ran(Command, Outcome)
                       | Unmet]
        ),
        (   Missing == []
        ->  Status = complete
        ;   Status = incomplete(Missing)
        )
    ;   Status = unknown
    ).
```

`missing/3` has one clause per kind of need and succeeds when the need is unmet within the scope, binding its third argument to the particular lack: a per-thread need unmet over the whole dump yields one `covering(Command, TID)` term per uncovered thread, so that `Missing` names the threads. Its last clause raises a domain error for a need outside the vocabulary, so that a misspelled need in a policy is an error and not a need silently met, by the rule of Section 15.4 for classifying atoms. `coverage/3` is one if-then-else. The condition gathers the records: the policy, any capture fact, the source, and the run of the command, where a derived relation, which has no command, counts as run. The then-branch collects the unmet needs with `findall/3` and puts the command's failure, if it had one, in front of them; the else-branch is `unknown`. A relation computed from others asks for their coverage through the `relation/1` need, so that `thread_state/2` is complete only when the three relations it is computed from are.

```
?- set_prolog_flag(toplevel_print_anon, false).
true.
```

```
?- coverage(handle_count/3, S).
S = complete.

?- coverage(thread_state/2, S).
S = complete.

?- coverage(hooked/3, S).
S = unknown.

?- coverage(last_error/2, incomplete(_Missing)),
   forall(member(M, _Missing), (print(M), nl)).
covering('!teb',105)
covering('!teb',106)
covering('!teb',107)
covering('!teb',108)
true.

?- thread_coverage(last_error/2, 109, S).
S = complete.

?- thread_coverage(last_error/2, 105, S).
S = incomplete([covering('!teb', 105)]).

?- forall(schema(R, _),
   ( coverage(R, S), functor(S, Name, _),
     format("~w~t~20|~w~n", [R, Name]) )).
process/2           complete
module/3            complete
module_info/3       complete
thread/2            complete
thread_time/3       complete
thread_state/2      complete
frame/5             complete
exception_record/3  complete
exception_info/3    complete
critical_section/4  complete
waits_for/3         complete
handle_count/3      complete
heap/4              complete
last_error/2        incomplete
hooked/3            unknown
true.
```

The flag `toplevel_print_anon` is set as in Chapter 5, so that the list bound to `_Missing` is printed one element per line by `print/1` and not again by the toplevel. Thirteen relations are complete, one is incomplete, and one is unknown. The incompleteness of `last_error/2` is the answer to the negated query about thread 105 earlier in this section: `\+ last_error(105, _)` is true, and it is not evidence, because `!teb` was not run on thread 105. For thread 109 the same question has complete coverage, and a missing fact there would be evidence. `thread_absence/3` states this as a three-valued answer about the facts of one thread; it is the per-thread form of the `absence/2` that Chapter 17 defines for relations.

```
% thread_absence(Relation, TID, Status): Status is
% present when Relation is defined and has a fact
% whose first argument is TID, absent when it has
% none and the coverage of Relation for TID is
```

```
% complete, and unknown otherwise. A relation
% that is not defined has no fact; an error raised
% by one that is defined is not caught.
thread_absence(F/A, TID, Status) :-
    (   current_predicate(F/A),
        functor(Goal, F, A),
        arg(1, Goal, TID),
        call(Goal)
    ->  Status = present
    ;   thread_coverage(F/A, TID, complete)
    ->  Status = absent
    ;   Status = unknown
    ).

?- thread_absence(last_error/2, 109, S).
S = present.

?- thread_absence(last_error/2, 105, S).
S = unknown.

?- thread_absence(unrecorded/2, 105, S).
S = unknown.
```

`current_predicate/1` guards the call, as it does in the `absence/2` of Chapter 17: under SWI-Prolog's default `unknown` flag, calling a relation that is not defined, because it was misspelled or because the facts file omitted it, raises an existence error, and the rule would raise it before `thread_coverage/3` could answer. The guard tests existence and nothing more, so that a relation that is defined and raises an error of its own, a derived relation with a defect in its rule, say, still raises it; converting every exception into an absence would hide the defect behind a three-valued answer. The third query is a relation with no definition and no policy, which is unknown, the answer the model gives to any relation the knowledge base cannot speak for. The checks below separate the other three answers on a relation that is defined and empty.

The following are checks of boundary cases, each run on a knowledge base altered for the duration of one goal and restored afterwards. `without_facts/2` retracts every fact matching a pattern, calls the goal once, and restores the facts under `setup_call_cleanup/3` of Chapter 7, so that a goal that fails or raises cannot leave the knowledge base altered; `with_fact/2` adds one fact the same way. Both use the `assertz/1`, `retract/1`, and `retractall/1` of Chapter 11 on the three dynamic relations.

```
% without_facts(Pattern, Goal): call Goal once
% with every fact matching Pattern retracted; the
% facts are restored afterwards, whether Goal
% succeeds, fails, or raises.
without_facts(Pattern, Goal) :-
    findall(Pattern, Pattern, Facts),
    setup_call_cleanup(
        retractall(Pattern),
        once(Goal),
        maplist(assertz, Facts)).
```

```
% with_fact(Fact, Goal): call Goal once with Fact
% added; Fact is retracted afterwards.
with_fact(Fact, Goal) :-
    setup_call_cleanup(
        assertz(Fact),
        once(Goal),
        retract(Fact)).

?- without_facts(dump_info(stream, handle_data),
                 coverage(handle_count/3, S)).
S = incomplete([stream(handle_data)]).

?- without_facts(dump_info(_, _),
                 coverage(frame/5, S)).
S = unknown.

?- without_facts(dump_info(_, _),
                 coverage(exception_record/3, S)).
S = unknown.

?- without_facts(ran('!handle 0 0', _),
                 coverage(handle_count/3, S)).
S = unknown.

?- without_facts(ran('!heap -s', _),
       with_fact(ran('!heap -s', failed(812)),
                 coverage(heap/4, S))).
S = incomplete([ran('!heap -s', failed(812))]).

?- without_facts(covering('~*kn', 107),
                 coverage(thread_state/2, _S)),
   _S = incomplete([relation(R, incomplete(M))]).
R = frame/5,
M = [covering('~*kn', 107)].

?- with_fact(covering('!teb', 105),
       thread_absence(last_error/2, 105, S)).
S = absent.

?- thread_absence(last_status/2, 105, S).
S = unknown.

?- thread_absence(last_status/2, 109, S).
S = absent.

?- with_fact(last_status(109, 0xc0000034),
       thread_absence(last_status/2, 109, S)).
S = present.

?- coverage(handle_count/3, complete),
   coverage(heap/4, complete),
   thread_absence(last_error/2, 105, unknown),
   \+ last_status(_, _),
   aggregate_all(count, dump_info(_, _), N).
N = 10.
```

The first check is a capture without the handle data stream: the handle summary is incomplete, and `Missing` names the stream. The second and third are a knowledge base with no capture record at

all, the case of a facts file consulted without its `dump_info/2` facts: the coverage of `frame/5` and of `exception_record/3`, two relations whose policies ask nothing of the optional streams, is unknown, because nothing records what the dump held. A rule that read an empty or satisfied requirement list as completeness would answer `complete` here, and that is the answer this rule exists to exclude. The fourth is a command not recorded as run, which is unknown, not incomplete: the knowledge base cannot say what the command would have shown. The fifth is a command recorded as having failed, which is incomplete with the failure in `Missing`. The sixth removes the stack trace of thread 107 from the coverage of `~*kn`, and the incompleteness propagates from `frame/5` to the derived `thread_state/2`, with the dependency spelled out in the one term of `Missing`, which the query takes apart. The seventh adds coverage of thread 105 by `!teb`, and the missing last error for that thread becomes evidence: absent. The next three are the per-thread answers on `last_status/2`, a relation that is defined and has no facts: for thread 105, which `!teb` did not cover, the absence is unknown; for thread 109, which it covered, the absence is a fact about the dump, absent; and with a fact for thread 109 added, present. Together with the undefined relation above, these are the four answers the guard must keep apart. The last query shows the knowledge base restored, with its ten capture facts, the empty relation, and the earlier answers unchanged.

The same records precede any use of a count disagreement. Chapter 20 reports a critical section whose `WaiterCount` differs from the number of threads waiting for it, and the disagreement measures the dump only when both relations are complete, as `coverage/2` says, so that the capture held both the section and every waiter's stack, and when the address that joins them was normalized to the same integer on both sides; otherwise it measures the pipeline. Chapter 17 uses `coverage/2` with `source/3` to state when the absence of a fact is evidence, and Chapter 29 uses provenance to attach a source to every step of an explanation.

15.6 Kernel and Complete Memory Dumps

A **kernel memory dump** contains the kernel's address space and the structures that describe every process and thread in the system; a **complete memory dump** contains all of physical memory and therefore the user-mode address spaces of every process as well. Both add a dimension the schema of Section 15.2 does not have: a thread belongs to a process, and a module is loaded into a process at a base that may differ from its base in another.

The extension that Part III discusses, and that this book does not use, prefixes the process identifier to the relations whose arguments are per process. `frame/6` is `frame(PID, TID, N, Module, Function, Offset)`, and `module/4` is `module(PID, Name, Base, Size)`; `thread/2` is unchanged, since it already carries the process, and `process/2` gains the facts for every process in the system. The following is illustrative and is not consulted:

```
% frame(PID, TID, N, Module, Function, Offset)
frame(4120, 109, 0, dbclient, 'Buffer::Copy',
      0x1a).

% module(PID, Name, Base, Size)
module(4120, dbclient, 0x7ffb_8d10_0000, 0x6e000).
```

The prefix on `frame/6` is a convenience rather than a necessity. Windows allocates process and thread identifiers from one system-wide table, so a thread identifier names one thread in the whole system while that thread exists, and `thread(TID, PID)` suffices to find its process; the prefix saves the join and lets first-argument indexing select a process's frames at once. The prefix on `module/4` is a necessity, because the same module can be at different bases in different processes and the containment test of Chapter 6 must know which process an address belongs to. A thread in a kernel dump also has frames in kernel mode, in the module `nt` and in drivers, above its user-mode frames when the dump caught it in a system call; the schema records them as frames like any other, and the module name tells them apart.

Managed-code frames are the other case the schema accommodates without using it. A frame compiled at run time by a managed runtime belongs to no module, and the debugger, with the runtime's extension loaded, prints a method name in place of the module and function. This book's convention for such a frame is the module atom `managed`, the method name as the function, and the offset as the extension shows it. The convention is stated so that a knowledge base with such frames can be built; the shared dump has none, and the pattern Stack Trace Collection (Managed Space) is not treated in this edition.

15.7 The Shared Dump

`dump1.pl` is listed in full in Appendix B. This section walks through its structure and the facts that the rest of the book relies on, so that the reader knows what is there without the listing.

The file begins with the directive `:- discontiguous frame/5.`, because its frames are grouped by thread with a comment line before each group, and SWI-Prolog warns, as Chapter 1 showed, when the clauses of a predicate are not contiguous unless told that they are not meant to be. The relations follow in the order of the schema table. One process, ten modules, twenty module attributes, ten threads with their times and states, 78 frames, one exception record with two parameters, three critical sections, three waits, six handle types, three heaps, six last errors, and two hooks, as `fact_count/2` showed.

`listing/1` prints a loaded relation as `portray_clause/1` would, in decimal:

```
?- listing(waits_for/3).
waits_for(101, critical_section, 140695026507968).
waits_for(103, critical_section, 140695026508096).
waits_for(104, critical_section, 140695026507968).

true.

?- listing(critical_section/4).
critical_section(140695026507968, 103, 2, 1).
critical_section(140695026508096, 104, 1, 1).
critical_section(140695026508224, 0, 0, 0).

true.
```

The two addresses are `0x7ff6_1d13_a0c0` and `0x7ff6_1d13_a140` in the file. Thread 103 owns the first and waits for the second; thread 104 owns the second and waits for the first; thread 101 waits for the first behind 104. These six facts are the deadlock and the wait chain that Chapters 4, 9, and 11 approached and that Chapter 20 treats, and the third section, unowned, is there so that a predicate over critical sections has a negative case.

The exception record names thread 109, the access violation code, and an address in `dbclient`, with the parameters saying a read of address `0x18`:

```
?- exception_record(T, Code, Addr),
   format("~16r at ~16r~n", [Code, Addr]).
c0000005 at 7ffb8d1031aa
T = 109,
Code = 3221225477,
Addr = 140718380167594.

?- exception_info(109, K, V).
K = access,
V = read ;
K = target,
V = 24.
```

The faulting thread's top frame is `dbclient!Buffer::Copy+0x1a`, in a module with export symbols only, which is the caveat Chapter 8 attached to that name. The state facts agree with the record, `thread_state(109, waiting(exception))`, and the last error of the same thread is `0x3e6`.

The module attributes show the three degrees of symbol information and the two kinds of vendor record:

```
?- module_info(M, symbols, S), S \== pdb.
M = dbclient,
S = export ;
M = cachelib,
S = none ;
M = thirdparty_hook,
S = none.

?- module_info(M, version, V),
   module_info(M, company, 'Example Corp').
M = appserver,
V = '3.2.1.0' ;
M = reportengine,
V = '3.2.1.0' ;
M = dbclient,
V = '3.1.0.7' ;
false.
```

Three modules from one vendor, two at one version and one at another, which Chapter 22 treats under **Not My Version** (page 429); two modules without symbols, one of which, `thirdparty_hook`, has no vendor and hooks two functions of `kernelbase`, which Chapter 22 treats under **Hooked Functions (User Space)** (page 433) and **Unknown Component** (page 430).

The remaining relations carry single notable values. Among the handle counts, `event` is 12,040 against a few dozen of every other type; among the heaps, the first has 1,835,008 KB committed of 2,097,152 KB reserved; among the thread times, 102 and 107 together account for 97 percent of the processor time. Each is a candidate instance for Chapter 21 or Chapter 18, and each is a number in the knowledge base that a threshold in Chapter 17 will judge.

A knowledge base built by the pipeline of Section 15.3 should satisfy the invariants of its schema before any analysis is run. The following states the ones that `dump1.pl` is built to satisfy, and `check_schema/0` reports any that fail:

```
% invariant(Name): the named invariant of the
% schema holds in the loaded knowledge base.
invariant(frame_module_loaded) :-
    forall(frame(_, _, M, _, _), module(M, _, _)).
invariant(thread_has_time) :-
    forall(thread(T, _), thread_time(T, _, _)).
invariant(thread_has_state) :-
    forall(thread(T, _), thread_state(T, _)).
invariant(thread_has_top_frame) :-
    forall(thread(T, _), frame(T, 0, _, _, _)).
invariant(frames_contiguous) :-
    forall(( frame(T, N, _, _, _), N > 0 ),
           ( N0 is N - 1,
             frame(T, N0, _, _, _)
           )).
invariant(wait_target_listed) :-
    forall(waits_for(_, critical_section, A),
           critical_section(A, _, _, _)).
invariant(owner_is_thread) :-
    forall(( critical_section(_, O, _, _),
             O =\= 0
           ),
           thread(O, _)).
invariant(exception_thread_exists) :-
    forall(exception_record(T, _, _),
           thread(T, _)).

% check_schema: print the name of every invariant
% that does not hold.
check_schema :-
    forall(( clause(invariant(Name), _),
             \+ invariant(Name)
           ),
           format("violated: ~w~n", [Name])).

?- check_schema.
true.

?- aggregate_all(count, clause(invariant(_), _),
   N).
N = 8.
```

`clause/2` from Chapter 11 enumerates the heads of `invariant/1` so that the checker need not list them twice. All eight hold. A parser that produces a knowledge base failing `frames_contiguous` has

dropped a line, which the writer of Section 15.3 cannot do, since it raises an error for a line it cannot parse and writes nothing; one failing `frame_module_loaded` has normalized a module name in one place and not another; and one failing `wait_target_listed` has a `waits_for/3` fact whose critical section `!locks` did not list, which is either a parsing error or a fact about the dump that Chapter 20 will want to know about, once `coverage/2` of Section 15.5 has said that the capture could have shown the section. The checker is run once after `consult/1` and before the recognizers of Part III, and a knowledge base that passes it is one the rest of the book can be applied to.

Chapter 16. Software Traces and Logs as Knowledge Bases

The second artifact of this book is the software trace, and `trace1.pl` has represented one since Chapter 2 as facts of a single predicate, `msg/6`. This chapter fixes that predicate as the schema of the trace knowledge base: what a message is, which of its attributes the schema records and in which argument, why the ordinal number of a message rather than its timestamp is the primary key, and how a knowledge base is loaded from the two forms in which traces arrive, text logs and comma-separated files. It then defines the derived relations that Part IV builds on, measures what indexing and memory a trace of a given size needs, and states the convention for holding several traces in one knowledge base.

As in Chapter 15, the chapter is definitional, and it keeps the same epistemic line. A trace is what a program wrote about itself while it ran, through whatever logging statements its authors placed and whatever buffering and clocks its logging library used. The knowledge base records what was written, in the order it was written, with the timestamps that were written; what happened between the messages, and whether the clock was right, are inferences the analyst makes, and Part IV makes them with predicates whose assumptions are stated.

16.1 Messages and Traces

A **software trace** is a sequence of messages written by a running system, and a **log** is a software trace kept in a file. This book uses the two words as the catalog does, trace for the artifact as analyzed and log for the file as found, and treats them as the same thing under two names. A **message** is the unit of a trace: one record, written by one logging statement at one moment, carrying attributes the logging library adds and a text the statement supplies.

The attributes vary with the library, but six are present in nearly every trace an analyst meets, and they are the six arguments of the schema. The **ordinal number** of a message is its position in the trace, counting from 1; a library rarely writes it, and the parser assigns it from the order of the lines. The **timestamp** is the time the library read from a clock when it wrote the message. The process and thread identifiers say which thread of which process executed the logging statement. The **component** is the name the statement gives for the part of the system it belongs to, a module, a subsystem, a class, or a logger name, as the library's conventions have it. The text is what the statement wrote: a sentence with values substituted into it, in the vocabulary of the programmer who wrote the statement.

What the trace does not contain is as important as what it does. It has no record of the statements that were not executed, so an activity that logs nothing is invisible, and Part IV names the patterns of that invisibility. It has no record of what happened between two messages, only of the two. Its timestamps come from the writer's clock, with the writer's resolution, and two messages with the same timestamp were not written at the same instant but within one tick; two traces from two machines have two clocks. Its ordinal numbers are reliable only as far as the library wrote messages in the order the statements executed, which a buffered library writing from several threads need

not do. The schema records the attributes; the patterns of Part IV say what can be concluded from them under which assumptions.

16.2 The msg/6 Schema

The **schema** of a trace knowledge base is one predicate, `msg(N, Time, PID, TID, Component, Text)`:

Argument	Type	Meaning
N	integer	ordinal number, from 1
Time	float	seconds from the first message
PID	integer	process identifier
TID	integer	thread identifier
Component	atom	source component
Text	string	the message text

The following clauses are illustrative copies of three facts of `trace1.pl`:

```
msg(1,  0.000, 4120, 100, appserver,
    "Service starting, version 3.2.1.0").
msg(49, 9.003, 4120, 108, appserver,
    "Housekeeping: worker 103 wait 7001 ms").
msg(50, 9.003, 4120, 108, appserver,
    "Housekeeping: worker 104 wait 6900 ms").
```

The ordinal number is the first argument and the key. It is the identity of a message: the forty-ninth message is the one written forty-ninth, whatever its timestamp, and a predicate that refers to a message refers to it by number. It is the order of the trace: the messages in order of their numbers are the trace as written, which is the order the catalog's **Event Sequence Order** (page 603) is about, and the next message after `N` is `N + 1`, a fact that the derived relations of Section 16.6 use and that no other argument provides. And it is the argument SWI-Prolog indexes first, so that a lookup by number is a hash lookup rather than a scan, as Section 16.7 shows.

The timestamp is second, and secondary, for reasons the two illustrative facts above display. Messages 49 and 50 have the same timestamp, 9.003, as do messages 7 and 8, and 9 and 10, in the shared trace: the clock's resolution is a millisecond and the housekeeping thread wrote two messages within one tick. Time does not order them; their numbers do. A timestamp can also be coarser than a millisecond, can come from a clock that was adjusted during the run, and in a knowledge base holding traces from several machines comes from several clocks, so that equal timestamps are not simultaneous and increasing numbers are not increasing times across traces. What the timestamp is for is measurement: the time between two messages is a difference of two timestamps, and the **Time Delta** (page 511) and **Discontinuity** (page 513) patterns of Chapter 25 are about such differences. The schema records it in seconds from the first message of the trace, as a float, so that a difference is one subtraction; Section 16.3 shows the conversion from a clock time, and Section 16.6 shows the floating-point care the representation requires.

The process and thread identifiers are integers, as in the dump schema, and for the same reason: `trace1.pl` and `dump1.pl` describe the same process, and the join between `msg(_, _, _, 109, _, _)` and `thread(109, 4120)` is a unification of integers. The component is an atom because it is drawn from a small fixed set, is compared by identity, and is used as a key for partitioning the trace, which is what the catalog's **Trace Partition** (page 489) does by component; it is lowercase by the same normalization rule as module names in Chapter 15, and in the shared trace the component names coincide with the module names of the dump. The text is a **string**, the SWI-Prolog string object of Chapter 2, which ISO Prolog does not have: it is compact, it is not entered in the atom table as an atom would be, so that a million distinct texts do not become a million atoms, and the predicates of Chapter 8, `sub_string/5`, `split_string/4`, and `string_code/3`, apply to it directly.

Nothing else is recorded. A severity level, a logger name distinct from the component, a source file and line, a request identifier: all occur in real traces, and a knowledge base that has them adds a relation keyed by the ordinal number, `msg_attr(N, Key, Value)`, in the manner of `module_info/3`, rather than widening `msg/6`. The six arguments are the ones every chapter of Part IV needs, and a wider predicate would make every pattern's clause longer for an attribute most traces lack.

16.3 Loading a Text Log

A text log has one message per line in a format fixed by the logging library, and the parser is a grammar for that format. Chapter 12 gives a grammar for lines of the form `2026-10-03 12:00:01.234 [4120:0101] appserver: Request 1 received`, with a date, a clock time with milliseconds, the process and thread identifiers in brackets, the component, a colon, and the text. The following four lines are the messages 7 to 10 of the shared trace in that format:

```
2026-10-03 12:00:00.215 [4120:0101] appserver: Worker 101 ready
2026-10-03 12:00:00.215 [4120:0102] appserver: Worker 102 ready
2026-10-03 12:00:00.216 [4120:0103] appserver: Worker 103 ready
2026-10-03 12:00:00.216 [4120:0104] appserver: Worker 104 ready
```

The grammar reads the clock time into an integer number of milliseconds since midnight rather than a float of seconds, and that choice is deliberate. `12:00:00.215` is 43,200,215 milliseconds, an integer that the arithmetic of Chapter 6 handles exactly; converting each timestamp to a float and subtracting the first would introduce rounding at each step, and `43200.215 - 43200.0` evaluates to `0.21499999999650754`, not to the float `0.215`. Integers are subtracted, and one division produces the float the schema wants.

```
:- use_module(library(dcg/basics)).
```

```
% log_line(Ms, PID, TID, Component, Text)//: one
% line of the log format; Ms is the clock time in
% milliseconds since midnight.
log_line(Ms, PID, TID, Component, Text) -->
    date, " ", clock(Ms), " [",
    integer(PID), ":", integer(TID), "] ",
    string_without(":", CC), ": ",
    remainder(TC),
    { atom_codes(Component, CC),
      string_codes(Text, TC) }.

% date//: a date, read and discarded.
date -->
    digits(_), "-", digits(_), "-", digits(_).

% clock(Ms)//: hh:mm:ss.fff as milliseconds.
clock(Ms) -->
    integer(H), ":", integer(M), ":", integer(S),
    ".", digits(FC),
    { number_codes(F, FC),
      Ms is ((H * 60 + M) * 60 + S) * 1000 + F }.
```

`integer//1` from `library(dcg/basics)` reads an optionally signed run of digits as an integer, so `0101` becomes 101; `digits//1` reads the digits as codes, which is right for the date, which is discarded, and for the fraction, whose three digits are milliseconds by the format's definition. `string_without//2` reads the component up to the colon, and `remainder//1` takes the rest of the line as the text. The component becomes an atom and the text a string, as the schema prescribes, in the `{}` goals at the end.

The log lines are too wide for a program block of this book, so the following assembles them from parts with `atomics_to_string/2`; a parser for a real log reads them with `phrase_from_file/2` over the whole file or with `read_line_to_string/2` line by line, as Chapter 13 showed, and strips the line terminator before `remainder//1` sees it.

```
% log_parts(Parts): the pieces of one log line.
log_parts(["2026-10-03 12:00:00.215 [4120:0101] ",
           "appserver: ", "Worker 101 ready"]).
log_parts(["2026-10-03 12:00:00.215 [4120:0102] ",
           "appserver: ", "Worker 102 ready"]).
log_parts(["2026-10-03 12:00:00.216 [4120:0103] ",
           "appserver: ", "Worker 103 ready"]).
log_parts(["2026-10-03 12:00:00.216 [4120:0104] ",
           "appserver: ", "Worker 104 ready"]).

% log_line(Line): Line is one line of the sample
% log, as a string.
log_line(Line) :-
    log_parts(Parts),
    atomics_to_string(Parts, Line).

% parse_line(Line, M): M is the m/5 term for Line.
parse_line(Line, m(Ms, PID, TID, C, Text)) :-
    string_codes(Line, Codes),
    phrase(log_line(Ms, PID, TID, C, Text),
           Codes).
```

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, log_line(_), N).
N = 4.

?- forall(log_line(L),
   ( parse_line(L, M), print(M), nl )).
m(43200215,4120,101,appserver,"Worker 101 ready")
m(43200215,4120,102,appserver,"Worker 102 ready")
m(43200216,4120,103,appserver,"Worker 103 ready")
m(43200216,4120,104,appserver,"Worker 104 ready")
true.
```

The flag `toplevel_print_anon` is set as in Chapter 5 so that variables whose names begin with an underscore are not printed. The parsed terms are printed by `print/1`, which quotes where needed and writes no spaces after commas, so that they fit the page; the lines themselves, 63 characters each, do not.

The intermediate term `m/5` holds the parsed attributes without a number and without a relative time, because neither is a property of one line: the number is the line's position, and the time is relative to the first message. `log_messages/2` supplies both. `maplist/3` parses every line; the first parsed term gives the base time; and `foldl/6` from Chapter 10 walks the parsed terms with a counter, producing one `msg/6` term for each.

```
% log_messages(Lines, Msgs): Msgs are the msg/6
% terms for the log lines Lines, numbered from 1,
% with Time in seconds from the first message.
log_messages(Lines, Msgs) :-
    maplist(parse_line, Lines, Parsed),
    Parsed = [m(Ms0, _, _, _, _) | _],
    foldl(number_message(Ms0), Parsed, Msgs,
          1, _).

% number_message(Ms0, M, Msg, N, N1): Msg is the
% msg/6 term numbered N for the m/5 term M, with
% Time relative to Ms0; N1 is the next number.
number_message(Ms0, m(Ms, PID, TID, C, Text), Msg,
               N, N1) :-
    Time is (Ms - Ms0) / 1000.0,
    Msg = msg(N, Time, PID, TID, C, Text),
    N1 is N + 1.

% write_facts(File, Facts): write Facts to File as
% clauses, one per line.
write_facts(File, Facts) :-
    setup_call_cleanup(
        open(File, write, S),
        maplist(portray_clause(S), Facts),
        close(S)).
```

```
?- findall(L, log_line(L), _Ls),
   log_messages(_Ls, _Msgs),
   forall(member(M, _Msgs), ( print(M), nl )).
msg(1,0.0,4120,101,appserver,"Worker 101 ready")
msg(2,0.0,4120,102,appserver,"Worker 102 ready")
msg(3,0.001,4120,103,appserver,"Worker 103 ready")
msg(4,0.001,4120,104,appserver,"Worker 104 ready")
true.

?- findall(L, log_line(L), _Ls),
   log_messages(_Ls, _Msgs),
   write_facts('sample.pl', _Msgs).
true.

?- [sample].
true.

?- msg(3, T, P, TID, C, Text).
T = 0.001,
P = 4120,
TID = 103,
C = appserver,
Text = "Worker 103 ready".

?- msg(N, 0.0, _, _, _, _).
N = 1 ;
N = 2.
```

The four lines become four facts, numbered by position, with times 0.0, 0.0, 0.001, and 0.001 relative to the first. `write_facts/2` writes them with `portray_clause/2`, one clause per line with the strings in double quotes, so that `consult/1` reads them back as strings under SWI-Prolog's default `double_quotes` flag. The division by `1000.0` produces a float even for the first message, whose difference is the integer 0, so that `Time` is a float in every fact; the last query shows that the float `0.0` unifies with the stored times, where the integer `0` would not, since `0` and `0.0` are different terms that compare equal only under arithmetic comparison. The written file is consulted with the list shorthand of Chapter 1, and the knowledge base now has an `msg/6` relation of four facts, the shape of `trace1.pl` on a smaller trace.

Two details of real logs are left to the parser's author. The first is the day boundary: a clock time in milliseconds since midnight goes backwards at midnight, and a log that crosses it needs the date, which `date//0` discards here, folded into the time. The second is the base: this book sets the first message of the trace at time zero, which is right for one trace and wrong for several, and Section 16.8 returns to it.

16.4 The Shared Trace

`trace1.pl` is listed in full in Appendix B. It is 71 facts of `msg/6`, one process, seven threads, and four components, covering fourteen seconds of the server whose dump is `dump1.pl`. The following starts a fresh toplevel and consults it.

```
:- [trace1].

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, msg(_, _, _, _, _, _), N).
N = 71.

?- msg(1, T, P, TID, C, Text).
T = 0.0,
P = 4120,
TID = 100,
C = appserver,
Text = "Service starting, version 3.2.1.0".

?- msg(71, T, _, TID, _, Text).
T = 14.01,
TID = 100,
Text = "Service stopping".

?- aggregate_all(set(TID),
   msg(_, _, _, TID, _, _), TIDs).
TIDs = [100, 101, 102, 103, 104, 108, 109].

?- aggregate_all(count,
   distinct(C, msg(_, _, _, _, C, _)), N).
N = 4.
```

The first and last messages bracket the trace: a service starting at time zero and stopping at 14.01 seconds, both on thread 100, the main thread of the dump. The seven threads are seven of the dump's ten; the three absent, 105, 106, and 107, wrote nothing, which is consistent with two of them being idle pool threads and tells nothing about the third, the thread with the deep recursion, except that its code does not log. The four components are four of the dump's modules.

The structure of the trace, as the rest of the book relies on it, is this. Messages 1 to 10 are startup: configuration, cache, database connection, and four workers reporting ready. Messages 11 to 17 are the lifecycle of request 1 on worker 101, from receipt to completion in 309 ms. Request 2 is dispatched to worker 102 at message 19 and never completes; from message 31 to 36 that worker writes a warning every half second about a table that keeps doubling. Requests 3 and 4 go to workers 103 and 104 at messages 24 and 28 and stall, which is the deadlock of the dump seen from the trace. Requests 5 to 9 arrive and are queued, with three error messages about the queue depth; the housekeeping thread, 108, reports at messages 48 to 52 how long each worker has been waiting; worker 101 then drains the queue in messages 53 to 66, with completion times falling from 4300 ms to 309 ms. There is no message between 9.309 and 14.000 seconds. Thread 109 then receives request 10, fails in the database client with the error code of the dump's last error, and reports the unhandled exception whose code the dump's exception record has; the service stops.

```
?- msg(N, _, _, 109, _, Text),
   format("~d ~s~n", [N, Text]), fail.
67 Request 10 received: GET /session/9/data
68 Receive start: session=71
69 Error: buffer copy failed, code=0x3e6
70 Unhandled exception 0xc0000005, thread 109
false.

?- msg(67, T, _, _, _, _), format("~3f~n", [T]).
14.000
T = 14.0.
```

The four messages of thread 109, two from the application server and two from the database client, written within three milliseconds of each other at the fourteenth second, are the trace's account of the thread whose stack trace, exception record, and last error the dump holds, and Chapter 28 puts the two accounts side by side under **Small DA+TA** (page 622). The `~3f` directive prints a time with three decimals, which is the clock's resolution, where the toplevel prints the float as `14.0`; Section 16.6 says why the distinction matters.

16.5 Loading and Saving CSV

Traces also arrive as comma-separated files, exported from a log viewer or a database, and `library(csv)` reads and writes them. `csv_read_file/2` reads a file into a list of `row/N` terms, one per line, with the fields as arguments; `csv_write_file/2` writes such a list, quoting fields that contain commas or quotes. The round trip from the shared facts through a file and back shows both and the conversions between the row shape and the schema.

```
% message_row(Row): Row is a row/6 term for a
% message of the trace.
message_row(row(N, T, P, TID, C, Text)) :-
    msg(N, T, P, TID, C, Text).

% row_message(Row, Msg): Msg is the msg/6 term for
% the row/6 term Row read from a CSV file.
row_message(row(N, T, P, TID, C, Text0), Msg) :-
    atom_string(Text0, Text),
    Msg = msg(N, T, P, TID, C, Text).

?- findall(R, message_row(R), _Rows),
   csv_write_file('trace1.csv', _Rows).
true.

?- read_file_to_string('trace1.csv', _S, []),
   split_string(_S, "\n", "\r", _Lines),
   nth1(7, _Lines, L7), nth1(14, _Lines, L14),
   format("~s~n~s~n", [L7, L14]).
7,0.215,4120,101,appserver,Worker 101 ready
14,1.048,4120,101,<...>,"Query end: 1 row, 44 ms"
L7 = <...>,
L14 = <...>.
```

```
?- csv_read_file('trace1.csv', _Rows),
   length(_Rows, Len),
   nth1(14, _Rows, row(N, T, P, TID, C, Text)).
Len = 71,
N = 14,
T = 1.048,
P = 4120,
TID = 101,
C = dbclient,
Text = 'Query end: 1 row, 44 ms'.

?- csv_read_file('trace1.csv', _Rows),
   forall(member(R, _Rows),
          ( row_message(R, M), call(M) )).
true.
```

The written file has one line per message, with Windows line endings, which `split_string/4` strips here, and with the text quoted only where it contains a comma: line 7 is unquoted and line 14, whose component is elided from the transcript for width, is quoted. Reading it back gives 71 rows whose numeric fields are numbers again, because `csv_read_file/2` converts a field that reads as a number by default, and whose text fields are atoms, because that is what the library produces for text; `row_message/2` converts the atom to the string the schema requires, and the last query confirms that every row, converted, is a fact of the loaded trace.

The default conversion is a hazard to know about. A message text consisting of digits alone, `"200"`, would come back as the integer 200, and a text such as `"1e5"` as a float. The option `convert(false)` to `csv_read_file/3` turns the conversion off and gives every field as an atom, after which the parser converts the four numeric fields itself with `atom_number/2` and leaves the text alone:

```
?- csv_read_file('trace1.csv', _Rows,
   [convert(false)]),
   nth1(14, _Rows, row(N, T, _, _, _, Text)).
N = '14',
T = '1.048',
Text = 'Query end: 1 row, 44 ms'.
```

The rule from Chapter 15 applies: the type of a field is fixed by its position in the schema, not by what its content happens to look like, and a loader that lets the content decide will one day turn a message into a number. Beyond the round trip, a CSV loader for a real export does what the log parser did: assigns ordinal numbers from row order when the file has none, converts a clock column to seconds from the first message, normalizes the component to a lowercase atom, and writes the facts with `portray_clause/2` to a file that is then consulted.

16.6 Derived Relations

Three relations are derived from `msg/6` so often in Part IV that this chapter defines them once. `delta/3` relates a message to the next one and the time between them; `thread_messages/2` gives the ordinal numbers of the messages a thread wrote; `component_messages/2` does the same for a component.

```
% delta(N1, N2, Delta): N2 is the message after N1
% in the trace and Delta the seconds between them.
delta(N1, N2, Delta) :-
    msg(N1, T1, _, _, _, _),
    N2 is N1 + 1,
    msg(N2, T2, _, _, _, _),
    Delta is T2 - T1.

% thread_messages(TID, Ns): Ns is the ordered list
% of the numbers of the messages written by TID.
thread_messages(TID, Ns) :-
    setof(N,
          T^P^C^Text^msg(N, T, P, TID, C, Text),
          Ns).

% component_messages(Component, Ns): Ns is the
% ordered list of the numbers of the messages of
% Component.
component_messages(Component, Ns) :-
    setof(N,
          T^P^TID^Text^msg(N, T, P, TID,
                           Component, Text),
          Ns).
```

`delta/3` is written for the mode `+N1`: with the number bound, two indexed lookups and a subtraction, deterministic; with it unbound, it enumerates every adjacent pair. The adjacency `N2 is N1 + 1` is where the ordinal number does what the timestamp cannot.

```
?- delta(66, N, D).
N = 67,
D = 4.691000000000001.

?- delta(16, N, D).
N = 17,
D = 0.0009999999999998899.

?- delta(49, N, D).
N = 50,
D = 0.0.

?- delta(66, _, D), format("~3f~n", [D]).
4.691
D = 4.691000000000001.

?- aggregate_all(max(D, N1-N2), delta(N1, N2, D),
   Max).
Max = max(4.691000000000001, 66-67).
```

The first answer is the gap in the trace: 4.691 seconds between the last message of the drained queue and the first message of thread 109, the **Discontinuity** (page 513) of Chapter 25 found by one aggregation. The second and third show what **floating-point** numbers do. The difference of 1.311 and 1.310 is not the decimal `0.001` but the nearest double to the difference of the two nearest doubles, which prints as `0.0009999999999998899`; and the difference of two equal timestamps is exactly `0.0`. These are not errors in the data; they are the representation, and three rules follow from them. A threshold comparison, `D > 0.5`, is unaffected, since the noise is many orders of

magnitude below the resolution of the clock. A printed delta is formatted with `~3f`, which rounds to the clock's resolution. And an equality test on times uses `=:=`, or better a tolerance, never `==` or unification. A knowledge base whose timestamps must be exact keeps them as integers in the clock's unit, microseconds or milliseconds, and divides only for display, which is what the log parser of Section 16.3 did until its last step; the schema keeps seconds as floats because log formats give them that way and because Part IV compares them against thresholds, where the noise does not matter.

The two list relations use `setof/3` rather than `findall/3`, and the choice is deliberate. With the thread bound, `setof/3` collects that thread's numbers in standard order, which for integers is numeric order, so the list is the thread's messages in trace order. With the thread unbound, the free variable `TID` in the goal makes `setof/3` group by thread, as Section 9.2 explained, and the predicate enumerates the threads of the trace, each with its list. And `setof/3` fails when there are no solutions, where `findall/3` would succeed with `[]`: a thread with no messages is not a thread of the trace, and failure is the right answer.

```
?- thread_messages(108, Ns).
Ns = [48, 49, 50, 51, 52].

?- thread_messages(109, Ns).
Ns = [67, 68, 69, 70].

?- thread_messages(107, Ns).
false.

?- thread_messages(TID, Ns), length(Ns, L),
   format("~d~t~6|~t~d~4+~n", [TID, L]), fail.
100     22
101     21
102     11
103      4
104      4
108      5
109      4
false.

?- component_messages(cachelib, Ns).
Ns = [3].

?- component_messages(C, Ns), length(Ns, L),
   format("~a~t~14|~t~d~4+~n", [C, L]), fail.
appserver       47
cachelib         1
dbclient        12
reportengine    11
false.
```

The list of a thread's messages is the catalog's **Thread of Activity** (page 537) in its simplest form, the messages one thread wrote in the order it wrote them, and Chapter 26 builds the activity patterns on it; the list of a component's messages is a **Trace Partition** (page 489) by component, which Chapter 24 treats. The table of counts by thread is the first thing an analyst learns about a trace, and

the table by component the second: here the application server wrote two thirds of the messages, and the cache library one.

16.7 Indexing and Scale

A trace of 71 messages is read in full by any query. A trace of a million is not, and whether a query over it takes a millisecond or a second depends on indexing, which SWI-Prolog does for `msg/6` without being asked but not uniformly. The following starts a fresh toplevel so that the state of the indexes is known.

```
:- [trace1].

?- predicate_property(msg(_, _, _, _, _, _),
   number_of_clauses(N)).
N = 71.

?- jiti_list(msg/6).
Predicate <...> Indexed  Buckets Speedup Flags
<...>
true.

?- msg(48, T, _, _, _, _).
T = 9.002.

?- jiti_list(msg/6).
Predicate <...> Indexed  Buckets Speedup Flags
<...>
user:msg/6 <...> 1 <...> 128 <...> 71.0
true.

?- aggregate_all(count, msg(_, _, _, 108, _, _),
   N).
N = 5.

?- jiti_list(msg/6).
Predicate <...> Indexed  Buckets Speedup Flags
<...>
user:msg/6 <...> 1 <...> 128 <...> 71.0
                 <...> 4 <...> 8 <...> 4.0
true.
```

`jiti_list/1` prints the indexes SWI-Prolog has built for a predicate; the table is abridged here to the width of the page. Immediately after consulting there are none. The first query with a bound first argument creates a hash index on that argument, with 128 buckets for 71 distinct keys and an estimated speedup of 71, meaning that a lookup by number examines one clause instead of all 71. The first query with a bound fourth argument and an unbound first creates a second index, on the thread identifier, with eight buckets for seven distinct values and a smaller estimated speedup, since each bucket holds several messages. This is **just-in-time indexing**: SWI-Prolog builds an index on an argument the first time a call would otherwise scan the clauses with that argument bound and others unbound, and keeps it. A query by component creates a third index on the fifth argument, and a

query with the text bound a fourth, which is of little use, since texts are nearly all distinct and a lookup by exact text is rare.

Two consequences for Part IV follow. A recognizer that visits messages by number, as `delta/3` does, costs one indexed lookup per message and runs in time proportional to the trace. A recognizer that selects by thread or component costs one indexed lookup per selected message after the first call has built the index, and the first call pays for building it, which on a million messages is noticeable once. A recognizer that selects by a property of the text, such as every message containing `Error`, scans the trace, because no index helps `sub_string/5`; Chapter 24 derives a relation of message kinds once, keyed by ordinal number, so that such selections are indexed thereafter.

Memory and loading time are the other dimension of scale, and the numbers are of an order of magnitude, not a specification. `predicate_property/2` reports the memory a predicate's clauses occupy, and `time/1` the cost of a goal:

```
?- predicate_property(msg(_, _, _, _, _, _),
   size(Bytes)).
Bytes = <...>.

?- time([trace1]).
% <...> inferences, <...> CPU in <...>
true.
```

In this edition, on a 64-bit build, the 71 clauses take about 25 kilobytes, some 350 bytes per message, of which the text is the larger part; a trace of a million messages of this shape would need on the order of a few hundred megabytes of clause storage, plus the indexes, and would consult in tens of seconds. These figures vary with the build, the platform, and the length of the messages, and the `<...>` in the transcript marks what varies between runs. Chapter 30 measures a generated trace of a hundred thousand messages and shows what to do when the figures matter: `qcompile/1` for faster loading, keeping only the attributes a question needs, and streaming a trace through `phrase_from_file/2` without loading it at all.

16.8 Several Traces in One Knowledge Base

A knowledge base that holds two traces, from two processes, two machines, or two runs of one program, cannot hold them both as `msg/6`, because their ordinal numbers collide and their time bases differ. The convention for this case, used in Chapter 28 for the **Master Trace** and **Inter-Correlation** patterns, is `msg/7`: the same six arguments preceded by a **trace identifier**, an atom naming the trace, `msg(Trace, N, Time, PID, TID, Component, Text)`.

The identifier comes first for the reason the ordinal number comes first in `msg/6`: it is the key by which a query selects one trace, and indexing on it makes the selection a lookup. The ordinal number is then unique within a trace, not across the knowledge base, and the time is seconds from that trace's first message, not from a common origin; aligning two traces in time requires an offset per trace that the analyst establishes from a message both traces record, which is the **Shared Point** (page

571) of Chapter 28, and the convention leaves the times as recorded rather than shifting them at load.

The shared trace is lifted to the convention by a rule, which Chapter 28 uses rather than a copy of the facts:

```
% msg(Trace, N, Time, PID, TID, Component, Text):
% the message N of the trace named Trace.
msg(trace1, N, Time, PID, TID, Component, Text) :-
    msg(N, Time, PID, TID, Component, Text).

?- msg(trace1, 71, T, _, TID, _, Text).
T = 14.01,
TID = 100,
Text = "Service stopping".

?- aggregate_all(count,
   msg(trace1, _, _, _, _, _, _), N).
N = 71.

?- aggregate_all(set(Tr),
   msg(Tr, _, _, _, _, _, _), Traces).
Traces = [trace1].
```

A second trace is added as facts of `msg/7` under its own identifier, parsed by the same grammar with the identifier supplied by the loader; the following is illustrative and not consulted:

```
msg(client1, 1, 0.000, 5200, 300, client,
    "GET /report/17").
msg(client1, 2, 0.412, 5200, 300, client,
    "200 OK, 309 ms").
```

The two predicates differ in arity and are therefore different predicates to Prolog, so `msg/6` and `msg/7` coexist, and a chapter that works on one trace keeps the shorter form. The derived relations of Section 16.6 have `msg/7` counterparts with the trace identifier added as a first argument, written when Chapter 28 needs them.

Chapter 17. The Pattern as a Predicate

Parts III and IV write one or more predicates for each diagnostic analysis pattern of the two catalogs, and this chapter fixes what such a predicate is, what it is not, and the forms it takes. A pattern in the catalog is a description written for an analyst: a name, the situation in which the pattern is found, what it looks like in the artifact, what it tends to mean, and what to do next. A predicate is a relation over the facts of a knowledge base. The chapter says how the second stands in for the first, what is lost in the substitution, and which conventions make the predicates of twelve chapters usable together: three forms for a recognizer, an evidence term that says why an instance was recognized, thresholds kept as facts, a catalog of patterns kept as data, a procedure that runs every recognizer and reports what it finds in order of severity, and a test suite that keeps the recognizers honest.

The chapter uses `dump1.pl` and a handful of memory analysis patterns that Part III treats in full; they are chosen because their recognizers are short, and their definitions here are provisional versions of the ones in Chapters 18 to 22. The vocabulary introduced, recognizer, instance, evidence, threshold, test form, enumerator, explainer, is the vocabulary of every pattern section that follows.

17.1 Description and Recognizer

A **pattern description** in the catalog is a text about a kind of situation, and it addresses a reader who brings context: knowledge of the system, of what was expected, of what the other artifacts say, and of what has been tried. A **pattern recognizer** is a predicate that holds for the elements of a knowledge base that have the shape the description gives, and it brings no context. The difference is not one of precision but of scope. The description of **Spiking Thread** (page 335) says, in outline, that a thread consuming processor time out of proportion to the others is likely to be in a loop or doing work that should not be happening, and that its stack trace shows where; the recognizer `spiking_thread/1` of Chapter 6 says that a thread's share of the process's processor time is at least 25 percent. The predicate is the description's observable part, cut off from its interpretive part, with a number put where the description had a judgment.

Three things are lost in the cut. The **context** is lost: whether 25 percent is a lot depends on how many threads do work in this process, which the description leaves to the analyst and the predicate fixes as a threshold. The **intent** is lost: the description says why the pattern matters and what to look at next, and the predicate says only that the shape is present. And the analyst's judgment about whether this instance is the pattern, or a coincidence with its shape, is lost: a thread may have a high share because the process has just started and only it has run, and the predicate cannot know that. What is gained is that the shape is checked on every thread of every dump the same way, in a form another analyst can read, run, and dispute.

The vocabulary follows the catalog's. A pattern is **recognized** by a predicate when the predicate finds the shape; it is **identified** by an analyst when the analyst accepts the instance as the pattern. A **pattern instance** is the element of the knowledge base the shape was found in: a thread, a module,

a pair of threads, a list of them, a message or a range of messages, whatever the pattern is about. The predicates of this book recognize; identification remains with the reader.

17.2 The Three Forms

A recognizer takes one of three forms, and a pattern section in Parts III and IV gives at least the first two. The forms are distinguished by what the caller supplies and what the caller gets.

The **test form**, `p(+Instance)`, is given a candidate and succeeds if it is an instance of the pattern. It is a Boolean test and must be semideterministic: it succeeds at most once and leaves no choice point. The **enumerator**, `p(-Instance)`, is given nothing and produces each instance in turn on backtracking, each exactly once. In Prolog the two are one predicate in two modes, and the discipline is to write the predicate so that both modes behave: indexing on a bound first argument gives the test form its determinism, and the generator at the head of the body gives the enumerator its completeness. The **explainer**, `p(?Instance, -Evidence)`, is the same relation with a second argument that says why, in the form of Section 17.3.

The following defines the three forms for **Spiking Thread** (page 335), a thread whose share of the process's processor time is out of proportion to the other threads', with the thresholds of Section 17.4 and the CPU share of Chapter 9.

```
:- [dump1].

% threshold(Name, Value): the named threshold used
% by the recognizers of this chapter.
threshold(spiking_share, 25).
threshold(handle_count, 10000).
threshold(frame_repeats, 5).
threshold(null_page, 0x1_0000).

% cpu_time(TID, Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% total_cpu(Total): the CPU time of all threads.
total_cpu(Total) :-
    aggregate_all(sum(Ms), cpu_time(_, Ms),
                  Total).

% cpu_share(TID, Percent): the CPU time of TID as
% a percentage of the total.
cpu_share(TID, Percent) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Percent is 100.0 * Ms / Total.
```

```
% spiking_thread(TID, E): TID is a Spiking Thread
% instance, and E is the evidence.
spiking_thread(TID, E) :-
    thread_time(TID, User, Kernel),
    cpu_share(TID, Share),
    threshold(spiking_share, Min),
    Share >= Min,
    E = evidence('Spiking Thread', TID,
        [ thread_time(TID, User, Kernel),
          cpu_share(TID, Share),
          threshold(spiking_share, Min)
        ]).

% spiking_thread(TID): TID is a Spiking Thread
% instance.
spiking_thread(TID) :-
    spiking_thread(TID, _).
```

The explainer is the definition and the other two forms are derived from it: `spiking_thread/1` calls `spiking_thread/2` and discards the evidence, and the test form is `spiking_thread/1` with its argument bound. Writing the explainer first costs the construction of an evidence term that the test form throws away, which is a few cells of memory per call and is the right trade in a book about explanation; a recognizer whose evidence is expensive to build defines the one-argument form directly and the explainer separately.

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- spiking_thread(102).
true.

?- spiking_thread(100).
false.

?- spiking_thread(T).
T = 102 ;
T = 107 ;
false.

?- spiking_thread(107, E).
E = evidence('Spiking Thread', 107, <...>).
```

The flag `toplevel_print_anon` is set as in Chapter 5, so that variables named with a leading underscore, which this chapter uses for lists of evidence terms that would fill a page, are not printed. The test form answers `true.` with no choice point, because `thread_time/3` with a bound first argument is selected by indexing and the rest of the body is deterministic. The enumerator produces the two instances and then `false.`, the choice point being `thread_time/3` trying its remaining facts; that it ends with `false.` rather than a third instance is what the enumerator form requires. The explainer gives the evidence, which the next section examines.

The first argument of a recognizer is the instance, and what an instance is depends on the pattern. For **Spiking Thread** (page 335) it is a thread identifier. For a pattern about a module it is the module's

atom; for a pattern about a relation between two threads it is a pair `T1-T2`; for a chain it is a list; for a trace pattern it is a message number, a pair of numbers bounding a range, or a thread identifier. The instance is always a ground term built from the identifiers of the knowledge base, so that two instances can be compared with `==` and a report can print one with `~w`.

17.3 Evidence

An **evidence term** is `evidence(Pattern, Instance, Facts)`: the pattern's catalog name as an atom, the instance, and a list of goals from which the recognizer concluded that the instance is one. The list is under a **contract** that every recognizer of this book keeps, and this section states it once. Each element is a ground goal that holds: a fact of the knowledge base, a `threshold/2` fact, or a goal of a predicate the program defines, such as `cpu_share/2`, with every argument bound, so that calling it succeeds. The list holds what the recognizer used, the facts it looked up, the derived values it computed, and the thresholds it compared against, in the order of the clause, and nothing else: an annotation such as the atom `suspected` or a term `depth(14)`, which describes rather than asserts, is not a goal and has no place in it. And the list is sufficient: the shape the recognizer found follows from the listed goals alone, without recourse to the rest of the knowledge base; in particular every threshold the clause compared against is in the list, since a comparison cannot be repeated without its limit. The first two parts of the contract are checked by calling the goals, and the third by re-deriving the shape from the list, as the end of this section shows.

The goals are written unqualified, as the facts and the thresholds are written in the knowledge base, and the contract fixes where they can be called: the **execution context** of an evidence list is the module that defines the recognizer and any module that imports it. A chapter's recognizers are defined in `user`, beside the facts of the dump, and their leaves are callable in that chapter's toplevel. The module `patterns_dump` of Chapter 23, which collects the recognizers of Chapters 18 to 22, therefore exports every predicate that can appear as a leaf, the helpers `cpu_time/2` and `total_cpu/1` among them, and not the recognizers alone: a leaf such as `cpu_time(102, 9680)` is then callable, as written, from any module that imports `patterns_dump`, and a reader who copies a list out of a report can run it. The alternative representations are worse on one side each. A list that qualified its derived leaves with their module, `patterns_dump:cpu_time(102, 9680)`, would be callable from anywhere and would not read as the facts of the analysis; a list of unqualified leaves whose helpers stayed private to the module would read well and raise an existence error in the hands of the reader. Section 23.10.3 checks the contract for every explainer of the module from a module that imports it.

```
?- spiking_thread(102, evidence(P, I, _Facts)),
   forall(member(F, _Facts), ( print(F), nl )).
thread_time(102,8450,1230)
cpu_share(102,68.48249027237354)
threshold(spiking_share,25)
P = 'Spiking Thread',
I = 102.
```

```
?- spiking_thread(T, evidence(_, _, _Facts)),
   maplist(call, _Facts).
T = 102 ;
T = 107 ;
false.
```

The evidence for thread 102 is its two recorded times, its computed share, and the threshold the share was compared with. Every element is a goal that succeeds: `thread_time(102, 8450, 1230)` is a fact of the dump, `cpu_share(102, 68.48249027237354)` recomputes to the same float, and `threshold(spiking_share, 25)` is a fact of this chapter. The second query checks this for every instance with `maplist(call, _Facts)`, which is the test of the first two parts of the contract: a recognizer that puts into its evidence a goal that does not hold, or a term that is not a goal, has a bug, and the test finds it.

The evidence is for three readers. An analyst reads it as the reason to look at thread 102 and as the numbers to quote. A report prints it under the instance. And a meta-interpreter, in Chapter 29, derives it automatically from the recognizer's clause rather than from a hand-written list, and compares the two. Until then the list is written by hand, under the contract.

A recognizer sometimes has grounds for a pattern it cannot establish from the facts. A deep and regular stack trace is grounds to suspect a **Stack Overflow (User Mode)** (page 366), which Chapter 19 treats, and only the exception code establishes one. Such a recognizer does not weaken its evidence term by putting a status among the goals. It reports a **suspicion term**, `suspected(Pattern, Instance, Grounds)`, whose third argument is under the same contract as `Facts`: ground goals that hold and are sufficient for the grounds, though not for the pattern. The functor carries the status and the list carries goals only. A consumer tells the two forms apart by the functor: a report lists suspected instances apart from recognized ones, and a test that an instance was recognized does not count a suspicion. Section 19.9 writes a recognizer with both forms.

Evidence terms nest. When a recognizer calls another, as Section 17.5 does, it puts the other's evidence term into its own list, and the list is then a tree whose leaves are goals. Checking such a tree needs a predicate that descends into nested evidence, and into suspicion terms, and calls the leaves:

```
% holds(Term): Term is a goal that succeeds, or an
% evidence or suspicion term all of whose goals
% hold.
holds(Term) :-
    (   Term = evidence(_, _, Goals)
    ->  maplist(holds, Goals)
    ;   Term = suspected(_, _, Goals)
    ->  maplist(holds, Goals)
    ;   once(Term)
    ).
```

`once/1` is used for the leaf case so that `holds/1` is semideterministic whatever the goals are: a derived goal such as `cpu_share/2` may leave a choice point, and a checker that leaves one for each of a hundred facts is a checker that plunit, in Section 17.10, will complain about.

A list of goals that all hold is not yet evidence for the instance it is attached to, since the goals may be true and beside the point: the times of thread 100 and the spiking threshold hold, and say nothing about thread 102. The third part of the contract is checked by **re-derivation**: the shape of the pattern, restated as a conjunction over the vocabulary of the evidence, must follow from the list alone. `entails/2` is the device. A goal follows from a list when it is a member of the list; an arithmetic comparison follows when it holds between values the members bound; a conjunction follows when both parts do.

```
% entails(Facts, Condition): Condition follows
% from the list Facts alone: a conjunction of
% goals, each a member of Facts or an arithmetic
% comparison between values the members bound.
entails(Facts, Condition) :-
    (   Condition = (A, B)
    ->  entails(Facts, A),
        entails(Facts, B)
    ;   comparison(Condition)
    ->  call(Condition)
    ;   member(Condition, Facts)
    ).

% comparison(Goal): Goal is an arithmetic
% comparison.
comparison(_ < _).
comparison(_ > _).
comparison(_ =< _).
comparison(_ >= _).
comparison(_ =:= _).
comparison(_ =\= _).

?- spiking_thread(102, evidence(_, T, _Fs)),
   entails(_Fs, ( cpu_share(T, S),
                  threshold(spiking_share, Min),
                  S >= Min )).
T = 102,
S = 68.48249027237354,
Min = 25 ;
false.

?- _Fs = [thread_time(100, 120, 40),
          threshold(spiking_share, 25)],
   maplist(call, _Fs).
true.

?- _Fs = [thread_time(100, 120, 40),
          threshold(spiking_share, 25)],
   entails(_Fs, ( cpu_share(102, S),
                  threshold(spiking_share, Min),
                  S >= Min )).
false.
```

The condition is the recognizer's clause body restated over the list: `member/2` binds the share and the threshold from the list, not from the knowledge base, and the comparison is then between the listed values. The `;` after the first answer is the choice point of `member/2`, which has not reached the

end of the list. The second and third queries check a boundary case: two true facts about another thread pass the calling test and fail the re-derivation, which is the difference between a list of true goals and evidence for thread 102. Every explainer of Parts III and IV can be checked this way, and the chapters do so where the shape is more than a comparison, as Chapter 20 does for a deadlock. The condition is written by hand here, beside the recognizer it restates; the module of Chapter 23 removes the hand from the check with `derivable/2`, a meta-interpreter in the manner of Chapter 29 that re-derives a finding through the recognizer's own clauses, taking a fact of the dump or a threshold from the list only, and Section 23.10.3 runs it over every explainer of the module.

17.4 Thresholds as Facts

A **threshold** is a number a recognizer compares a measured value against, and the discipline of this book is that every threshold is a fact of `threshold/2`, named, and never a literal in a clause body. Chapter 6 kept the spiking threshold as `spiking_threshold/1`; the general form is `threshold(Name, Value)`, so that every threshold in a knowledge base is found by one query, printed in one report, and changed in one place.

```
?- threshold(Name, Value).
Name = spiking_share,
Value = 25 ;
Name = handle_count,
Value = 10000 ;
Name = frame_repeats,
Value = 5 ;
Name = null_page,
Value = 65536.
```

The reasons are three. A threshold is a judgment, not a fact about the artifact, and it belongs where an analyst can see it; the evidence term includes it, so a report says not only that thread 107 is spiking but that it is spiking against 25 percent, and a reader who thinks 25 is low knows what to dispute. A threshold is a parameter of the analysis, and an analysis that runs the same recognizers with different thresholds over the same dump, which is how an analyst learns which thresholds suit a system, is one `retract/1` and `assertz/1` away, or one `threshold/2` clause away in a file consulted after the defaults. And a threshold that is a fact can be derived: a knowledge base that holds a hundred dumps of one system can compute the distribution of CPU shares and set `spiking_share` from it, which a literal in a clause body cannot be.

Not every number in `threshold/2` is a judgment. The null page limit, `0x1_0000`, is a property of the platform: the lowest 64 KB of a user-mode address space on Windows are never mapped, and an access below it is an access through a null pointer with an offset. It is kept in `threshold/2` all the same, because the discipline is a named fact for every constant a recognizer compares against, and because another platform has another limit.

The recognizers below use the remaining thresholds. `handle_leak/2` compares a handle count with `handle_count`; `frame_regularity/2` compares the length of the longest run of consecutive frames

naming one function with `frame_repeats`; `null_pointer_data/2` compares the target of an access violation with `null_page`.

```
% handle_leak(Type, E): handles of Type exceed the
% threshold; a Handle Leak candidate.
handle_leak(Type, E) :-
    handle_count(PID, Type, Count),
    threshold(handle_count, Max),
    Count > Max,
    E = evidence('Handle Leak', Type,
        [ handle_count(PID, Type, Count),
          threshold(handle_count, Max)
        ]).

handle_leak(Type) :-
    handle_leak(Type, _).

% null_pointer_data(TID, E): TID faulted reading
% or writing an address in the null page.
null_pointer_data(TID, E) :-
    exception_record(TID, Code, Addr),
    Code =:= 0xc0000005,
    exception_info(TID, access, Access),
    Access \== execute,
    exception_info(TID, target, Target),
    threshold(null_page, Limit),
    Target < Limit,
    E = evidence('NULL Pointer (Data)', TID,
        [ exception_record(TID, Code, Addr),
          exception_info(TID, access, Access),
          exception_info(TID, target, Target),
          threshold(null_page, Limit)
        ]).

null_pointer_data(TID) :-
    null_pointer_data(TID, _).

% frame_function(Frame, F): F is the function of
% the frame/5 term Frame.
frame_function(frame(_, _, _, F, _), F).

% frame_runs(Frames, Runs): Runs is the list of
% maximal runs of consecutive frames of Frames
% that name one function, each run a list.
frame_runs([], []).
frame_runs([Frame | Frames],
           [[Frame | Run] | Runs]) :-
    frame_function(Frame, F),
    same_function(Frames, F, Run, Rest),
    frame_runs(Rest, Runs).
```

```
% same_function(Frames, F, Run, Rest): Run is the
% longest prefix of Frames whose frames name F;
% Rest is what follows it.
same_function(Frames, F, Run, Rest) :-
    (   Frames = [Frame | Frames1],
        frame_function(Frame, F)
    ->  Run = [Frame | Run1],
        same_function(Frames1, F, Run1, Rest)
    ;   Run = [],
        Rest = Frames
    ).

% frame_regularity(TID, E): the longest run of
% consecutive frames of TID naming one function is
% at least the threshold long; the frames of the
% run are the evidence.
frame_regularity(TID, E) :-
    thread(TID, _),
    findall(frame(TID, N, M, F, O),
            frame(TID, N, M, F, O), Frames0),
    sort(2, @=<, Frames0, Frames),
    frame_runs(Frames, Runs),
    aggregate_all(max(C, R),
                  ( member(R, Runs),
                    length(R, C)
                  ),
                  max(Count, Run)),
    threshold(frame_repeats, Min),
    Count >= Min,
    append(Run, [threshold(frame_repeats, Min)],
           Facts),
    E = evidence('Frame Regularity', TID, Facts).

frame_regularity(TID) :-
    frame_regularity(TID, _).


?- handle_leak(T, E).
T = event,
E = evidence('Handle Leak', event, <...>) ;
false.

?- null_pointer_data(T, evidence(_, _, _Facts)),
   forall(member(F, _Facts), ( print(F), nl )).
exception_record(109,3221225477,140718380167594)
exception_info(109,access,read)
exception_info(109,target,24)
threshold(null_page,65536)
T = 109.

?- frame_regularity(T).
T = 107 ;
false.

?- frame_regularity(107, evidence(_, _, _Facts)),
   length(_Facts, N), last(_Facts, Last).
N = 9,
Last = threshold(frame_repeats, 5).
```

Three recognizers, three kinds of instance, three thresholds. The handle type `event` has 12,040 handles against a limit of 10,000, which Chapter 21 treats as **Handle Leak** (page 409), a handle count growing without bound, with the caution that one dump shows a count and not its growth. Thread 109 read address `0x18`, below the null page limit, with the access violation code, which Chapter 19 treats as **NULL Pointer (Data)** (page 360); the recognizer checks the code, the access kind, and the target, and would not fire on an execute access, which is **NULL Pointer (Code)** (page 361), or on a different exception code. Thread 107 has eight consecutive frames naming `Node::Visit` against a threshold of five, which Chapter 18 treats as **Frame Regularity** (page 342); its evidence is the eight frames of the run and the threshold, nine terms. `frame_runs/2` splits a trace, ordered by frame number, into maximal runs of one function, in the manner of the run counting that Chapter 18 develops, and `aggregate_all(max(C, R), ...)` from Chapter 9 takes the longest run without a choice point escaping into the recognizer. This provisional recognizer reports one run per thread, the longest; Chapter 18's `frame_regularity/3`, which the later chapters and the module of Chapter 23 use, is written differently: it enumerates every run of a thread's signature that reaches its threshold, one answer per run, and selects no longest. The two agree on which threads have the pattern and differ in what they report.

A run is consecutive frames. A function that occurs often but not consecutively is not regular in this sense, and the lengths of separate runs are not added together; the following query checks that boundary case on a constructed frame list in which one function occurs six times in three runs of two:

```
?- findall(frame(0, N, m, F, 0),
           nth0(N, [f, f, g, f, f, g, f, f], F),
           _Fs),
   frame_runs(_Fs, _Runs),
   maplist(length, _Runs, Lengths).
Lengths = [2, 1, 2, 1, 2].
```

The longest run is two, and a recognizer that counted occurrences, or added the runs, would report six and be wrong about the pattern, whose intent is recursion visible as repetition down the stack. `nth0/3` enumerates the positions of the list of function names and gives each synthetic frame its number.

17.5 Composition

A pattern is a goal, and a recognizer may call another recognizer in its body. The catalog's patterns are related in this way: a **Deadlock (Critical Sections)** (page 388) is a set of threads each blocked on a critical section that another of them owns, so that none can proceed, in its simplest form two threads each holding what the other waits for; each of them is a **Blocked Thread (Software)** (page 336), a thread stopped by a synchronization object under software control; and a **Wait Chain (Critical Sections)** (page 384) is a sequence of threads each blocked on a section the next owns, of which a deadlock is the case where the chain returns to its start. The recognizers compose the same way. This section writes the two-thread deadlock; Chapter 20 writes the general cycle.

```
% blocked_on(TID, Addr, Owner): TID waits for the
% critical section at Addr, which Owner holds.
blocked_on(TID, Addr, Owner) :-
    waits_for(TID, critical_section, Addr),
    critical_section(Addr, Owner, _, _).

% blocked_thread(TID, E): TID is blocked on a
% critical section.
blocked_thread(TID, E) :-
    blocked_on(TID, Addr, Owner),
    critical_section(Addr, Owner, Lock, Rec),
    E = evidence('Blocked Thread (Software)', TID,
        [ waits_for(TID, critical_section, Addr),
          critical_section(Addr, Owner, Lock, Rec)
        ]).

blocked_thread(TID) :-
    blocked_thread(TID, _).

% deadlock(Pair, E): Pair is T1-T2, T1 < T2, each
% blocked on a section the other owns.
deadlock(T1-T2, E) :-
    blocked_on(T1, _, T2),
    blocked_on(T2, _, T1),
    T1 < T2,
    blocked_thread(T1, E1),
    blocked_thread(T2, E2),
    E = evidence('Deadlock (Critical Sections)',
                 T1-T2, [E1, E2]).

deadlock(Pair) :-
    deadlock(Pair, _).
```

`blocked_on/3` is a helper, not a pattern: it joins the two relations of the schema that describe a wait from the two sides, and both recognizers use it. `deadlock/2` finds the cycle of length two through two `blocked_on/3` goals, orders the pair so that each deadlock is found once, and then calls `blocked_thread/2` on each thread for its evidence, which it nests in its own. The order of goals matters for determinism: the structural test comes first, and the explainers are called with bound arguments, so that they add no choice points.

```
?- deadlock(P).
P = 103-104 ;
false.

?- deadlock(103-104).
true.

?- deadlock(104-103).
false.

?- deadlock(P, _E), holds(_E).
P = 103-104 ;
false.
```

```
?- deadlock(_, evidence(_, _, [_E1, _E2])),
   _E1 = evidence(P1, I1, _),
   _E2 = evidence(P2, I2, _).
P1 = P2, P2 = 'Blocked Thread (Software)',
I1 = 103,
I2 = 104 ;
false.
```

The deadlock is the pair 103-104 and only that pair: 104-103 fails by the ordering, and 101, which is blocked behind 104, is not in a cycle. The evidence is a tree with two **Blocked Thread (Software)** (page 336) terms as children, and `holds/1` verifies the tree down to its four facts. The ordering `T1 < T2` is the general device against reporting one instance twice from a symmetric relation, and every recognizer over pairs uses it or `@<` for non-numeric instances.

The sufficiency check of Section 17.3 applies to the tree once its leaves are collected. `leaves/2` flattens a nested evidence term into the list of goals at its leaves, and the shape of the two-thread deadlock, each thread waiting for a section the other owns, is then re-derived from that list and from nothing else:

```
% leaves(Term, Goals): Goals are the goals at the
% leaves of a nested evidence term, in order; a
% goal is its own leaf.
leaves(Term, Goals) :-
    (   Term = evidence(_, _, Facts)
    ->  maplist(leaves, Facts, Lists),
        append(Lists, Goals)
    ;   Goals = [Term]
    ).

?- deadlock(T1-T2, _E), leaves(_E, _Gs),
   length(_Gs, N),
   entails(_Gs,
           ( waits_for(T1, critical_section, A1),
             critical_section(A1, T2, _, _),
             waits_for(T2, critical_section, A2),
             critical_section(A2, T1, _, _) )).
T1 = 103,
T2 = 104,
N = 4,
A1 = 140695026508096,
A2 = 140695026507968 ;
false.
```

The four leaves are the two waits and the two ownerships, and they entail the cycle: thread 103 waits for the section at `0x7ff6_1d13_a140`, which 104 owns, and 104 waits for the section at `0x7ff6_1d13_a0c0`, which 103 owns. `append/2` concatenates a list of lists.

The wait chain generalizes the deadlock to any length and to chains that end at an unblocked thread. Its instance is the list of threads from the one that heads the chain to the one that holds the last section, and when the chain closes on itself the closing thread appears twice, at the end, so that the cycle is visible in the instance.

```
% wait_chain(Chain, E): Chain is a list of threads
% headed by a blocked thread, each waiting for a
% critical section the next owns, ending at a
% thread that is not blocked or that repeats.
wait_chain([TID | Rest], E) :-
    blocked_on(TID, _, _),
    chain_from(TID, [TID], Rest, Facts),
    E = evidence('Wait Chain (Critical Sections)',
                 [TID | Rest], Facts).

% chain_from(TID, Seen, Rest, Facts): Rest is the
% chain of owners from TID on, stopping at an
% unblocked thread or at one in Seen.
chain_from(TID, Seen, Rest, Facts) :-
    (   blocked_on(TID, A, Owner)
    ->  critical_section(A, Owner, L, R),
        Facts = [ waits_for(TID, critical_section,
                            A),
                  critical_section(A, Owner, L, R)
                | Facts1 ],
        (   memberchk(Owner, Seen)
        ->  Rest = [Owner],
            Facts1 = []
        ;   Rest = [Owner | Rest1],
            chain_from(Owner, [Owner | Seen],
                       Rest1, Facts1)
        )
    ;   Rest = [],
        Facts = []
    ).

wait_chain(Chain) :-
    wait_chain(Chain, _).

?- wait_chain(C).
C = [101, 103, 104, 103] ;
C = [103, 104, 103] ;
C = [104, 103, 104].

?- wait_chain([101, 103, 104, 103]).
true.

?- wait_chain([101 | Rest]).
Rest = [103, 104, 103].
```

Three chains, one from each blocked thread, and each ends in the cycle 103, 104, 103. The chain from 101 is the one an analyst would report: a thread waiting behind a deadlock it is not part of. `chain_from/4` uses the if-then-else of Chapter 7 for both choices, so the recognizer is deterministic in test mode and leaves in enumerator mode only the choice point of `blocked_on/3` at its head. Chapter 20 treats the three patterns in full, with the tabled formulation of Chapter 11 for the closure and the cases the shared data does not have.

17.6 The Catalog as Data

The patterns a knowledge base knows about are themselves facts. `pattern(Name, Kind, Chapter)` records a pattern's catalog name, the kind of artifact it is found in, `memory_dump` or `software_trace`, and the chapter of this book that treats it; `recognizer(Name, Predicate)` records the name of the predicate whose two-argument form is the pattern's explainer. The two relations are separate because a knowledge base can know of a pattern without having a recognizer for it, which is the case for every pattern this book treats in prose only.

```
% pattern(Name, Kind, Chapter): the pattern Name
% occurs in artifacts of Kind; Chapter treats it.
pattern('Spiking Thread', memory_dump, 18).
pattern('Blocked Thread (Software)', memory_dump,
        18).
pattern('Frame Regularity', memory_dump, 18).
pattern('NULL Pointer (Data)', memory_dump, 19).
pattern('Wait Chain (Critical Sections)',
        memory_dump, 20).
pattern('Deadlock (Critical Sections)',
        memory_dump, 20).
pattern('Handle Leak', memory_dump, 21).
pattern('No Component Symbols', memory_dump, 22).
pattern('Hooked Functions (User Space)',
        memory_dump, 22).
pattern('Time Delta', software_trace, 25).
pattern('Discontinuity', software_trace, 25).

% recognizer(Name, Pred): Pred/2 is the explainer
% for the pattern Name.
recognizer('Spiking Thread', spiking_thread).
recognizer('Blocked Thread (Software)',
           blocked_thread).
recognizer('Frame Regularity', frame_regularity).
recognizer('NULL Pointer (Data)',
           null_pointer_data).
recognizer('Wait Chain (Critical Sections)',
           wait_chain).
recognizer('Deadlock (Critical Sections)',
           deadlock).
recognizer('Handle Leak', handle_leak).
recognizer('No Component Symbols',
           no_component_symbols).
recognizer('Hooked Functions (User Space)',
           hooked_functions).

% no_component_symbols(Module, E): Module has no
% symbol information.
no_component_symbols(M, E) :-
    module_info(M, symbols, none),
    E = evidence('No Component Symbols', M,
        [ module_info(M, symbols, none) ]).

no_component_symbols(M) :-
    no_component_symbols(M, _).
```

```
% hooked_functions(Function, E): the entry of
% Function in some module transfers control into
% another module.
hooked_functions(F, E) :-
    hooked(F, Module, Hooker),
    E = evidence('Hooked Functions (User Space)',
                 F, [hooked(F, Module, Hooker)]).

hooked_functions(F) :-
    hooked_functions(F, _).
```

The two last recognizers are the simplest kind, a single fact of the schema restated as a pattern: **No Component Symbols** (page 431) is a module for which the debugger had no symbol file, which Chapters 2 and 8 found by matching `none`, and **Hooked Functions (User Space)** (page 433) is a function whose entry has been redirected into another module, which the schema records directly in `hooked/3`. A recognizer this thin still earns its place in the catalog, because the report of Section 17.7 runs everything in `recognizer/2` and nothing else.

With the catalog as data, one predicate runs every recognizer. `pattern_instances/1` collects the evidence terms of every instance of every pattern that has a recognizer, using `call/3` from Chapter 10 to call the predicate named in the fact with two arguments:

```
% pattern_instances(Evidence): Evidence is the
% list of evidence terms of every pattern instance
% that a recognizer in the catalog finds.
pattern_instances(Evidence) :-
    findall(E,
            ( recognizer(_, Recognizer),
              call(Recognizer, _, E)
            ),
            Evidence).

?- pattern_instances(_Es), length(_Es, N).
N = 16.

?- pattern_instances(_Es),
   forall(member(evidence(P, I, _), _Es),
          format("~w: ~w~n", [P, I])).
Spiking Thread: 102
Spiking Thread: 107
Blocked Thread (Software): 101
Blocked Thread (Software): 103
Blocked Thread (Software): 104
Frame Regularity: 107
NULL Pointer (Data): 109
Wait Chain (Critical Sections): [101,103,104,103]
Wait Chain (Critical Sections): [103,104,103]
Wait Chain (Critical Sections): [104,103,104]
Deadlock (Critical Sections): 103-104
Handle Leak: event
No Component Symbols: cachelib
No Component Symbols: thirdparty_hook
Hooked Functions (User Space): CreateFileW
```

```
Hooked Functions (User Space): ReadFile
true.

?- pattern_instances(_Es), maplist(ground, _Es).
true.

?- pattern_instances(_Es), maplist(holds, _Es).
true.

?- pattern(P, _, _), \+ recognizer(P, _).
P = 'Time Delta' ;
P = 'Discontinuity'.

?- recognizer(P, _), \+ pattern(P, _, _).
false.
```

Sixteen instances of nine patterns over the shared dump, every one with evidence that is ground, as `ground/1` checks, and that holds, and the two patterns the catalog knows without a recognizer are the trace patterns, whose recognizers Chapter 25 writes against `trace1.pl`. The last query is a consistency check on the catalog itself: every recognizer belongs to a listed pattern. The list of evidence terms is kept out of the transcript by naming it `_Es` and printing only what is asked for.

17.7 Reports and Severity

A report lists the instances found in an order an analyst wants to read them in, and the order is by **severity**. Severity is a judgment attached to a pattern, not computed from the dump, and like a threshold it is a fact: `severity(Pattern, Level)` assigns each pattern one of five levels, and `rank/2` embeds the levels in the integers so that they can be sorted.

```
% severity(Pattern, Level): the level at which an
% instance of Pattern is reported.
severity('Deadlock (Critical Sections)',
         critical).
severity('NULL Pointer (Data)', critical).
severity('Spiking Thread', high).
severity('Handle Leak', high).
severity('Wait Chain (Critical Sections)',
         medium).
severity('Blocked Thread (Software)', medium).
severity('Frame Regularity', medium).
severity('Hooked Functions (User Space)', low).
severity('No Component Symbols', info).

% rank(Level, N): the levels in order of severity.
rank(critical, 1).
rank(high, 2).
rank(medium, 3).
rank(low, 4).
rank(info, 5).
```

```
% severity_rank(E, Rank): Rank is the rank of the
% pattern of the evidence term E.
severity_rank(evidence(Pattern, _, _), Rank) :-
    severity(Pattern, Level),
    rank(Level, Rank).

% report: print every recognized instance, grouped
% by severity, most severe first.
report :-
    pattern_instances(Evidence),
    map_list_to_pairs(severity_rank, Evidence,
                      Pairs),
    keysort(Pairs, Sorted),
    print_report(Sorted, 0).

% print_report(Pairs, Last): print the Rank-E
% pairs, with a heading whenever the rank changes
% from Last.
print_report([], _).
print_report([Rank-evidence(P, I, _) | Rest],
             Last) :-
    (   Rank == Last
    ->  true
    ;   rank(Level, Rank),
        format("~a~n", [Level])
    ),
    format("  ~a~t~33|~w~n", [P, I]),
    print_report(Rest, Rank).

?- report.
critical
  NULL Pointer (Data)            109
  Deadlock (Critical Sections)   103-104
high
  Spiking Thread                 102
  Spiking Thread                 107
  Handle Leak                    event
medium
  Blocked Thread (Software)      101
  Blocked Thread (Software)      103
  Blocked Thread (Software)      104
  Frame Regularity               107
  Wait Chain (Critical Sections) [101,103,104,103]
  Wait Chain (Critical Sections) [103,104,103]
  Wait Chain (Critical Sections) [104,103,104]
low
  Hooked Functions (User Space)  CreateFileW
  Hooked Functions (User Space)  ReadFile
info
  No Component Symbols           cachelib
  No Component Symbols           thirdparty_hook
true.
```

The report is the one an analyst reads first: the two findings that explain the dump, the deadlock and the fault, at the top; the two spiking threads and the handle count next; the blocked threads and chains that the deadlock already accounts for; and at the bottom the facts about the environment that qualify everything above, two modules without symbols and two hooked

functions. Chapter 23 writes the full report over every pattern of Part III and discusses what it must not claim.

The sort deserves a remark. Severity is a **partial order** on instances: an instance at a higher level comes before one at a lower level, and two instances at the same level are not ordered at all. `map_list_to_pairs/3` from `library(pairs)` attaches the rank to each evidence term, and `keysort/2` sorts the pairs by rank, keeping pairs with equal keys in their original order and keeping all of them. `predsort/3` would not: it removes an element when the ordering predicate says `=`, and an ordering by severity alone says `=` for every two instances at one level.

```
% by_severity(Order, E1, E2): compare two evidence
% terms by severity rank only.
by_severity(Order, E1, E2) :-
    severity_rank(E1, R1),
    severity_rank(E2, R2),
    compare(Order, R1, R2).

?- pattern_instances(_Es),
   predsort(by_severity, _Es, _Sorted),
   length(_Es, N), length(_Sorted, M).
N = 16,
M = 5.
```

Eleven of sixteen instances are gone, one survivor per level. The rule is that `predsort/3` is for orders that are total on the elements, and an order on instances that is total must break ties with the instance itself, by `compare/3` on the pair `Rank-Instance`, which standard order of terms makes total; `keysort/2` on rank alone is the simpler device when the order within a level does not matter.

17.8 Absence and the Closed World

Some patterns are about what is not there. **Manual Dump (Process)** (page 451), which Chapter 23 treats, is a dump written at an analyst's request rather than by a crash, and one of its marks is the absence of an exception record. A recognizer for it has a goal of the form `\+ exception_record(_, _, _)`, and the meaning of that goal is the **closed-world assumption** of Chapter 7: there is no exception fact in the knowledge base. Whether that means there was no exception depends on how the knowledge base was built.

Chapter 15 gave the device for saying so, in two parts. `source/3` records which command's output each relation was parsed from, and `coverage/2` says whether the capture, the command's run, and the relation's declared policy together make the relation complete, so that the absence of a fact may be read as evidence. Neither alone suffices: a command that was run on a dump lacking the stream it reads lists nothing, and a dump that holds the stream says nothing about a command that was not run. The following carries the records of Chapter 15 for the two relations this section asks about and makes the distinction a three-valued predicate. `coverage/2` here is Chapter 15's rule without its per-thread part, which neither relation needs; its needs are streams only, and a need of another kind is an error rather than a need silently met.

```
% source(Relation, Command, Log): as in Chapter
% 15, for the relations this section asks about.
source(exception_record/3, '.exr -1',
       'dump1.log').
source(unloaded_module/2, lm, 'dump1.log').

% dump_info(Key, Value), ran(Command, Outcome),
% capture_policy(Relation, Needs): the capture
% record of Chapter 15 for the same relations.
% Dynamic, for the checks at the end of this
% section.
:- dynamic dump_info/2, ran/2, capture_policy/2.

dump_info(file, 'dump1.dmp').
dump_info(stream, unloaded_modules).
ran('.exr -1', ok).
ran(lm, ok).
capture_policy(exception_record/3, []).
capture_policy(unloaded_module/2,
               [stream(unloaded_modules)]).

% missing(Need): the recorded capture does not
% meet Need, a stream need; any other kind of
% need is an error.
missing(stream(S)) :-
    \+ dump_info(stream, S).
missing(Need) :-
    Need \= stream(_),
    domain_error(capture_need, Need).

% coverage(Relation, Status): complete when a
% policy is declared, a capture is recorded, the
% command ran and succeeded, and every need is
% met; incomplete(Missing) when needs are unmet or
% the command failed; unknown when the policy,
% the capture, or the command's run is not
% recorded.
coverage(Relation, Status) :-
    (   capture_policy(Relation, Needs),
        dump_info(_, _),
        source(Relation, Command, _),
        ran(Command, Outcome)
    ->  findall(Need,
                ( member(Need, Needs),
                  missing(Need) ),
                Unmet),
        (   Outcome == ok
        ->  Missing = Unmet
        ;   Missing = [ran(Command, Outcome)
                       | Unmet]
        ),
        (   Missing == []
        ->  Status = complete
        ;   Status = incomplete(Missing)
        )
    ;   Status = unknown
    ).
```

```
% absence(Relation, Status): Status is present
% when facts of Relation are loaded; absent when
% there are none, the command that produces them
% was run, and the coverage of Relation is
% complete; and unknown otherwise.
absence(F/A, Status) :-
    functor(Goal, F, A),
    (   current_predicate(F/A),
        call(Goal)
    ->  Status = present
    ;   source(F/A, _, _),
        coverage(F/A, complete)
    ->  Status = absent
    ;   Status = unknown
    ).

?- absence(exception_record/3, S).
S = present.

?- absence(unloaded_module/2, S).
S = absent.

?- absence(mutant/3, S).
S = unknown.
```

The dump has an exception record, so a **Manual Dump (Process)** (page 451) recognizer fails on it for the right reason. The relation `unloaded_module/2`, an extension for the list of unloaded modules that `lm` prints, has no facts, a recorded source, and complete coverage: the command was run, and the dump holds the unloaded module list stream, which the policy says the command needs to list them; so the absence is a fact about the dump. The relation `mutant/3`, which Chapter 20 introduces for mutex objects, has neither facts nor a source nor a policy, and nothing follows from its absence. `current_predicate/1` guards the call, since calling an undefined predicate is an existence error, not a failure, under SWI-Prolog's default `unknown` flag.

`absence/2` answers `absent` only through `coverage/2`, and the following checks of boundary cases take away one record at a time and restore it, so that `absent` turns into `unknown` while the source is still there:

```
?- retract(dump_info(stream, unloaded_modules)),
   absence(unloaded_module/2, S),
   coverage(unloaded_module/2, C).
S = unknown,
C = incomplete([stream(unloaded_modules)]).

?- assertz(dump_info(stream, unloaded_modules)),
   absence(unloaded_module/2, S).
S = absent.

?- retract(ran(lm, ok)),
   absence(unloaded_module/2, S),
   coverage(unloaded_module/2, C).
S = C, C = unknown.
```

```
?- assertz(ran(lm, ok)),
   retract(capture_policy(unloaded_module/2, P)),
   absence(unloaded_module/2, S).
P = [stream(unloaded_modules)],
S = unknown.

?- assertz(capture_policy(unloaded_module/2,
       [stream(unloaded_modules)])),
   absence(unloaded_module/2, S).
S = absent.
```

Without the stream, `lm` on this dump would have listed no unloaded modules whether or not the process had any, so the coverage is incomplete and the absence is unknown; with the command not recorded as run, or with no policy declared for the relation, the coverage itself is unknown, and so is the absence. In each case `source/3` still names the command, which is why `source/3` alone was not enough. Chapter 15's `thread_absence/3` asks the same question of the facts about one thread, for relations such as `last_error/2` whose command is run per thread.

The caveat is **Truncated Dump** (page 452), which Chapter 23 also treats: a dump that was cut short, by a disk filling or a writer that stopped, has stacks, modules, or whole threads missing, and every command was run, every relation has a source, and the coverage is complete by every recorded measure, yet facts are absent that the process had. No knowledge base can detect its own truncation from inside; the signs are external, a file shorter than the debugger expected, or internal inconsistencies of the kind `check_schema/0` in Chapter 15 reports, a thread with no frames, a frame in no module. A recognizer that concludes from absence states its assumption, and a report that prints such a conclusion prints the assumption with it. In this book, every negated goal in a recognizer is read as "no such fact is loaded", and the pattern sections say when that is enough.

17.9 Determinism

A recognizer's **determinism** is part of its contract, and the toplevel shows it. The test form is semideterministic: `true.` or `false.`, never `true ;`. The enumerator produces each instance once and ends with `false.` or with the last instance and `.`; a duplicate instance is a bug, and so is an instance that depends on the order of facts in the file.

Three devices give a recognizer these properties. Indexing gives the test form its determinism when the instance is the first argument of a fact the body looks up first, which is why `spiking_thread/2` begins with `thread_time/3` and `blocked_thread/2` with `waits_for/3`. If-then-else, as in `chain_from/4`, chooses without leaving a choice point where two clauses with complementary tests would leave one. And aggregation contains nondeterminism: `frame_regularity/2` needs the longest run of a thread's trace, which is a search over runs, and `aggregate_all(max(C, R), ...)` runs the search to completion and produces one answer.

Where a recognizer is nondeterministic by nature, because the same instance can be reached by several derivations, `distinct/1` from `library(solution_sequences)` removes the duplicates, at the cost of remembering every instance produced so far, and the recognizer says so in its comment.

And where a caller needs the test form of a recognizer that was written as an enumerator, `once/1` makes it semideterministic from outside, as `holds/1` does for the goals in an evidence list.

```
?- aggregate_all(count, spiking_thread(_), N).
N = 2.

?- aggregate_all(count, wait_chain(_), N).
N = 3.

?- aggregate_all(count, deadlock(_), N).
N = 1.

?- once(frame_regularity(T)).
T = 107.
```

Counting instances with `aggregate_all(count, ...)` is the way to check an enumerator for duplicates against a count known by other means, and the test suite of the next section does it for every recognizer of the chapter.

17.10 Testing Recognizers

A recognizer is a program, and `library(plunit)` from Chapter 13 tests it. A test unit for the recognizers of this chapter states, for the shared dump, what each must find, what each must not find, and that every evidence term holds. The tests are the chapter's claims about `dump1.pl` written down so that a change to a recognizer, or to a threshold, that breaks a claim is noticed.

```
:- use_module(library(plunit)).

:- begin_tests(dump1_patterns).

test(spiking, all(T == [102, 107])) :-
    spiking_thread(T).

test(not_spiking, fail) :-
    spiking_thread(100).

test(deadlock, [nondet, true(P == 103-104)]) :-
    deadlock(P).

test(null_pointer, T == 109) :-
    null_pointer_data(T).

test(chains, all(C == [[101, 103, 104, 103],
                       [103, 104, 103],
                       [104, 103, 104]])) :-
    wait_chain(C).

test(instance_count, N == 16) :-
    pattern_instances(Es),
    length(Es, N).
```

```
test(evidence_ground) :-
    pattern_instances(Es),
    maplist(ground, Es).

test(evidence_holds) :-
    pattern_instances(Es),
    maplist(holds, Es).

test(deadlock_entailed, [nondet]) :-
    deadlock(T1-T2, E),
    leaves(E, Gs),
    entails(Gs,
            ( waits_for(T1, critical_section, A1),
              critical_section(A1, T2, _, _),
              waits_for(T2, critical_section, A2),
              critical_section(A2, T1, _, _) )).

:- end_tests(dump1_patterns).
```

`begin_tests/1` and `end_tests/1` bracket a unit, and each `test/2` clause has a name, an option, and a body. `all(T == [...])` collects every solution of the body and compares the list; `fail` requires the body to fail; `true(P == 103-104)` checks the binding after the first solution, and `nondet` says that a choice point after it is expected, which it is for `deadlock/1`, whose enumerator still has facts of `waits_for/3` to try; a test without `nondet` whose body leaves a choice point is reported as a warning, which is how the suite enforces Section 17.9. A test with no option, like `evidence_holds`, passes when its body succeeds. The last three tests are the three parts of the evidence contract for the chapter's instances: ground, holding, and, for the deadlock, sufficient.

```
?- set_prolog_flag(verbose, normal).
true.

?- run_tests.
% PL-Unit: dump1_patterns ......... passed <...>
% All 9 tests passed
true.
```

`run_tests/0` runs every unit and reports. The toplevel of this book was started with `-q`, which suppresses informational messages, and the `verbose` flag is set to `normal` so that the summary is printed; the dots are one test each. A failing test prints what it expected and what it got:

```
:- begin_tests(wrong_claims).

test(main_thread_spikes) :-
    spiking_thread(100).

test(share, true(S =:= 50)) :-
    cpu_share(102, S).

:- end_tests(wrong_claims).
```

```
?- run_tests(wrong_claims).
% PL-Unit: wrong_claims
ERROR: <...>
<...>test main_thread_spikes: failed
ERROR: <...>
<...>test share: wrong answer (compared using =:=)
ERROR:     Expected: 50
ERROR:     Got:      68.48249027237354
<...>
% 2 tests failed
% 0 tests passed
false.
```

The two claims are false of the shared dump, and the report says so with the file and line of each test, elided here, and with the value that was found. `run_tests/1` runs one unit by name and fails when any test in it fails, so that a script can use the result as an exit code, as Chapter 13 showed.

Every pattern chapter of Parts III and IV ends its predicates with claims of this kind about the shared data, and the suite over all of them is the regression test of the book.

17.11 Conventions for Parts III and IV

Each pattern section of Parts III and IV follows one shape, and this section states it as the contract the reader can hold the following chapters to.

A section is headed by the pattern's catalog name and opens with a definition paragraph that gives the pattern's intent in this book's words, naming the artifact it is found in and what an instance of it is. The recognizer follows, named by the snake_case of the catalog name with the variant dropped, or with the variant as its first argument where one chapter treats several variants, in the enumerator form of Section 17.2, and in the explainer form `p(?Instance, -Evidence)` where the evidence is what the analyst needs to see; every threshold is a `threshold/2` fact, every helper is named as a helper, and every instance is a ground term of the kind Section 17.2 described. A transcript then runs the recognizer over `dump1.pl` or `trace1.pl`, and where the shared data has no instance, a small in-chapter extension supplies one and is marked as such. The section ends with the limits of the formalization: what the predicate does not capture of the description, under which assumptions its negations are sound, and what the analyst must still judge.

The evidence term `evidence(Pattern, Instance, Facts)` is the interface between a recognizer and everything downstream: the report of Chapter 23, the meta-interpreters of Chapter 29, and the tests of each chapter. Its counterpart for what a recognizer can only suspect is the suspicion term `suspected(Pattern, Instance, Grounds)`, and both are under the contract of Section 17.3: ground goals that hold, nothing that is not a goal, and a list from which the shape re-derives. The catalog facts `pattern/3`, with the artifact kinds `memory_dump` and `software_trace`, are restated by the chapters of Part III for the patterns they treat, since each chapter is a program of its own, and Chapter 23 builds the report of a whole dump on this chapter's form of `pattern_instances/1` and `report/0`, over the recognizers of Chapters 18 to 22 collected in one module.

Part III. Memory Dump Analysis Patterns

Part III treats the memory analysis patterns that a process memory dump rendered into the schema of Chapter 15 can exhibit, one section per pattern, in six chapters: stack traces and threads, exceptions and faults, synchronization, memory and resources, modules and environment, and dump quality with the composition of patterns into a report. Each section gives the pattern's intent, a recognizer over `dump1.pl` or over a stated extension of it, the transcript of the recognizer, and the limits of the formalization. The patterns whose evidence lies outside what the schema records are treated in prose and marked as such; the catalogs remain the authority on their descriptions.

Chapter 18. Stack Traces and Threads

The memory analysis patterns of this chapter concern the stack traces of a process memory dump and the threads they belong to: the trace of one thread, the collection of all of them, the kinds of thread an analyst sets apart before reading the rest, the comparisons between traces, and the ways a trace can mislead. Every predicate is written over the schema of Chapter 15 and run on `dump1.pl`. Each section gives the pattern's intent, the predicate that recognizes its instances, a transcript, and what the predicate does not capture.

The chapter starts a fresh toplevel, consults the dump, and defines the helpers it needs, several of which earlier chapters built up step by step. Where a pattern needs data the shared schema does not record, a small extension is defined as facts under a name of its own. The thresholds are `threshold/2` facts, as Chapter 17 prescribes, and the patterns treated with a predicate are registered as `pattern/3` facts for the report of Chapter 23.

```
:- [dump1].
:- use_module(library(pprint)).

% threshold(Name, Value): the thresholds of this
% chapter, as facts an analyst can inspect.
threshold(spiking_share, 25).
threshold(frame_run, 4).
threshold(set_size, 2).

% pattern(Name, Artifact, Chapter): the patterns
% this chapter recognizes with a predicate.
pattern('Stack Trace', memory_dump, 18).
pattern(
    'Stack Trace Collection (Unmanaged Space)',
    memory_dump, 18).
pattern('Stack Trace Collection (Predicate)',
        memory_dump, 18).
pattern('Main Thread', memory_dump, 18).
pattern('Special Thread', memory_dump, 18).
pattern('Passive Thread', memory_dump, 18).
pattern('Spiking Thread', memory_dump, 18).
pattern('Blocked Thread (Software)',
    memory_dump, 18).
pattern('Active Thread', memory_dump, 18).
pattern('Stack Trace Signature', memory_dump, 18).
pattern('Least Common Frame', memory_dump, 18).
pattern('Stack Trace Set', memory_dump, 18).
pattern('Frame Regularity', memory_dump, 18).
pattern('Caller-n-Callee', memory_dump, 18).
pattern('Procedure Call Chain', memory_dump, 18).
pattern('Incorrect Stack Trace', memory_dump, 18).
pattern('Truncated Stack Trace', memory_dump, 18).
pattern('Coincidental Symbolic Information',
        memory_dump, 18).
pattern('Glued Stack Trace', memory_dump, 18).
pattern('Execution Residue (User Space)',
        memory_dump, 18).
```

```
% show(Term): print Term over several lines that
% fit the width of this book's pages.
show(Term) :-
    print_term(Term, [right_margin(46),
                      tab_width(0)]),
    nl.
```

`print_term/2` from `library(pprint)` prints a term with line breaks and indentation within a right margin; the toplevel prints every answer on one line, however long, and lists of frames do not fit on a pocket page. `show/1` is used wherever a transcript would otherwise run off the page; it prints arguments without the space after the comma that the toplevel inserts. The flag `toplevel_print_anon` is set to `false` for the whole chapter, as in Chapter 9, so that variables beginning with an underscore are bound without being printed:

```
?- set_prolog_flag(toplevel_print_anon, false).
true.
```

18.1 Stack Trace

A **stack trace** is the sequence of frames of one thread, read from the frame that was executing when the dump was written down to the frame that started the thread; each frame names a function in progress, and the sequence is the record of the calls that had not yet returned. It is the first thing an analyst reads about any thread. The trace is a reconstruction the debugger makes from the thread's stack memory and the symbol files it has, not a record the process kept, and the last sections of this chapter say when the reconstruction is to be doubted.

The predicate is the `findall/3` form of Chapter 9, with the frames ordered by number so that the result does not depend on the order of the facts in the file, and a printer that renders each frame in the debugger's `module!function+0xoffset` notation of Chapter 8:

```
% stack_trace(TID, Frames): Frames is the list of
% frame/5 terms of thread TID, top frame first.
stack_trace(TID, Frames) :-
    findall(frame(TID, N, M, F, O),
            frame(TID, N, M, F, O), Frames0),
    sort(2, @=<, Frames0, Frames).

% frame_symbol(Frame, Symbol): Symbol is the
% debugger's text for the frame/5 term Frame.
frame_symbol(frame(_, _, M, F, O), Symbol) :-
    (   F == 'Unknown'
    ->  format(atom(Symbol), "~a+0x~16r", [M, O])
    ;   format(atom(Symbol), "~a!~a+0x~16r",
               [M, F, O])
    ).
```

```
% print_stack_trace(TID): print the stack trace of
% TID one frame per line, numbered as k does.
print_stack_trace(TID) :-
    stack_trace(TID, Frames),
    forall(member(Frame, Frames),
           ( Frame = frame(_, N, _, _, _),
             frame_symbol(Frame, Symbol),
             format("~`0t~d~2| ~a~n", [N, Symbol])
           )).

?- stack_trace(103, _Frames), length(_Frames, N).
N = 10.

?- print_stack_trace(103).
00 ntdll!NtWaitForAlertByThreadId+0x14
01 ntdll!RtlpWaitOnCriticalSection+0x1cf
02 ntdll!RtlpEnterCriticalSectionContended+0x1cd
03 ntdll!RtlEnterCriticalSection+0x40
04 dbclient!Connection::Send+0x58
05 appserver!Session::Commit+0x9e
06 appserver!Worker::Run+0x120
07 ucrtbase!thread_start<...>+0x42
08 kernel32!BaseThreadInitThunk+0x14
09 ntdll!RtlUserThreadStart+0x21
true.

?- stack_trace(999, Frames).
Frames = [].
```

The mode is `+TID`; with the identifier unbound the collection would merge every thread's frames into one list. The trace of thread 103 reads, from the bottom, as a worker that entered `Session::Commit`, called into the database client, and is waiting to enter a critical section; Chapter 20 says which one and why it will not get it. A thread absent from the dump has the empty trace, which is the right answer to a question about frames and a wrong one to a question about threads; `thread/2` answers the latter.

What the predicate does not capture is what the facts do not record: the return addresses and stack pointers that `k` prints in its first columns, and the frame parameters. A stack trace in this book is the symbolic column of `k`, and Sections 18.17 and 18.18 say what follows from that for its correctness.

18.2 Stack Trace Collection (Unmanaged Space)

The **Stack Trace Collection (Unmanaged Space)** is the set of stack traces of all threads of a process, taken as one object of study: the analyst reads it as `~*k` prints it, looking across the threads for the ones that differ from the rest. The variant name says that the frames are native code; the managed variant needs the runtime's own stack walker and is not treated here.

```
% stack_trace_collection(Collection): Collection
% is the list of TID-Frames pairs, one per thread.
stack_trace_collection(Collection) :-
    findall(TID-Frames,
            ( thread(TID, _),
              stack_trace(TID, Frames)
            ),
            Collection).

% print_top_frames(Collection): one line per
% thread with the symbol of its top frame.
print_top_frames(Collection) :-
    forall(member(TID-[Top | _], Collection),
           ( frame_symbol(Top, Symbol),
             format("~d ~a~n", [TID, Symbol])
           )).

?- stack_trace_collection(_C), length(_C, N).
N = 10.

?- stack_trace_collection(_C),
   print_top_frames(_C).
100 ntdll!NtWaitForMultipleObjects+0x14
101 ntdll!NtWaitForAlertByThreadId+0x14
102 reportengine!Table::Find+0x3e
103 ntdll!NtWaitForAlertByThreadId+0x14
104 ntdll!NtWaitForAlertByThreadId+0x14
105 ntdll!NtWaitForWorkViaWorkerFactory+0x14
106 ntdll!NtWaitForWorkViaWorkerFactory+0x14
107 reportengine!Node::Visit+0x12
108 ntdll!NtDelayExecution+0x14
109 dbclient!Buffer::Copy+0x1a
true.
```

The top frames alone sort the ten threads into the waiting majority in `ntdll` and three exceptions: two threads executing in `reportengine` and one stopped in `dbclient`. The rest of the chapter formalizes what an analyst does next with such a list. The `member/2` goal in `print_top_frames/1` unifies each element with `TID-[Top | _]` and so skips a thread with no frames; such a thread is itself a finding, and Section 18.18 returns to it.

The limit is the one Chapter 9 noted: the collection is a list, and the pattern is a way of reading it. The comparisons that make the list informative are the patterns of Sections 18.11 to 18.16.

18.3 Stack Trace Collection (Predicate)

The **Stack Trace Collection (Predicate)** variant restricts the collection to the threads that satisfy a condition: the threads whose trace passes through a given module, that call a given function, that are deeper than some bound. In a debugger this is done with scripting over threads; in Prolog the condition is a predicate on the frame list, passed as a closure, and the restriction is `include/3` of Chapter 10:

```
% stack_trace_collection(Pred, Collection): the
% stack traces of the threads whose frame list
% satisfies the closure Pred.
stack_trace_collection(Pred, Collection) :-
    stack_trace_collection(All),
    include([_-Frames]>>call(Pred, Frames), All,
            Collection).

% has_module(Module, Frames): some frame of the
% list Frames is in Module.
has_module(Module, Frames) :-
    memberchk(frame(_, _, Module, _, _), Frames).

% has_function(F, Frames): some frame of the list
% Frames names the function F.
has_function(F, Frames) :-
    memberchk(frame(_, _, _, F, _), Frames).

?- stack_trace_collection(has_module(dbclient),
   _C), pairs_keys(_C, TIDs).
TIDs = [103, 104, 109].

?- stack_trace_collection(
   has_function('Worker::Run'), _C),
   pairs_keys(_C, TIDs).
TIDs = [101, 102, 103, 107, 109].

?- stack_trace_collection(
   [Fs]>>(length(Fs, D), D > 9), _C),
   pairs_keys(_C, TIDs).
TIDs = [101, 103, 104, 107].
```

The two helpers are the common conditions, and the third query shows a lambda written in the query. `memberchk/2` is `member/2` committed to the first match, which is right for a test and leaves no choice point. The three threads in the database client are the two halves of the deadlock of Chapter 20 and the faulting thread; the five in `Worker::Run` are the application's worker pool; the four deepest threads are the three blocked ones and the recursion of thread 107.

What the variant does not decide is which condition to ask. The condition is the analyst's hypothesis; the predicate makes testing it cheap enough to try several.

18.4 Main Thread

The **Main Thread** is the thread that entered the process's entry point and runs the message loop or the main procedure of the program; its stack trace reaches `main` or one of its variants, and its state says most about whether the process as a whole was responsive. The entry functions are facts, since what counts as one depends on the runtime:

```
% entry_function(F): F is an entry function that
% the main thread's stack trace passes through.
entry_function(main).
entry_function(wmain).
entry_function('WinMain').
entry_function('wWinMain').

% main_thread(TID): TID is the main thread.
main_thread(TID) :-
    thread(TID, _),
    once(( frame(TID, _, _, F, _),
           entry_function(F)
         )).

?- main_thread(T).
T = 100 ;
false.

?- main_thread(T), print_stack_trace(T).
00 ntdll!NtWaitForMultipleObjects+0x14
01 kernelbase!WaitForMultipleObjectsEx+0xe9
02 appserver!MainLoop+0x1b2
03 appserver!wmain+0x55
04 appserver!invoke_main+0x22
05 kernel32!BaseThreadInitThunk+0x14
06 ntdll!RtlUserThreadStart+0x21
T = 100 ;
false.
```

The `once/1` makes the test for one thread deterministic even if its trace named two entry functions; the choice point the toplevel reports belongs to `thread/2`. The main thread of `AppServer.exe` is in its message loop, waiting on several objects with no record of what they are, which is a healthy state for a main thread: whatever went wrong happened on the workers.

The predicate finds the main thread by its trace. It fails on a process whose main thread has exited while other threads keep the process alive, and on a dump whose main thread's trace is truncated above the entry function. The debugger's convention that the first thread listed is the main thread is a different criterion, and the schema does not record the listing order.

18.5 Special Thread

A **Special Thread** is one the system or the runtime created for its own purposes, with a stack trace that identifies the role: a thread pool worker waiting for work, the thread a debugger injects to break into a process. They are recognized so that they can be set aside; a dump of a hung process may have dozens, and none is the cause. The marker functions are facts with the role they mark:

```
% special_frame(Function, Role): a frame naming
% Function marks a thread as serving Role.
special_frame('TppWorkerThread', thread_pool).
special_frame('DbgUiRemoteBreakin', debugger).
```

```
% special_thread(TID, Role): TID is a thread of
% the system or runtime serving Role.
special_thread(TID, Role) :-
    thread(TID, _),
    frame(TID, _, _, F, _),
    special_frame(F, Role).

?- special_thread(T, R).
T = 105,
R = thread_pool ;
T = 106,
R = thread_pool ;
false.
```

The two idle pool threads that Chapter 7 classed as `pool_idle` are the dump's special threads. The list of markers is short and tied to one operating system's naming. A thread pool implemented by the application itself, as `Worker::Run` suggests `AppServer.exe` has, is not special in this sense: its threads do the application's work, and their traces are the ones to read.

18.6 Passive Thread

A **Passive Thread** waits for something the dump does not identify, is blocked by no object the analyst can trace, and has nothing to show for itself: the normal state of most threads in most dumps, named so that the others stand out. The schema's `thread_state/2` and `waits_for/3` give the test directly:

```
% passive_thread(TID): TID waits on a user request
% and is not recorded as waiting for an object.
passive_thread(TID) :-
    thread_state(TID, waiting(user_request)),
    \+ waits_for(TID, _, _).

?- passive_thread(T).
T = 100 ;
T = 105 ;
T = 106.

?- passive_thread(T), \+ special_thread(T, _).
T = 100 ;
false.
```

The main thread and the two pool threads are passive, and the main thread is the only passive thread that is not also special. The negation `\+ waits_for(TID, _, _)` succeeds iff no `waits_for/3` fact mentions the thread; under the closed-world reading this means "the dump records no object it waits for", not "it waits for nothing": thread 100 waits for several objects in `WaitForMultipleObjectsEx`, and the schema has no fact for them because `!cs` reports critical sections only. A `waits_for/3` relation extended with the objects of `!handle` would move thread 100 out of the pattern; Chapter 20 extends it that way for mutexes and threads.

18.7 Spiking Thread

A **Spiking Thread** has consumed a share of the process's processor time out of proportion to the other threads; it is what `!runaway` shows at the top of its list, and in a process that is not supposed to be computing it is a thread caught in a loop. Chapters 6 and 9 developed the recognizer; it is repeated with the threshold as a `threshold/2` fact and with the explainer form of Chapter 17, whose evidence term names the facts the recognition rests on:

```
% cpu_time(TID, Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% total_cpu(Total): the CPU time of all threads.
total_cpu(Total) :-
    aggregate_all(sum(Ms), cpu_time(_, Ms),
                  Total).

% cpu_share(TID, Share): the CPU time of TID as a
% percentage of the total.
cpu_share(TID, Share) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Total > 0,
    Share is 100.0 * Ms / Total.

% spiking_thread(TID): the CPU share of TID is at
% least the threshold.
spiking_thread(TID) :-
    cpu_share(TID, Share),
    threshold(spiking_share, Min),
    Share >= Min.

% spiking_thread(TID, Evidence): as the test
% above, with the evidence of the recognition.
spiking_thread(TID, Evidence) :-
    spiking_thread(TID),
    cpu_time(TID, Ms),
    total_cpu(Total),
    threshold(spiking_share, Min),
    Evidence = evidence('Spiking Thread', TID,
                        [ cpu_time(TID, Ms),
                          total_cpu(Total),
                          threshold(spiking_share,
                                    Min)
                        ]).

?- spiking_thread(T).
T = 102 ;
T = 107 ;
false.
```

```
?- spiking_thread(107, _E), show(_E).
evidence('Spiking Thread',
         107,
         [ cpu_time(107,4100),
           total_cpu(14135),
           threshold(spiking_share,25)
         ])
true.

?- spiking_thread(100, E).
false.
```

The guard `Total > 0` makes the predicate fail rather than raise an evaluation error on a dump taken before any thread has used time. The evidence lists the thread's time, the total, and the threshold, which is everything a reader needs to recompute the share and to disagree with the threshold; the share is derived and left out. Chapter 6 said what the threshold does not capture: 29 percent is a spike next to eight idle threads and would not be in a process with three busy workers. And a thread that spun for a minute an hour ago has the same CPU time as one spinning now; the facts do not date the time.

18.8 Blocked Thread (Software)

A **Blocked Thread (Software)** is stopped by a synchronization object under software control, a critical section, a mutex, an event, and will run again only when another thread acts; the variant name distinguishes it from a thread blocked on hardware and from one whose wait has a timeout. The `waits_for/3` relation is the schema's record of `!cs` and `!locks` output, and the stack trace is the independent witness, since the wait path through `RtlEnterCriticalSection` is visible in the frames:

```
% blocked_thread(TID): the dump records an object
% TID waits for.
blocked_thread(TID) :-
    waits_for(TID, _, _).

% wait_function(F): F is a frame of the critical
% section wait path in ntdll.
wait_function('RtlpWaitOnCriticalSection').
wait_function('RtlEnterCriticalSection').

% blocked_thread(TID, Evidence): as
% blocked_thread/1, with the object and the wait
% frames as evidence.
blocked_thread(TID, Evidence) :-
    waits_for(TID, Kind, Addr),
    findall(frame(TID, N, M, F, O),
            ( frame(TID, N, M, F, O),
              wait_function(F)
            ),
            Frames),
    Facts = [waits_for(TID, Kind, Addr) | Frames],
    Name = 'Blocked Thread (Software)',
    Evidence = evidence(Name, TID, Facts).
```

```
?- blocked_thread(T).
T = 101 ;
T = 103 ;
T = 104.

?- blocked_thread(104, _E), show(_E).
evidence('Blocked Thread (Software)',
         104,
         [ waits_for(104,
                     critical_section,
                     140695026507968),
           frame(104,
                 1,
                 ntdll,
                 'RtlpWaitOnCriticalSection',
                 463),
           frame(104,
                 3,
                 ntdll,
                 'RtlEnterCriticalSection',
                 64)
         ])
true.

?- forall(blocked_thread(T),
   frame(T, _, ntdll,
         'RtlpWaitOnCriticalSection', _)).
true.
```

The three blocked threads are the ones Chapter 7 classed as `blocked`, and the last query is the consistency check between the two sources: every thread the object facts call blocked has the wait path on its stack. The reverse check, that every thread with the wait path has an object fact, holds here too. `print_term/2` prints integers in decimal, as the toplevel does; the address is `0x7ff6_1d13_a0c0`, the first critical section of the dump.

The predicate says that a thread is blocked and on what, and no more. Whether the owner will ever release the object and whether the chain of owners closes into a cycle are the patterns of Chapter 20, each of which begins with this one.

18.9 Active Thread

An **Active Thread** was executing instructions when the dump was written: its state is running, and its top frame is in application or library code rather than in a wait system call. It is the complement of the waiting majority, and a dump of a responsive process may have none.

```
% active_thread(TID): TID was running when the
% dump was written.
active_thread(TID) :-
    thread_state(TID, running).

?- active_thread(T).
T = 102 ;
```

```
T = 107.

?- active_thread(T), spiking_thread(T).
T = 102 ;
T = 107.

?- thread_state(109, S).
S = waiting(exception).
```

The two active threads are the two spiking threads, which is usual in a process busy for no good reason and not necessary: a thread can be caught running an ordinary instruction once, and a spiking thread can be in a wait between two bursts. Thread 109 was executing the instruction that faulted, and the dump records it as waiting on its exception, because by the time the dump was written the exception had been raised and the thread handed to the debugger. Whether such a thread counts as active is a matter of the dump's timing; the schema follows the debugger's state, and Chapter 19 treats it as the faulting thread, which is the more informative description.

18.10 Thread Age

Thread Age is the time a thread has existed, from its creation to the moment of the dump; the pattern distinguishes the long-lived threads of a process, its main thread and its workers, from threads created recently, which in a dump of a hang are often the ones created in response to the problem, such as pool threads spawned because the existing ones were blocked. The debugger shows creation and elapsed times alongside CPU times, and a recognizer would compare each thread's elapsed time with the process's own.

The schema has no fact for it. `thread_time/3` records user and kernel time, which measure what a thread did and not how long it has existed; a thread idle for an hour has the same fact as one created a second ago. An extension `thread_created/2` with the creation time and `dump_time/1` with the time of the dump would make the age a subtraction and the pattern a threshold on it, in the manner of `spiking_thread/1`. What no extension recovers is the thread that had already exited; a process that creates and destroys threads at a high rate shows that only in its thread identifiers, which the operating system reuses and which are therefore no evidence of age either.

18.11 Stack Trace Signature

A **Stack Trace Signature** is a reduced form of a stack trace that keeps what identifies it and drops what varies between instances: the module and function names in order, without offsets, addresses, or parameters. Two threads with the same signature are doing the same thing, and a signature seen in one dump can be looked for in the next; it is the form in which traces are compared, counted, and remembered. The predicate maps each frame to its `Module-Function` pair:

```
% frame_key(Frame, Key): Key is the pair
% Module-Function of the frame/5 term Frame.
frame_key(frame(_, _, M, F, _), M-F).
```

```
% stack_trace_signature(TID, Signature): the keys
% of the frames of TID, top frame first.
stack_trace_signature(TID, Signature) :-
    stack_trace(TID, Frames),
    maplist(frame_key, Frames, Signature).

% top_signature(TID, Depth, Signature): the keys
% of the top Depth frames of TID.
top_signature(TID, Depth, Signature) :-
    stack_trace_signature(TID, Full),
    length(Signature, Depth),
    append(Signature, _, Full).

?- stack_trace_signature(105, _S), show(_S).
[ ntdll-'NtWaitForWorkViaWorkerFactory',
  ntdll-'TppWorkerThread',
  kernel32-'BaseThreadInitThunk',
  ntdll-'RtlUserThreadStart'
]
true.

?- stack_trace_signature(105, _A),
   stack_trace_signature(106, _B), _A == _B.
true.

?- top_signature(109, 1, S).
S = [dbclient-'Buffer::Copy'].

?- top_signature(105, 6, S).
false.
```

`maplist/3` applies `frame_key/2` to each frame; the test `_A == _B` is syntactic identity of the two signatures, which is what equality of signatures means. `top_signature/3` keeps the first `Depth` keys and fails on a shallower trace, because `length/2` fixes the length of the prefix before `append/3` finds it; a signature of the top few frames is coarser and groups more threads, as Section 18.13 uses.

A signature is as reliable as the names in it. Two threads in a module without symbols have the same `Module-'Unknown'` key for different code, and two threads at different offsets of one function get the same key by design; the choice of what to drop is the choice of what to call the same, and the predicate fixes one such choice.

18.12 Least Common Frame

The **Least Common Frame** of several stack traces is the frame nearest the top that they all share, with everything below it shared as well: the point from which the threads diverged, as the least common ancestor of several nodes in a tree is the deepest node above all of them. For the workers of a server it is the dispatch point; for a set of blocked threads, the function that took the lock.

The common part of several traces is a common suffix of their frame lists, since traces agree at the bottom and differ at the top; reversing the signatures turns it into a common prefix, and the least common frame is the last element of that prefix:

```
% reversed_signature(TID, Reversed): the signature
% of TID from the bottom frame up.
reversed_signature(TID, Reversed) :-
    stack_trace_signature(TID, Signature),
    reverse(Signature, Reversed).

% common_prefix(Lists, Prefix): Prefix is the
% longest list that every list in Lists begins
% with.
common_prefix(Lists, Prefix) :-
    (   Lists = [[X | _] | _],
        forall(member(L, Lists), L = [X | _])
    ->  maplist(list_tail, Lists, Tails),
        common_prefix(Tails, Prefix0),
        Prefix = [X | Prefix0]
    ;   Prefix = []
    ).

list_tail([_ | Tail], Tail).

% least_common_frame(TIDs, Key): Key is the
% topmost frame key shared, as a common suffix, by
% the stack traces of all threads in TIDs.
least_common_frame(TIDs, Key) :-
    maplist(reversed_signature, TIDs, Reversed),
    common_prefix(Reversed, Common),
    last(Common, Key).

?- least_common_frame([101, 103], K).
K = appserver-'Worker::Run'.

?- least_common_frame([101, 103, 104], K).
K = ucrtbase-'thread_start<...>'.

?- least_common_frame([100, 105], K).
K = kernel32-'BaseThreadInitThunk'.

?- findall(T, thread(T, _), _Ts),
   least_common_frame(_Ts, K).
K = kernel32-'BaseThreadInitThunk'.
```

`common_prefix/2` takes the head of the first list, checks with `forall/2` that every list begins with it, and recurses on the tails; when some list is empty or begins differently the condition fails and the prefix ends. Threads 101 and 103 parted in `Worker::Run`, which dispatched one to a lookup and the other to a commit; adding thread 104, the database client's reader thread rather than a worker, moves the common frame down to the runtime's thread start; and the whole process shares only the two system frames at the bottom, which is the right answer for a process whose threads have different origins.

The predicate compares keys, so two threads in `Worker::Run` at different offsets share the frame, as intended, and it fails when the traces share nothing, as a glued or truncated trace among them would cause. It does not know that `Worker::Run` is a dispatch point; the analyst reads that from the name.

18.13 Stack Trace Set

A **Stack Trace Set** is a group of threads with the same stack trace, or the same trace to a chosen depth, read as one: twenty threads waiting in the same function are one fact about the process, and the size of the set is the fact. The grouping is `bagof/3` with the signature as the free variable, and the threshold says how many threads make a set worth reporting:

```
% stack_trace_set(Signature, TIDs): TIDs are the
% threads whose signature is Signature, at least
% the threshold many.
stack_trace_set(Sig, TIDs) :-
    bagof(TID,
          P^( thread(TID, P),
              stack_trace_signature(TID, Sig)
            ),
          TIDs),
    threshold(set_size, Min),
    length(TIDs, Size),
    Size >= Min.

% stack_trace_set(Depth, Signature, TIDs): as
% stack_trace_set/2 on the top Depth frames.
stack_trace_set(Depth, Sig, TIDs) :-
    bagof(TID,
          P^( thread(TID, P),
              top_signature(TID, Depth, Sig)
            ),
          TIDs),
    threshold(set_size, Min),
    length(TIDs, Size),
    Size >= Min.

?- stack_trace_set(_S, TIDs).
TIDs = [105, 106] ;
false.

?- stack_trace_set(4, _S, TIDs).
TIDs = [101, 103, 104] ;
TIDs = [105, 106] ;
false.

?- stack_trace_set(4, _S, TIDs), show(_S).
[ ntdll-'NtWaitForAlertByThreadId',
  ntdll-'RtlpWaitOnCriticalSection',
  ntdll-'RtlpEnterCriticalSectionContended',
  ntdll-'RtlEnterCriticalSection'
]
TIDs = [101, 103, 104] ;
[ ntdll-'NtWaitForWorkViaWorkerFactory',
  ntdll-'TppWorkerThread',
  kernel32-'BaseThreadInitThunk',
  ntdll-'RtlUserThreadStart'
]
TIDs = [105, 106] ;
false.
```

The thread enumeration goes inside the `bagof/3` goal so that `stack_trace_signature/2` is called with the identifier bound, and the process identifier is quantified away with `^`. With full signatures the only set is the pair of idle pool threads; with the top four frames the three threads blocked on critical sections form a set as well, since the wait path through `ntdll` is the same for all of them. That grouping is what an analyst sees in `~*k`: three threads stopped in the same four frames, which is the **Blocked Queue** (page 396) of Chapter 20 when they wait for the same object and a coincidence when they do not. Here two wait for the same object and the third for another.

The threshold of two suits a dump of ten threads; a server with hundreds wants a higher one. What the predicate does not capture is similarity short of equality: two traces that differ in one frame of a hundred are different signatures, and a measure of distance between traces is a different formalization.

18.14 Frame Regularity

Frame Regularity is a repeating structure in a stack trace: the same frame or group of frames recurring down the stack, which is what recursion looks like and what a stack overflow looks like before it becomes one. Chapters 5 and 7 found the run of `Node::Visit` at the top of thread 107; the recognizer here finds a run anywhere in a trace and requires a threshold length:

```
% runs(List, Runs): Runs is the list of X-N pairs,
% one per maximal run of N consecutive elements
% equal to X in List.
runs([], []).
runs([X | Xs], [X-N | Runs]) :-
    run_length(Xs, X, 1, N, Rest),
    runs(Rest, Runs).

run_length(Xs, X, N0, N, Rest) :-
    (   Xs = [Y | Xs1],
        Y == X
    ->  N1 is N0 + 1,
        run_length(Xs1, X, N1, N, Rest)
    ;   N = N0,
        Rest = Xs
    ).

% frame_regularity(TID, Key, Count): the signature
% of TID has a run of Count frames with Key, Count
% at least the threshold.
frame_regularity(TID, Key, Count) :-
    thread(TID, _),
    stack_trace_signature(TID, Signature),
    runs(Signature, Runs),
    member(Key-Count, Runs),
    threshold(frame_run, Min),
    Count >= Min.
```

```
?- frame_regularity(T, K, N).
T = 107,
K = reportengine-'Node::Visit',
N = 8 ;
false.

?- stack_trace_signature(103, _S), runs(_S, _R),
   pairs_values(_R, _Ls), max_list(_Ls, Longest).
Longest = 1.
```

`runs/2` is the run counting of Chapter 9's histogram, applied to the signature in order rather than sorted; a trace without recursion has runs of length one throughout, as thread 103 shows. The predicate enumerates: `member/2` over the runs gives one answer for every run of the signature that reaches the threshold, with that run's key and length, and no longest run is selected, so that a thread with two runs of four is two instances. The second query computes the longest run of thread 103 by hand, with `max_list/2`, to show that none of its runs comes near the threshold. Thread 107's eight consecutive `Node::Visit` frames are the dump's one instance.

The predicate finds runs of one repeated frame. Mutual recursion, in which two or three frames alternate, is a regularity it does not see; a longer period needs a search for repeated sublists, with `append/3` as in Section 18.16. Nor does it say whether the recursion is a fault: eight levels of a tree walk is ordinary, and the pattern becomes a problem when the run exhausts the stack, which Chapter 19 treats under **Stack Overflow (User Mode)** (page 366) with the exception code that settles the question, and, short of the code, with a second and longer run threshold of its own, `overflow_run`, for the suspicion of it; `frame_run` remains the threshold of this pattern.

18.15 Caller-n-Callee

Caller-n-Callee is a pair of adjacent frames considered together: the function that made a call and the function it called, as a unit of analysis. It answers "who calls this function" across all threads, which finds the call sites of a lock or a hooked function, and "what does this function call", which finds the paths out of a dispatch point. In a frame list ordered top first the callee precedes the caller, and `nextto/3` from `library(lists)` relates adjacent elements:

```
% caller_n_callee(Caller, Callee, TID): in the
% stack trace of TID the frame with key Callee is
% directly above the frame with key Caller.
caller_n_callee(Caller, Callee, TID) :-
    thread(TID, _),
    stack_trace_signature(TID, Signature),
    nextto(Callee, Caller, Signature).

?- caller_n_callee(C,
   ntdll-'RtlEnterCriticalSection', T).
C = cachelib-'Unknown',
T = 101 ;
C = dbclient-'Connection::Send',
T = 103 ;
```

```
C = appserver-'Session::Lookup',
T = 104 ;
false.

?- caller_n_callee(appserver-'Worker::Run', C, T).
C = appserver-'Session::Lookup',
T = 101 ;
C = reportengine-'Report::Build',
T = 102 ;
C = appserver-'Session::Commit',
T = 103 ;
C = reportengine-'Report::Build',
T = 107 ;
C = appserver-'Session::Fetch',
T = 109 ;
false.
```

The first query lists the call sites of the critical section entry: an unresolved frame in the cache library, the database client's send routine, and the session lookup. Chapter 20 shows that two of the three wait for the same section, and the call sites say which code paths contend for it. The second query is the fan-out of `Worker::Run`, the five things the workers were asked to do, with `Report::Build` twice.

The relation is between adjacent frames as the debugger reconstructed them. An inlined function has no frame and appears in no pair, and a misplaced frame, as Section 18.17 discusses, produces a pair of functions that never called each other. The predicate does not know whether a call is possible; Section 18.20 uses a weak form of that knowledge.

18.16 Procedure Call Chain

A **Procedure Call Chain** is a stack trace read in the direction of execution, from the function that started the thread to the one that was executing, as the sequence of calls that brought the thread where it is; the reading makes the trace a path through the program's components and lets a known path be looked for in other traces. The chain is the reversed signature without the modules, and a sub-chain is a contiguous part of it:

```
% procedure_call_chain(TID, Chain): Chain is the
% list of functions of TID from the bottom frame
% to the top, in the order the calls were made.
procedure_call_chain(TID, Chain) :-
    reversed_signature(TID, Reversed),
    pairs_values(Reversed, Chain).

% chain_contains(TID, Sub): Sub is a contiguous
% part of the call chain of TID.
chain_contains(TID, Sub) :-
    procedure_call_chain(TID, Chain),
    append(_, Rest, Chain),
    append(Sub, _, Rest).
```

```
?- procedure_call_chain(109, _C), show(_C).
[ 'RtlUserThreadStart',
  'BaseThreadInitThunk',
  'thread_start<...>',
  'Worker::Run',
  'Session::Fetch',
  'Connection::Receive',
  'Buffer::Copy'
]
true.

?- thread(T, _),
   chain_contains(T,
   ['Worker::Run', 'Session::Lookup']).
T = 101 ;
false.

?- thread(T, _),
   chain_contains(T, ['Session::Lookup']).
T = 101 ;
T = 104 ;
false.
```

The two `append/3` goals split the chain at every position and take every prefix of the remainder, the standard way to enumerate sublists; with `Sub` bound it is a search. The chain of thread 109 is the story of the fault in order: a worker fetched a session, received from the connection, and copied into a buffer. `Session::Lookup` is called from `Worker::Run` on thread 101 and from `Connection::OnReply` on thread 104: the two threads reach the same lock by different paths.

The limits are those of the stack trace it reads, and one more: the chain says what called what, not when or how often. A function called in a loop appears once, with no record of the iterations that completed; the software trace of Part IV records the sequence of events, and the chain is the sequence of calls still open.

18.17 Incorrect Stack Trace

An **Incorrect Stack Trace** shows frames the thread did not have: functions named by the nearest symbol in a module without proper symbols, frames the stack walker took from stale values when it lost the frame chain, or a trace walked from a corrupt stack. The analyst recognizes it by its implausibility, a function that cannot call the one above it, and checks it against the raw stack. A predicate cannot recognize incorrectness, since the facts hold the trace as the debugger gave it and nothing to check it against; what it can recognize are the conditions under which the trace is unreliable:

```
% incorrect_stack_trace(TID, Reason): the stack
% trace of TID may be incorrect for Reason, a
% module on it without full symbols.
incorrect_stack_trace(TID, no_symbols(M)) :-
    thread(TID, _),
    distinct(M, frame(TID, _, M, _, _)),
    module_info(M, symbols, none).
incorrect_stack_trace(TID, export_symbols(M)) :-
    thread(TID, _),
    distinct(M, frame(TID, _, M, _, _)),
    module_info(M, symbols, export).

?- incorrect_stack_trace(T, R).
T = 101,
R = no_symbols(cachelib) ;
T = 103,
R = export_symbols(dbclient) ;
T = 104,
R = export_symbols(dbclient) ;
T = 109,
R = export_symbols(dbclient) ;
false.
```

`distinct/2` from `library(solution_sequences)` reports each module of a trace once, however many frames it has there. Four of the ten traces pass through a module whose symbols are missing or export symbols only, and Chapter 8 explained the two degrees: with no symbols the frame is an offset from the module base and the frames below it depend on the walker's recovery of the frame chain; with export symbols the function name is the nearest exported function before the address and may be wrong. The faulting thread is among the four, and Chapter 19 keeps that in view when it names `Buffer::Copy`.

The predicate's name states the pattern, and its answers are candidates: the four traces may be incorrect, and the predicate cannot say whether they are. The raw stack of Section 18.21 is what an analyst compares them against.

18.18 Truncated Stack Trace

A **Truncated Stack Trace** ends before it should: the stack walker stopped at a frame it could not unwind, because of a missing symbol file, missing unwind information, or a corrupt return address, and the frames below are absent rather than wrong. A complete user-mode trace ends in the system's thread start routine, and the test is whether the last frame is one:

```
% bottom_function(F): F ends a complete user-mode
% stack trace.
bottom_function('RtlUserThreadStart').
bottom_function('LdrInitializeThunk').
```

```
% truncated_frames(Frames): the non-empty frame
% list Frames does not end in a bottom function.
truncated_frames(Frames) :-
    last(Frames, frame(_, _, _, F, _)),
    \+ bottom_function(F).

% truncated_stack_trace(TID): the stack trace of
% TID is truncated.
truncated_stack_trace(TID) :-
    thread(TID, _),
    stack_trace(TID, Frames),
    truncated_frames(Frames).

?- truncated_stack_trace(T).
false.

?- stack_trace(101, _F), length(_P, 5),
   append(_P, _, _F), truncated_frames(_P),
   last(_P, _Last), frame_symbol(_Last, S).
S = 'cachelib+0x2a10'.

?- truncated_frames([]).
false.
```

No trace of `dump1.pl` is truncated. The second query cuts the trace of thread 101 after its fifth frame to show the shape: it ends in the unresolved `cachelib` frame, which is where a real walker lacking symbols for the module might have stopped. `last/2` fails on the empty list, so a thread without frames is not truncated by this test; it is a different finding, a thread the debugger could not walk at all, which shows as `stack_trace/2` giving `[]` for a thread that `thread/2` knows.

The test knows the bottom functions of one operating system family, listed as facts, and reports a trace ending in an unlisted bottom function as truncated. It cannot tell a truncated trace from one glued to a wrong bottom, which Section 18.20 treats, and cannot say how many frames are missing.

18.19 Coincidental Symbolic Information

Coincidental Symbolic Information is a value in memory that happens to fall within the address range of a module and is therefore shown with a symbol by `dps` or `dpu`, although it is not a code address: a handle, a counter, a data pointer whose value lies among the code. The analyst reads the raw stack as a column of symbols and must decide which are return addresses and which are coincidences.

The shared schema does not hold raw memory, so the pattern needs an extension: `raw_stack(TID, Addr, Value)` records a selection of the eight-byte words of a thread's stack, as `dps` would list them. The following facts are such a selection for thread 108, the suspended housekeeping thread, invented to the same standard as the shared files:

```
% raw_stack(TID, Addr, Value): the word at stack
% address Addr of thread TID holds Value. An
% extension of the schema.
raw_stack(108, 0xf1_2bff_e8a0, 0x18).
raw_stack(108, 0xf1_2bff_e8a8, 0x7ffb_b0b0_8f5e).
raw_stack(108, 0xf1_2bff_e8b0, 0xf1_2bff_e900).
raw_stack(108, 0xf1_2bff_e8b8, 0x7ffb_b2a0_0000).
raw_stack(108, 0xf1_2bff_e8c0, 0x7ff6_1d10_3a2c).
raw_stack(108, 0xf1_2bff_e8c8, 0x7ffb_8d10_12c4).
raw_stack(108, 0xf1_2bff_e8d0, 0x1f4).
raw_stack(108, 0xf1_2bff_e8d8, 0x7ffb_6f30_1040).

% address_module(Addr, Module): Addr lies in the
% address range of Module.
address_module(Addr, Module) :-
    module(Module, Base, Size),
    Base =< Addr,
    Addr < Base + Size.

% symbol_like(Value, Module, Offset): Value lies
% in Module at Offset; dps would show a symbol.
symbol_like(Value, Module, Offset) :-
    address_module(Value, Module),
    module(Module, Base, _),
    Offset is Value - Base.

% coincidental_symbolic_information(TID, Addr, M):
% the raw stack word at Addr of TID lies in module
% M at a page-aligned offset.
coincidental_symbolic_information(TID, Addr, M) :-
    raw_stack(TID, Addr, Value),
    symbol_like(Value, M, Offset),
    Offset /\ 0xfff =:= 0.

?- raw_stack(108, A, V), symbol_like(V, M, O),
   format("~16r ~a+0x~16r~n", [A, M, O]), fail.
f12bffe8a8 kernelbase+0x8f5e
f12bffe8b8 ntdll+0x0
f12bffe8c0 appserver+0x3a2c
f12bffe8c8 dbclient+0x12c4
f12bffe8d8 thirdparty_hook+0x1040
false.

?- coincidental_symbolic_information(108, A, M),
   format("~16r~n", [A]).
f12bffe8b8
A = 1035825309880,
M = ntdll ;
false.
```

Five of the eight words lie in some module and would be printed with symbols; the other three are a small integer, a stack address, and another small integer. The address-range test is `address_module/2` of Chapter 6, and the recognizer adds the one heuristic the facts support: a value at a page-aligned offset, and at offset zero in particular, is almost never a return address and very often a module handle, since a handle is the module's base address. The word at `0xf1_2bff_e8b8` is

the base of `ntdll`, which `dps` would show with the module's first symbol, and is coincidental in the pattern's sense.

The heuristic catches one shape of coincidence. A data pointer into a module's read-only data, a counter that happens to equal an address, and a stale return address all pass `symbol_like/3` and the alignment test, and the predicate cannot separate them from the return addresses of the live frames; the frame addresses that would settle it are not in the schema. The pattern is a caution about reading, and the predicate marks the one case in which the caution is certainly warranted.

18.20 Glued Stack Trace

A **Glued Stack Trace** is a reconstruction in which the stack walker, having lost the true frame chain, continued from stale values and appended the frames of an earlier or unrelated call sequence to the current one: the result reads as two traces joined, and the join shows as a frame that could not have called the one above it. The strongest form of the heuristic the facts allow is a bottom function in the middle of a trace, since the thread start routine is called by nothing within the process:

```
% glued_frames(Frames): some frame of the list
% Frames names a bottom function and is not the
% last frame.
glued_frames(Frames) :-
    append(_, [Frame, _ | _], Frames),
    Frame = frame(_, _, _, F, _),
    bottom_function(F).

% glued_stack_trace(TID): the stack trace of TID
% has a bottom function above another frame.
glued_stack_trace(TID) :-
    thread(TID, _),
    stack_trace(TID, Frames),
    glued_frames(Frames).

?- glued_stack_trace(T).
false.

?- stack_trace(105, _A), stack_trace(109, _B),
   append(_A, _B, _G), glued_frames(_G).
true ;
false.

?- stack_trace(105, _A), stack_trace(109, _B),
   append(_A, _B, _G), length(_G, N),
   nth0(I, _G, frame(_, _, _, F, _)),
   bottom_function(F).
N = 11,
I = 3,
F = 'RtlUserThreadStart' ;
N = 11,
I = 10,
F = 'RtlUserThreadStart'.
```

No trace in the dump is glued. The constructed example appends the trace of a pool thread to that of the faulting thread, which is the shape a glued trace has: eleven frames with `RtlUserThreadStart` at position 3 and again at the end. The `append/3` in `glued_frames/1` succeeds for every frame followed by at least one more and leaves a choice point, which the toplevel reports; the third query enumerates both occurrences with `nth0/3`.

A trace glued below a frame that is not a bottom function, the common case, passes this test. The general heuristic, a frame whose module cannot be the caller of the next, needs a relation of which modules call which, and the import tables that would provide it are not in the schema; the weaker rule that a system module does not call into application modules except through known callbacks holds often enough to be useful and seldom enough to be a fact. The raw stack of Section 18.21 is again what the analyst compares against.

18.21 Execution Residue (User Space)

Execution Residue (User Space) is what past execution leaves on a thread’s stack: return addresses, parameters, and local values of calls that have completed, lying below the current stack pointer or in the gaps between live frames, which the stack walker does not show and `dps` does. It is how an analyst learns what a thread did before its current wait, and it is the raw material of a **Glued Stack Trace** (page 349) when the walker mistakes residue for live frames. The `raw_stack/3` extension of Section 18.19 is the data, and the predicate is the complement of the coincidence test: a word in a module’s range at an offset that is not page-aligned is a candidate return address of a past call.

```
% execution_residue(TID, Addr, Module, Offset):
% the raw stack word at Addr of TID holds a code
% address of Module that is not page-aligned.
execution_residue(TID, Addr, Module, Offset) :-
    raw_stack(TID, Addr, Value),
    symbol_like(Value, Module, Offset),
    Offset /\ 0xfff =\= 0.

?- execution_residue(108, _, M, O),
   format("~a+0x~16r~n", [M, O]), fail.
kernelbase+0x8f5e
appserver+0x3a2c
dbclient+0x12c4
thirdparty_hook+0x1040
false.

?- execution_residue(108, _, M, _),
   \+ frame(108, _, M, _, _).
M = dbclient ;
M = thirdparty_hook.
```

```
?- print_stack_trace(108).
00 ntdll!NtDelayExecution+0x14
01 kernelbase!SleepEx+0x9e
02 appserver!Housekeeping::Run+0x3a
03 kernel32!BaseThreadInitThunk+0x14
04 ntdll!RtlUserThreadStart+0x21
true.
```

Four words of the housekeeping thread's stack point into code. Two are in modules on its current trace and may be return addresses of live frames or residue of earlier calls through the same modules; the facts cannot tell, since they do not hold the frame addresses. The other two are in `dbclient` and `thirdparty_hook`, which are not on the trace at all, and that is the finding: the housekeeping thread has at some earlier time executed code in the database client and in the hooking module, and its current trace, five frames of a thread asleep in `SleepEx`, does not say so. Chapter 7 found no thread executing in `thirdparty_hook`; the residue shows that one had been.

The predicate reports candidates, which is all residue is. Without symbols for the hooking module nothing in the dump says whether `0x1040` is a code or a data offset; `dbclient+0x12c4` can be named from export symbols, with the reliability Chapter 8 described. A value can be residue from a call made minutes before the dump or from the stack region's earlier use for an unrelated purpose. The pattern is the analyst's reading of that palimpsest, and the predicate narrows the reading to the words that can be code.

Chapter 19. Exceptions and Faults

The patterns of this chapter concern the exception records of a process memory dump and the faults they report: where the faulting thread was, what it touched, what kind of fault it was, and which module to blame. `dump1.pl` has one exception record, the access violation of thread 109, and most of the chapter's patterns describe faults it does not contain; the chapter therefore extends the shared dump with synthetic threads, each exhibiting one fault configuration, and runs every predicate over both.

The extension is the file `faults.pl`. It includes `dump1.pl` with the directive `include/1`, which inserts the clauses of the included file at that point of the including one, so that the schema predicates have the clauses of both files; `consult/1` would instead have loaded `dump1.pl` as a file of its own, and clauses added afterwards would have replaced its definitions with a warning. The threads 110 to 119 are not a consistent state of one process, since each was invented to show one fault. Two further modules, the C++ runtime and the managed runtime, are added because two of the faults pass through them; two extension relations, `frame_address/3` and `stored_exception/4`, record what the schema does not; and the directive `discontiguous/1` declares that the clauses of the extended predicates are not adjacent in the file, so that the loader does not warn.

```
% faults.pl -- dump1.pl extended with synthetic threads
% 110-119, each showing one fault configuration that the
% shared dump lacks. The threads do not form a consistent
% process state and are not meant to. An extension of the
% schema; every value is invented. Three relations are
% dynamic, so that Section 19.9 can add synthetic stacks
% and take them away again.
:- discontiguous module/3, module_info/3, thread/2,
   thread_time/3, thread_state/2, exception_record/3,
   exception_info/3.
:- dynamic thread/2, frame/5, exception_record/3.
:- include(dump1).

module(vcruntime140, 0x7ffb_9b10_0000, 0x1e000).
module(coreclr, 0x7ffb_7c00_0000, 0x5a0000).
module_info(vcruntime140, symbols, pdb).
module_info(coreclr, symbols, pdb).

thread(110, 4120).
thread(111, 4120).
thread(112, 4120).
thread(113, 4120).
thread(114, 4120).
thread(115, 4120).
thread(116, 4120).
thread(117, 4120).
thread(119, 4120).
```

```
thread_time(110, 10, 5).
thread_time(111, 20, 5).
thread_time(112, 15, 5).
thread_time(113, 30, 10).
thread_time(114, 25, 10).
thread_time(115, 20, 5).
thread_time(116, 15, 5).
thread_time(117, 10, 5).
thread_time(119, 400, 50).

thread_state(110, waiting(exception)).
thread_state(111, waiting(exception)).
thread_state(112, waiting(exception)).
thread_state(113, waiting(exception)).
thread_state(114, waiting(exception)).
thread_state(115, waiting(user_request)).
thread_state(116, waiting(exception)).
thread_state(117, waiting(exception)).
thread_state(119, waiting(exception)).

% 110: a call through a null function pointer
frame(110, 0, none, 'Unknown', 0).
frame(110, 1, appserver, 'Plugin::Invoke', 0x3c).
frame(110, 2, appserver, 'Worker::Run', 0x120).
frame(110, 3, ucrtbase, 'thread_start<...>', 0x42).
frame(110, 4, kernel32, 'BaseThreadInitThunk', 0x14).
frame(110, 5, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(110, 0xc0000005, 0x0).
exception_info(110, access, execute).
exception_info(110, target, 0x0).

% 111: integer division by zero
frame(111, 0, reportengine, 'Stats::Mean', 0x40).
frame(111, 1, reportengine, 'Report::Summarize', 0x88).
frame(111, 2, appserver, 'Worker::Run', 0x120).
frame(111, 3, ucrtbase, 'thread_start<...>', 0x42).
frame(111, 4, kernel32, 'BaseThreadInitThunk', 0x14).
frame(111, 5, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(111, 0xc0000094, 0x7ffb_8e41_2b40).

% 112: an unhandled C++ exception
frame(112, 0, kernelbase, 'RaiseException', 0x6e).
frame(112, 1, vcruntime140, '_CxxThrowException', 0x90).
frame(112, 2, appserver, 'Parser::Expect', 0x5a).
frame(112, 3, appserver, 'Parser::Parse', 0x1c4).
frame(112, 4, appserver, 'Worker::Run', 0x120).
frame(112, 5, ucrtbase, 'thread_start<...>', 0x42).
frame(112, 6, kernel32, 'BaseThreadInitThunk', 0x14).
frame(112, 7, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(112, 0xe06d7363, 0x7ffb_b0b2_1a3e).
```

```
% 113: an unhandled managed exception; frame 3 is
% JIT-compiled code outside every module; the frames
% below it are abridged
frame(113, 0, kernelbase, 'RaiseException', 0x6e).
frame(113, 1, coreclr, 'RaiseTheExceptionInternalOnly', 0x2a8).
frame(113, 2, coreclr, 'IL_Throw', 0x1c0).
frame(113, 3, none, 'Unknown', 0).
frame(113, 4, kernel32, 'BaseThreadInitThunk', 0x14).
frame(113, 5, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(113, 0xe0434352, 0x7ffb_b0b2_1a3e).

% 114: a C++ exception thrown by the handler of an
% access violation
frame(114, 0, kernelbase, 'RaiseException', 0x6e).
frame(114, 1, vcruntime140, '_CxxThrowException', 0x90).
frame(114, 2, appserver, 'Handler::Report', 0x66).
frame(114, 3, appserver, 'Handler::OnFault', 0x2e).
frame(114, 4, ntdll, 'RtlpExecuteHandlerForException', 0xf).
frame(114, 5, ntdll, 'RtlDispatchException', 0x244).
frame(114, 6, ntdll, 'KiUserExceptionDispatch', 0x2e).
frame(114, 7, dbclient, 'Buffer::Copy', 0x1a).
frame(114, 8, dbclient, 'Connection::Receive', 0x8c).
frame(114, 9, appserver, 'Session::Fetch', 0x44).
frame(114, 10, appserver, 'Worker::Run', 0x120).
frame(114, 11, ucrtbase, 'thread_start<...>', 0x42).
frame(114, 12, kernel32, 'BaseThreadInitThunk', 0x14).
frame(114, 13, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(114, 0xe06d7363, 0x7ffb_b0b2_1a3e).

% 115: a handler waiting for a log flush; no current
% exception record
frame(115, 0, ntdll, 'NtWaitForSingleObject', 0x14).
frame(115, 1, kernelbase, 'WaitForSingleObjectEx', 0x8e).
frame(115, 2, appserver, 'Logger::Flush', 0x72).
frame(115, 3, appserver, 'Handler::OnFault', 0x4a).
frame(115, 4, ntdll, 'RtlpExecuteHandlerForException', 0xf).
frame(115, 5, ntdll, 'RtlDispatchException', 0x244).
frame(115, 6, ntdll, 'KiUserExceptionDispatch', 0x2e).
frame(115, 7, appserver, 'Session::Parse', 0xb0).
frame(115, 8, appserver, 'Worker::Run', 0x120).
frame(115, 9, ucrtbase, 'thread_start<...>', 0x42).
frame(115, 10, kernel32, 'BaseThreadInitThunk', 0x14).
frame(115, 11, ntdll, 'RtlUserThreadStart', 0x21).

% 116: a read through a pointer holding text
frame(116, 0, dbclient, 'Row::Get', 0x32).
frame(116, 1, dbclient, 'Cursor::Next', 0x7d).
frame(116, 2, appserver, 'Session::Fetch', 0x58).
frame(116, 3, appserver, 'Worker::Run', 0x120).
frame(116, 4, ucrtbase, 'thread_start<...>', 0x42).
frame(116, 5, kernel32, 'BaseThreadInitThunk', 0x14).
frame(116, 6, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(116, 0xc0000005, 0x7ffb_8d10_4c12).
exception_info(116, access, read).
exception_info(116, target, 0x4141_4141).

% 117: execution at an address outside every module,
% with return addresses overwritten by text
frame(117, 0, none, 'Unknown', 0).
```

```
frame(117, 1, none, 'Unknown', 0).
frame(117, 2, none, 'Unknown', 0).
frame(117, 3, appserver, 'Parser::Parse', 0x1c4).
frame(117, 4, appserver, 'Worker::Run', 0x120).
frame(117, 5, ucrtbase, 'thread_start<...>', 0x42).
frame(117, 6, kernel32, 'BaseThreadInitThunk', 0x14).
frame(117, 7, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(117, 0xc0000005, 0x4141_4141).
exception_info(117, access, execute).
exception_info(117, target, 0x4141_4141).

% 119: stack exhausted by recursion; the run of
% Node::Visit frames is abridged
frame(119, 0, reportengine, 'Node::Visit', 0x12).
frame(119, 1, reportengine, 'Node::Visit', 0x6b).
frame(119, 2, reportengine, 'Node::Visit', 0x6b).
frame(119, 3, reportengine, 'Node::Visit', 0x6b).
frame(119, 4, reportengine, 'Node::Visit', 0x6b).
frame(119, 5, reportengine, 'Node::Visit', 0x6b).
frame(119, 6, reportengine, 'Node::Visit', 0x6b).
frame(119, 7, reportengine, 'Tree::Walk', 0x30).
frame(119, 8, reportengine, 'Report::Build', 0x1f0).
frame(119, 9, appserver, 'Worker::Run', 0x120).
frame(119, 10, ucrtbase, 'thread_start<...>', 0x42).
frame(119, 11, kernel32, 'BaseThreadInitThunk', 0x14).
frame(119, 12, ntdll, 'RtlUserThreadStart', 0x21).
exception_record(119, 0xc00000fd, 0x7ffb_8e40_c012).

% frame_address(TID, N, Addr): the address of frame N
% of TID, recorded for the frames outside every module
% and their neighbors. An extension of the schema.
frame_address(110, 0, 0x0).
frame_address(110, 1, 0x7ff6_1d11_203c).
frame_address(117, 0, 0x4141_4141).
frame_address(117, 1, 0x4141_4141).
frame_address(117, 2, 0x4141_4141).
frame_address(117, 3, 0x7ff6_1d11_a1c4).

% stored_exception(TID, Addr, Code, ExcAddr): an
% exception record found in the memory of TID at Addr,
% recording exception Code raised at ExcAddr. An
% extension of the schema.
stored_exception(109, 0xf1_2aff_e2a0, 0xc0000005,
                 0x7ffb_8d10_31aa).
stored_exception(115, 0xf1_2cff_e5c0, 0xc0000005,
                 0x7ff6_1d11_b0c8).
```

A frame whose address lies in no module, the result of a call through a null or corrupt pointer or of JIT-compiled code, is recorded with the module `none` and the function `'Unknown'`, and the address itself in `frame_address/3` where the chapter needs it. The file is consulted as it is defined; the rest of the chapter's helpers follow, with the `threshold/2` facts and the `pattern/3` registry of Chapter 17 and the **status codes** that several patterns test, as `status_name/2` facts with the names Windows gives them; `system_module/1` is the chapter's division of the modules into system and application code:

```
:- use_module(library(pprint)).

% threshold(Name, Value): the thresholds.
% frame_run is Chapter 18's run length for a Frame
% Regularity; overflow_run is the longer run that
% makes a deep stack a suspected overflow.
threshold(null_page, 0x1_0000).
threshold(stack_depth, 12).
threshold(frame_run, 4).
threshold(overflow_run, 6).
threshold(text_bytes, 4).

% status_name(Code, Name): the name of a status
% code that marks a kind of exception.
status_name(0xc0000005, access_violation).
status_name(0xc0000094, divide_by_zero).
status_name(0xc00000fd, stack_overflow).
status_name(0xe06d7363, cpp_exception).
status_name(0xe0434352, clr_exception).
status_name(0x80000003, breakpoint).

% system_module(M): M belongs to the operating
% system or a runtime, not to the application.
system_module(ntdll).
system_module(kernelbase).
system_module(kernel32).
system_module(ucrtbase).
system_module(msvcp140).
system_module(vcruntime140).
system_module(coreclr).

% pattern(Name, Artifact, Chapter): the patterns
% this chapter recognizes with a predicate.
pattern('Exception Stack Trace', memory_dump, 19).
pattern('Fault Context', memory_dump, 19).
pattern('NULL Pointer (Data)', memory_dump, 19).
pattern('NULL Pointer (Code)', memory_dump, 19).
pattern('Invalid Pointer (General)',
    memory_dump, 19).
pattern('Wild Pointer', memory_dump, 19).
pattern('Wild Code', memory_dump, 19).
pattern('Local Buffer Overflow (User Space)',
        memory_dump, 19).
pattern('Stack Overflow (User Mode)',
    memory_dump, 19).
pattern('Divide by Zero (User Mode)',
    memory_dump, 19).
pattern('Nested Exceptions (User Space)',
        memory_dump, 19).
pattern('Multiple Exceptions (User Space)',
        memory_dump, 19).
pattern('Hidden Exception (User Space)',
        memory_dump, 19).
pattern('Handled Exception (User Space)',
        memory_dump, 19).
pattern('C++ Exception', memory_dump, 19).
pattern('Managed Code Exception',
    memory_dump, 19).
pattern('Stored Exception', memory_dump, 19).
```

```
pattern('Exception Module', memory_dump, 19).
pattern('Problem Module', memory_dump, 19).
pattern('Crash Signature', memory_dump, 19).

% show(Term): print Term over several lines that
% fit the width of this book's pages.
show(Term) :-
    print_term(Term, [right_margin(46),
                      tab_width(0)]),
    nl.

% address_module(Addr, Module): Addr lies in the
% address range of Module.
address_module(Addr, Module) :-
    module(Module, Base, Size),
    Base =< Addr,
    Addr < Base + Size.

% stack_trace(TID, Frames): the frame/5 terms of
% TID, top frame first.
stack_trace(TID, Frames) :-
    findall(frame(TID, N, M, F, O),
            frame(TID, N, M, F, O), Frames0),
    sort(2, @=<, Frames0, Frames).

% frame_symbol(Frame, Symbol): the debugger's text
% for Frame; the address alone for a frame in no
% module.
frame_symbol(frame(TID, N, M, F, O), Symbol) :-
    (   M == none
    ->  (   frame_address(TID, N, Addr)
        ->  format(atom(Symbol), "0x~16r", [Addr])
        ;   Symbol = '<no module>'
        )
    ;   F == 'Unknown'
    ->  format(atom(Symbol), "~a+0x~16r", [M, O])
    ;   format(atom(Symbol), "~a!~a+0x~16r",
               [M, F, O])
    ).

% print_stack_trace(TID): one line per frame.
print_stack_trace(TID) :-
    stack_trace(TID, Frames),
    forall(member(Frame, Frames),
           ( Frame = frame(_, N, _, _, _),
             frame_symbol(Frame, Symbol),
             format("~`0t~d~2| ~a~n", [N, Symbol])
           )).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, thread(_, _), N).
N = 19.

?- aggregate_all(count,
                 exception_record(_, _, _), N).
N = 9.
```

The knowledge base has nineteen threads, ten from the shared dump and nine from the extension, and nine of them have exception records. The lowest 64 KB of the address space are the **null page**, the region Windows never maps, so that any access below `0x1_0000` is an access through a null pointer plus a small offset; the threshold makes the bound a fact. The toplevel prints status codes and addresses in decimal; the transcripts use `format/2` with `~16r` where the hexadecimal form is the one an analyst knows.

19.1 Exception Stack Trace

The **Exception Stack Trace** is the stack trace of the thread whose exception record the dump holds, read from the faulting instruction down: it is the first trace an analyst reads in a crash dump, and in a dump with one exception record it is the one the debugger's `!analyze -v` prints. The predicate selects the threads with a record and gives their traces:

```
% exception_stack_trace(TID, Frames): TID has an
% exception record and Frames is its stack trace.
exception_stack_trace(TID, Frames) :-
    exception_record(TID, _, _),
    stack_trace(TID, Frames).

?- exception_stack_trace(T, _Frames), T < 110,
   print_stack_trace(T).
00 dbclient!Buffer::Copy+0x1a
01 dbclient!Connection::Receive+0x8c
02 appserver!Session::Fetch+0x44
03 appserver!Worker::Run+0x120
04 ucrtbase!thread_start<...>+0x42
05 kernel32!BaseThreadInitThunk+0x14
06 ntdll!RtlUserThreadStart+0x21
T = 109 ;
false.

?- findall(T, exception_stack_trace(T, _), _Ts),
   length(_Ts, N), last(_Ts, Last).
N = 9,
Last = 119.
```

The one exception stack trace of the shared dump is thread 109's: a worker fetching a session received data from the connection and faulted while copying it into a buffer. The second query counts nine threads of the extended knowledge base with a trace of this kind; `multiple_exceptions/1` in Section 19.12 is the pattern that there is more than one.

The predicate is the composition of two goals and adds nothing to them; what it fixes is the reading. The trace is the fault's context only if the frames were recorded from that context, the subject of the next pattern, and its correctness is subject to everything Chapter 18 said; the faulting thread of `dump1.pl` passes through a module with export symbols only.

19.2 Fault Context

The **Fault Context** is the processor state saved when the exception was raised, the registers and the stack pointer as they were at the faulting instruction, which the debugger restores with `.ecxr` so that `k` shows the exception stack trace rather than the trace of the thread as it stood when the dump was written, inside the exception dispatcher or the unhandled exception filter. The schema records the frames of a faulting thread from the fault context, as Chapter 15 said the parser should, and the exception record with its code, address, and parameters. The predicate gathers everything the schema knows about one fault into a term:

```
% fault_context(TID, Context): Context collects
% the exception record of TID, the module of its
% address, its parameters and the thread's last
% error.
fault_context(TID, Context) :-
    exception_record(TID, Code, Addr),
    (   status_name(Code, Name)
    ->  true
    ;   Name = unknown
    ),
    (   address_module(Addr, M)
    ->  true
    ;   M = none
    ),
    findall(K-V, exception_info(TID, K, V), Info),
    (   last_error(TID, E)
    ->  true
    ;   E = unrecorded
    ),
    Context = context(status(Code, Name),
                      address(Addr, M),
                      Info, last_error(E)).

?- fault_context(109, _C), show(_C).
context(status(3221225477,access_violation),
        address(140718380167594,dbclient),
        [access-read,target-24],
        last_error(998))
true.

?- fault_context(111, _C), show(_C).
context(status(3221225620,divide_by_zero),
        address(140718400154432,reportengine),
        [],
        last_error(unrecorded))
true.

?- forall(( exception_record(T, _, A),
            address_module(A, M) ),
          frame(T, 0, M, _, _)).
true.
```

The three if-then-elses are the lookups with defaults of Chapter 7: a status code the table does not name, an address in no module, a thread without a `last_error/2` fact. A division by zero has no

access and target parameters, so its `Info` is empty, where an access violation has both. The last query is the consistency check that the recorded frames are the fault context: for every exception whose address lies in a module, the top frame is in that module. It holds for all nine, and it would fail for a dump whose frames were recorded from the dispatcher's context, with `ntdll` at the top of a thread whose exception address is in `dbclient`; that failure is how the pattern shows itself in a knowledge base built from the wrong `k`.

The registers themselves are not in the schema, and what the analyst reads from them, which register held the bad pointer and what the function's parameters were, is outside what these facts support. The context term is the schema's whole knowledge of the fault, and the next patterns are its parts.

19.3 NULL Pointer (Data)

NULL Pointer (Data) is an **access violation** whose target address lies in the null page: the code read or wrote through a pointer that was null, and the target address is the offset of the field it meant to reach. Chapter 6 found the configuration in thread 109 with one comparison; the recognizer here states it with the threshold as a fact, adds the explainer form with its evidence term, and distinguishes data access from execution, which Section 19.4 treats:

```
% access_violation(TID): the exception of TID is
% an access violation.
access_violation(TID) :-
    exception_record(TID, 0xc0000005, _).

% null_pointer_data(TID): TID read or wrote
% through a null pointer.
null_pointer_data(TID) :-
    access_violation(TID),
    exception_info(TID, access, Access),
    Access \== execute,
    exception_info(TID, target, Target),
    threshold(null_page, Limit),
    Target < Limit.

% null_pointer_data(TID, Evidence): as above, with
% the facts the recognition rests on: the record,
% its parameters, and the null page limit.
null_pointer_data(TID, Evidence) :-
    null_pointer_data(TID),
    exception_record(TID, Code, Addr),
    findall(exception_info(TID, K, V),
            exception_info(TID, K, V), Info),
    threshold(null_page, Limit),
    Facts = [exception_record(TID, Code, Addr)
             | Rest],
    append(Info, [threshold(null_page, Limit)],
           Rest),
    Evidence = evidence('NULL Pointer (Data)',
                        TID, Facts).
```

```
?- null_pointer_data(T).
T = 109 ;
false.

?- null_pointer_data(109, _E), show(_E).
evidence('NULL Pointer (Data)',
         109,
         [ exception_record(109,
                            3221225477,
                            140718380167594),
           exception_info(109,access,read),
           exception_info(109,target,24),
           threshold(null_page,65536)
         ])
true ;
false.

?- null_pointer_data(T),
   exception_info(T, target, Offset),
   format("field at offset 0x~16r~n", [Offset]).
field at offset 0x18
T = 109,
Offset = 24 ;
false.
```

Thread 109 read through a null pointer at offset `0x18`, 24 bytes into a structure it expected `Buffer::Copy` to have been given. The evidence is the exception record, its parameters, and the null page limit the target was compared with: the record and the parameters alone would be true facts from which the recognition does not follow, since the comparison `Target < Limit` needs the limit, and the contract of Section 17.3 asks that every threshold the clause compared against be in the list. The trailing `false.` in each query is the choice point of `exception_record/3` over the other eight records.

The predicate recognizes the null page and nothing about the pointer’s origin: a field never initialized, a lookup that returned nothing and was not checked, a structure freed and zeroed. The offset names the field, and the mapping from offset to field is in the program’s type information, which the symbol files may carry and the schema does not. The threshold of 64 KB is the size of the reserved region on Windows; a large structure accessed far from its start can exceed it and be missed.

19.4 NULL Pointer (Code)

NULL Pointer (Code) is an attempt to execute at an address in the null page: a call through a function pointer that was null, or through a table entry never filled in. The exception address and the target coincide, since the instruction pointer itself is the bad address, and the access parameter says `execute`:

```
% null_pointer_code(TID): TID tried to execute
% code at an address in the null page.
null_pointer_code(TID) :-
    access_violation(TID),
    exception_info(TID, access, execute),
    exception_info(TID, target, Target),
    threshold(null_page, Limit),
    Target < Limit.

?- null_pointer_code(T).
T = 110 ;
false.

?- null_pointer_code(T), print_stack_trace(T).
00 0x0
01 appserver!Plugin::Invoke+0x3c
02 appserver!Worker::Run+0x120
03 ucrtbase!thread_start<...>+0x42
04 kernel32!BaseThreadInitThunk+0x14
05 ntdll!RtlUserThreadStart+0x21
T = 110 ;
false.
```

The synthetic thread 110 called through a null function pointer from `Plugin::Invoke`; its top frame is in no module and `frame_symbol/2` prints the address the extension recorded, as `k` prints a bare address for a frame it cannot name. The frame below is the caller and the one to read, since the null pointer was its value.

What the predicate cannot say is where the pointer came from, and a stack walk from a null instruction pointer is itself unreliable: the frame below is recovered from the return address the call pushed, which is right for a call through a register and may be wrong after other corruption. An address in the null page that is not zero, say `0x40`, is a function pointer field at that offset of a null structure, and the offset is the clue, as in Section 19.3.

19.5 Invalid Pointer (General)

Invalid Pointer (General) is an access violation whose target is neither in the null page nor otherwise accounted for: a pointer that is not null but points at memory the process cannot access, because it was freed and its pages released, never allocated, or corrupted into a value that happens to be unmapped. It is the general case from which the more specific pointer patterns are separated by what the target value looks like:

```
% invalid_pointer(TID): TID read or wrote through
% a pointer outside the null page.
invalid_pointer(TID) :-
    access_violation(TID),
    exception_info(TID, access, Access),
    Access \== execute,
    exception_info(TID, target, Target),
    threshold(null_page, Limit),
    Target >= Limit.
```

```
?- invalid_pointer(T).
T = 116 ;
false.

?- invalid_pointer(T),
   exception_info(T, target, A),
   format("0x~16r~n", [A]).
0x41414141
T = 116,
A = 1094795585 ;
false.
```

One thread of the extension has such a pointer, and its value, `0x4141_4141`, is the subject of the next section. The test is the complement of the null page test and covers every data access violation that is not a null pointer; the catalog's intent is narrower, the pointer whose value does not itself explain the fault, and the recognizer is a first classification that the next two patterns refine. Heap ranges, which `heap/4` gives only as totals, would allow a further distinction between a target inside a heap, a freed block, and a target outside every known region.

19.6 Wild Pointer

A **Wild Pointer** is an invalid pointer whose value is recognizably not an address at all: the bytes of a string, a small integer, a floating-point number, a value that was written over the pointer or read from the wrong field. The test the facts support is the one an analyst applies by eye to `0x4141_4141`: every byte is a printable character.

```
% value_bytes(Value, Bytes): Bytes are the bytes
% of the non-negative integer Value, least
% significant first, without leading zero bytes.
value_bytes(Value, Bytes) :-
    (   Value =:= 0
    ->  Bytes = []
    ;   B is Value /\ 0xff,
        Rest is Value >> 8,
        value_bytes(Rest, Bytes0),
        Bytes = [B | Bytes0]
    ).

% printable(B): byte B is a printable ASCII
% character.
printable(B) :-
    B >= 0x20,
    B =< 0x7e.

% looks_like_text(Value): Value has at least the
% threshold many bytes and all are printable.
looks_like_text(Value) :-
    value_bytes(Value, Bytes),
    threshold(text_bytes, Min),
    length(Bytes, N),
    N >= Min,
    maplist(printable, Bytes).
```

```
% wild_pointer(TID): the invalid pointer of TID
% looks like text.
wild_pointer(TID) :-
    invalid_pointer(TID),
    exception_info(TID, target, Target),
    looks_like_text(Target).

?- wild_pointer(T).
T = 116 ;
false.

?- wild_pointer(T), exception_info(T, target, V),
   value_bytes(V, _Bytes),
   format("\"~s\"~n", [_Bytes]).
"AAAA"
T = 116,
V = 1094795585 ;
false.

?- looks_like_text(0x18).
false.

?- looks_like_text(0x7ffb_8d10_31aa).
false.
```

`value_bytes/2` takes the value apart with the bit operations of Chapter 6, least significant byte first, which is the order in which the bytes lie in memory on this architecture and so the order in which the string was stored. The pointer of thread 116 spells `"AAAA"`, the signature of a buffer filled with a repeated character and overrun into the pointer. The null pointer's offset and the faulting address of thread 109 both fail the test, the first for having one byte, the second because its bytes are not all printable.

The heuristic recognizes text and nothing else wild. A pointer corrupted to a small integer, a count or a flag, passes `invalid_pointer/1` and fails this test and is as wild; a pointer whose bytes are printable by chance, which is unlikely for an eight-byte value and less unlikely for four bytes, passes it and is not. The threshold of four bytes sets the trade-off, and the predicate reports the value for the analyst to read.

19.7 Wild Code

Wild Code is execution at an address that is in no module: the instruction pointer was loaded from a corrupt return address or function pointer, or control passed into data, and the thread faulted either at once, because the address was unmapped, or after executing whatever bytes it found there. The test is on the exception address rather than the target, since the fault is in the fetching of the instruction:

```
% wild_code(TID): the exception of TID was raised
% at an address outside the null page and outside
% every module.
wild_code(TID) :-
    exception_record(TID, _, Addr),
    threshold(null_page, Limit),
    Addr >= Limit,
    \+ address_module(Addr, _).

?- wild_code(T).
T = 117 ;
false.

?- wild_code(T), print_stack_trace(T).
00 0x41414141
01 0x41414141
02 0x41414141
03 appserver!Parser::Parse+0x1c4
04 appserver!Worker::Run+0x120
05 ucrtbase!thread_start<...>+0x42
06 kernel32!BaseThreadInitThunk+0x14
07 ntdll!RtlUserThreadStart+0x21
T = 117 ;
false.
```

Thread 117 is executing at `0x4141_4141`, and the two frames below it have the same address, which is the shape of a stack whose return addresses were overwritten with the same bytes; Section 19.8 names the cause. The null page is excluded so that a call through a null pointer is **NULL Pointer (Code)** (page 361) rather than wild code, and the two tests partition the execution faults outside modules between them.

The predicate does not know JIT-compiled code, which legitimately executes outside every module in a managed process, from code in a region the dump does not list as a module, such as a shellcode buffer; `module/3` is the loader's list, and code can run outside it. A frame in no module below a healthy top frame, as thread 113 has, is not wild code by this test, since the exception address is in a module; what it is depends on the runtime.

19.8 Local Buffer Overflow (User Space)

A **Local Buffer Overflow (User Space)** is a write past the end of a buffer on the stack that overwrites the return address and other control data of the frame; the thread continues until the function returns, when it jumps to the overwritten address, and the stack trace shows return addresses that lie in no module and often spell the bytes that overran. The recognizer looks for the second symptom in the frame addresses the `frame_address/3` extension records:

```
% local_buffer_overflow(TID, Frames): TID has an
% exception and Frames lists the N-Addr pairs of
% its frames below the top whose return address is
% in no module and looks like text.
local_buffer_overflow(TID, Frames) :-
    exception_record(TID, _, _),
```

```
    findall(N-Addr,
            ( frame_address(TID, N, Addr),
              N >= 1,
              \+ address_module(Addr, _),
              looks_like_text(Addr)
            ),
            Frames),
    Frames \== [].

?- local_buffer_overflow(T, Fs).
T = 117,
Fs = [1-1094795585, 2-1094795585] ;
false.

?- local_buffer_overflow(T, _), wild_code(T).
T = 117 ;
false.
```

Frames 1 and 2 of thread 117 have return addresses that are text, which is the evidence; the exception at the same address is the consequence. The two patterns coincide on this thread, and the second query says so: the overflow is the hypothesis about the cause, and the wild code is the observed fault. The function whose buffer overflowed is the one that returned into the text, `Parser::Parse` or a function it called whose frame the walker could not recover.

The recognition needs frame addresses, which the shared schema lacks and the extension records for a few frames; a parser that keeps the address columns of `k` would supply them for every frame. The heuristic misses an overflow that overwrote the return address with a valid code address, as an exploit does deliberately and an overrun with pointer data does by accident; such an overflow shows as a **Glued Stack Trace** (page 349) of Chapter 18 or as a plausible but wrong trace, and the raw stack is the recourse.

19.9 Stack Overflow (User Mode)

A **Stack Overflow (User Mode)** is the exhaustion of a thread's stack, nearly always by unbounded recursion: the thread touches the guard page at the end of its stack, the system raises the status `0xc00000fd`, and the trace shows the same frame repeated for as far as the debugger will print it. The exception code is conclusive when present, and the **Frame Regularity** (page 342) of Chapter 18 is the sign of the same condition before the code is raised. The recognizer has a clause for each, and the form of its finding says which: the code establishes the pattern and is reported as an evidence term, while depth and regularity are grounds for suspecting it and are reported as the suspicion term `suspected/3` of Section 17.3, whose grounds are callable goals under the same contract as evidence, never a status atom or a descriptive term among them.

Two run thresholds are in play, and they are two policies with two names. `frame_run`, four, is Chapter 18's: a run that long is a **Frame Regularity** (page 342), and `frame_regularity/3`, repeated here with the clauses Chapter 18 gave it, enumerates every run of a thread's signature that reaches it, one answer per run, without selecting a longest. `overflow_run`, six, is this section's: a run that

long on a stack at least `stack_depth` deep is grounds to suspect an overflow. A regular stack is not a suspected overflow until its run reaches the second threshold, and the second clause of `stack_overflow/2` compares the run `frame_regularity/3` found against `overflow_run`, not against `frame_run`, which that predicate has already applied.

```
% runs(List, Runs): Runs is the list of X-N pairs,
% one per maximal run of N consecutive elements
% equal to X in List.
runs([], []).
runs([X | Xs], [X-N | Runs]) :-
    run_length(Xs, X, 1, N, Rest),
    runs(Rest, Runs).

run_length(Xs, X, N0, N, Rest) :-
    (   Xs = [Y | Xs1],
        Y == X
    ->  N1 is N0 + 1,
        run_length(Xs1, X, N1, N, Rest)
    ;   N = N0,
        Rest = Xs
    ).

% frame_key(Frame, Key): Key is the pair
% Module-Function of the frame/5 term Frame.
frame_key(frame(_, _, M, F, _), M-F).

% stack_trace_signature(TID, Signature): the keys
% of the frames of TID, top frame first.
stack_trace_signature(TID, Signature) :-
    stack_trace(TID, Frames),
    maplist(frame_key, Frames, Signature).

% frame_regularity(TID, Key, Count): the signature
% of TID has a run of Count frames with Key, Count
% at least the threshold.
frame_regularity(TID, Key, Count) :-
    thread(TID, _),
    stack_trace_signature(TID, Signature),
    runs(Signature, Runs),
    member(Key-Count, Runs),
    threshold(frame_run, Min),
    Count >= Min.

% stack_depth(TID, Depth): the stack trace of TID
% has Depth frames.
stack_depth(TID, Depth) :-
    stack_trace(TID, Frames),
    length(Frames, Depth).

% stack_overflow(TID, Finding): Finding is the
% evidence that TID overflowed its stack, by the
% status code, or the suspicion of it, by depth
% and a run at least overflow_run long.
stack_overflow(TID, Finding) :-
    exception_record(TID, 0xc00000fd, Addr),
    Name = 'Stack Overflow (User Mode)',
    Code = 0xc00000fd,
```

```
    Facts = [exception_record(TID, Code, Addr)],
    Finding = evidence(Name, TID, Facts).
stack_overflow(TID, Finding) :-
    thread(TID, PID),
    stack_depth(TID, Depth),
    threshold(stack_depth, MinDepth),
    Depth >= MinDepth,
    frame_regularity(TID, Key, Count),
    threshold(overflow_run, MinRun),
    Count >= MinRun,
    Name = 'Stack Overflow (User Mode)',
    Grounds = [ thread(TID, PID),
                stack_depth(TID, Depth),
                threshold(stack_depth, MinDepth),
                frame_regularity(TID, Key, Count),
                threshold(overflow_run, MinRun)
              ],
    Finding = suspected(Name, TID, Grounds).

?- stack_overflow(T, _F), functor(_F, Form, 3),
   format("~a ~d~n", [Form, T]), fail.
evidence 119
suspected 107
suspected 119
false.

?- stack_overflow(T, evidence(_, _, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
exception_record(119,3221225725,140718400126994)
T = 119 ;
false.

?- stack_overflow(T, suspected(_, T, _Gs)),
   forall(member(G, _Gs), ( print(G), nl )),
   nl, fail.
thread(107,4120)
stack_depth(107,14)
threshold(stack_depth,12)
frame_regularity(107,reportengine-'Node::Visit',8)
threshold(overflow_run,6)

thread(119,4120)
stack_depth(119,13)
threshold(stack_depth,12)
frame_regularity(119,reportengine-'Node::Visit',7)
threshold(overflow_run,6)

false.
```

Thread 119 has the status code and is an instance; it is also suspected by the second clause, since its trace is deep and regular, and the two findings for it are a recognition and a suspicion of one fact, told apart by their functors, which `functor/3` extracts in the first query. Thread 107 of the shared dump is suspected and nothing more: fourteen frames with eight of `Node::Visit`, running, no exception. The grounds are five goals: the thread fact the clause looked up, the depth as `stack_depth/2` computes it, the run as `frame_regularity/3` finds it, and the two thresholds the clause compared them with; each can be called, and the comparisons re-derive from the five. The

run threshold `frame_run` that `frame_regularity/3` applied is not listed, because that goal is a derived leaf which stands for its own derivation, as `cpu_share/2` stands for the times and the total in Chapter 17. The thresholds are set for a dump of this size; a real stack overflow has thousands of frames, and the depth threshold of twelve exists so that the toy data has a suspect. The failure-driven loops print each finding and end in `false.`, which is the loop's own failure and not a verdict.

The contract of Section 17.3 is checked on both forms: every finding is ground, every goal in it holds, and the shape re-derives from the list alone. For the evidence term the shape is the status code among the facts, and for the suspicion term it is the two comparisons between the listed values:

```
?- stack_overflow(T, evidence(_, _, _Fs)),
   ground(_Fs), maplist(call, _Fs),
   memberchk(exception_record(T, 0xc00000fd, _),
             _Fs).
T = 119 ;
false.

?- stack_overflow(T, suspected(_, _, _Gs)),
   ground(_Gs), maplist(call, _Gs),
   memberchk(stack_depth(T, D), _Gs),
   memberchk(threshold(stack_depth, MinD), _Gs),
   memberchk(frame_regularity(T, _, C), _Gs),
   memberchk(threshold(overflow_run, MinC), _Gs),
   D >= MinD, C >= MinC.
T = 107,
D = 14,
MinD = 12,
C = 8,
MinC = 6 ;
T = 119,
D = 13,
MinD = 12,
C = 7,
MinC = 6 ;
false.
```

The `memberchk/2` goals take the values from the list and not from the knowledge base, so the two comparisons at the end are between listed values, which is the re-derivation Chapter 17 asks for. A consumer that counts recognized instances counts one, thread 119; a consumer that lists hypotheses for the analyst lists two, and the report of Chapter 23 keeps the two lists apart.

What the second clause does not know is whether the recursion is bounded. A tree walk of depth eight on a tree of depth eight is correct, and the same walk on a cyclic graph never ends; the dump shows one moment of either, and only the exception code, or a second dump in which the depth has grown, tells them apart.

19.9.1 The Two Run Thresholds

The difference between the two policies is observable, and the following checks it on synthetic stacks that the extension does not have, added for the check and taken away after it. `synthetic_stack/2`

adds a thread of the process whose frames name the given functions, top frame first, all in `appserver`; the three relations it adds to were declared dynamic in `faults.pl`. Five stacks are built: three of twelve frames whose longest run is four, five, and six; one of eight frames with a run of six; and one of three frames with the status code and no regularity. `run_report/1` prints, for one thread, its depth, the runs `frame_regularity/3` enumerates for it, and the functors of the findings `stack_overflow/2` reports:

```
% synthetic_stack(TID, Functions): add thread TID
% to the process with one frame per function of
% Functions, top frame first, all in appserver.
synthetic_stack(TID, Functions) :-
    assertz(thread(TID, 4120)),
    forall(nth0(N, Functions, F),
           assertz(frame(TID, N, appserver, F,
                         0x10))).

% run_report(TID): the depth of TID, the runs that
% reach frame_run, and the forms of its findings.
run_report(TID) :-
    stack_depth(TID, Depth),
    findall(C, frame_regularity(TID, _, C), Runs),
    findall(Form,
            ( stack_overflow(TID, F),
              functor(F, Form, 3)
            ),
            Forms),
    format("~d depth ~d runs ~w ~w~n",
           [TID, Depth, Runs, Forms]).


?- synthetic_stack(140,
       [f, f, f, f, g, h, i, j, k, l, m, n]),
   synthetic_stack(141,
       [f, f, f, f, f, g, h, i, j, k, l, m]),
   synthetic_stack(142,
       [f, f, f, f, f, f, g, h, i, j, k, l]),
   synthetic_stack(143, [f, f, f, f, f, f, g, h]),
   synthetic_stack(144, [a, b, c]),
   assertz(exception_record(144, 0xc00000fd,
                            0x7ff6_1d11_0010)).
true.

?- threshold(frame_run, R),
   threshold(overflow_run, O).
R = 4,
O = 6.
```

```
?- forall(member(T, [107, 119, 140, 141, 142,
                     143, 144]),
          run_report(T)).
107 depth 14 runs [8] [suspected]
119 depth 13 runs [7] [evidence,suspected]
140 depth 12 runs [4] []
141 depth 12 runs [5] []
142 depth 12 runs [6] [suspected]
143 depth 8 runs [6] []
144 depth 3 runs [] [evidence]
true.

?- stack_overflow(T, suspected(_, _, _)),
   T >= 140.
T = 142 ;
false.
```

The three twelve-frame stacks are all **Frame Regularity** (page 342) instances, since their runs reach four, and only the one whose run reaches six is a suspected overflow; a recognizer that compared the run with `frame_run` would suspect all three, and would suspect the stacks with runs of four and five that this section's rule rejects. The eight-frame stack has a run of six and is not deep enough. The three-frame stack with the status code is an instance by the first clause, which asks nothing of depth or regularity: the status is conclusive on its own. The facts are then removed, and the knowledge base is as the extension left it:

```
?- forall(between(140, 144, T),
          ( retractall(thread(T, _)),
            retractall(frame(T, _, _, _, _)),
            retractall(exception_record(T, _, _))
          )),
   aggregate_all(count, thread(_, _), N),
   aggregate_all(count,
                 exception_record(_, _, _), E).
N = 19,
E = 9.
```

19.10 Divide by Zero (User Mode)

Divide by Zero (User Mode) is the status `0xc0000094`, raised when an integer division instruction is given a zero divisor. The recognizer is the status code, and the interest of the pattern is in where the zero came from:

```
% divide_by_zero(TID): TID divided an integer by
% zero.
divide_by_zero(TID) :-
    exception_record(TID, 0xc0000094, _).
```

```
?- divide_by_zero(T), print_stack_trace(T).
00 reportengine!Stats::Mean+0x40
01 reportengine!Report::Summarize+0x88
02 appserver!Worker::Run+0x120
03 ucrtbase!thread_start<...>+0x42
04 kernel32!BaseThreadInitThunk+0x14
05 ntdll!RtlUserThreadStart+0x21
T = 111.
```

A mean of zero samples is the usual story, and the function name of the synthetic thread 111 tells it. The code is the integer division's; floating-point division by zero raises no exception unless the program enables it, and then a different status, which the `status_name/2` table does not list. The predicate says that a division by zero happened and in which function; the divisor's origin, a count never incremented or a size read from a corrupt structure, is in the registers and the source, not in the facts.

19.11 Nested Exceptions (User Space)

Nested Exceptions (User Space) are an exception raised while an earlier exception was being dispatched or handled: the stack shows the dispatcher's frames, `KiUserExceptionDispatch` and the functions it calls, below the handler, and above the handler a second raise. The second exception is the one the dump records, and the first is visible only in the frames. The recognizer finds a dispatch frame below the top and checks that the top frame is the current exception's own, so that the trace was recorded from the current fault and the dispatch frame below belongs to an earlier one:

```
% dispatch_frame(TID, N): frame N of TID is the
% user-mode exception dispatcher.
dispatch_frame(TID, N) :-
    frame(TID, N, ntdll, F, _),
    F == 'KiUserExceptionDispatch'.

% nested_exceptions(TID): the exception of TID was
% raised while an earlier exception of TID was
% being dispatched.
nested_exceptions(TID) :-
    exception_record(TID, _, Addr),
    dispatch_frame(TID, N),
    N > 0,
    frame(TID, 0, M, _, _),
    address_module(Addr, M).

?- nested_exceptions(T), print_stack_trace(T).
00 kernelbase!RaiseException+0x6e
01 vcruntime140!_CxxThrowException+0x90
02 appserver!Handler::Report+0x66
03 appserver!Handler::OnFault+0x2e
04 ntdll!RtlpExecuteHandlerForException+0xf
05 ntdll!RtlDispatchException+0x244
06 ntdll!KiUserExceptionDispatch+0x2e
07 dbclient!Buffer::Copy+0x1a
08 dbclient!Connection::Receive+0x8c
09 appserver!Session::Fetch+0x44
```

```
10 appserver!Worker::Run+0x120
11 ucrtbase!thread_start<...>+0x42
12 kernel32!BaseThreadInitThunk+0x14
13 ntdll!RtlUserThreadStart+0x21
T = 114 ;
false.
```

The synthetic thread 114 faulted in `Buffer::Copy` as thread 109 did, the dispatcher found a handler in `Handler::OnFault`, and the handler threw a C++ exception while reporting the fault; the recorded exception is the C++ one, raised in frame 0, and the access violation is the one frame 7 was executing when frame 6 took over. The trace reads as two exception stack traces stacked, which is what the pattern names.

The predicate knows one dispatcher function, the user-mode one of one operating system family. It does not recover the first exception's record, which the dispatcher passed to the handler as a parameter and which lies on the stack; Section 19.17 treats such records when the parser has found them.

19.12 Multiple Exceptions (User Space)

Multiple Exceptions (User Space) are exception records on more than one thread of the same process: either several threads faulted independently, which points at a shared corrupt structure or a systemic cause, or a dump was taken late enough for a first fault to have provoked others. The recognizer is a count:

```
% multiple_exceptions(TIDs): more than one thread
% has an exception record; TIDs lists them.
multiple_exceptions(TIDs) :-
    findall(TID, exception_record(TID, _, _),
            TIDs),
    length(TIDs, Count),
    Count > 1.

?- multiple_exceptions(_TIDs), length(_TIDs, N).
N = 9.

?- findall(T,
       ( exception_record(T, _, _), T =< 109 ),
   Ts), length(Ts, N).
Ts = [109],
N = 1.
```

The extended knowledge base has nine faulting threads by construction; restricted to the threads of the shared dump it has one, and the pattern does not apply to `dump1.pl`. The predicate counts records and does not compare them; whether the exceptions have the same code, module, or crash signature is the **Crash Signature** (page 380) of Section 19.20 applied to each thread of the list.

A debugger shows one current exception record, the one that stopped the process, and the records of other threads are found by switching to each thread and inspecting its context. The schema's

`exception_record/3` facts are whatever the parser found by doing that, and a dump in which the other threads were not inspected has one fact and no instance of this pattern, however many threads faulted.

19.13 Hidden Exception (User Space)

A **Hidden Exception (User Space)** is an exception the dump does not report as current but whose traces are on a thread's stack: the thread is in a handler or has returned from one, or the raw stack holds the dispatcher's frames as execution residue. In the schema it shows as a thread without an exception record whose trace contains an exception frame:

```
% exception_frame(F): F is a function that raises
% or dispatches an exception.
exception_frame('KiUserExceptionDispatch').
exception_frame('RaiseException').

% hidden_exception(TID, N): TID has no exception
% record and frame N of its trace is an exception
% frame.
hidden_exception(TID, N) :-
    thread(TID, _),
    \+ exception_record(TID, _, _),
    frame(TID, N, _, F, _),
    exception_frame(F).

?- hidden_exception(T, N).
T = 115,
N = 6 ;
false.

?- hidden_exception(T, _), print_stack_trace(T).
00 ntdll!NtWaitForSingleObject+0x14
01 kernelbase!WaitForSingleObjectEx+0x8e
02 appserver!Logger::Flush+0x72
03 appserver!Handler::OnFault+0x4a
04 ntdll!RtlpExecuteHandlerForException+0xf
05 ntdll!RtlDispatchException+0x244
06 ntdll!KiUserExceptionDispatch+0x2e
07 appserver!Session::Parse+0xb0
08 appserver!Worker::Run+0x120
09 ucrtbase!thread_start<...>+0x42
10 kernel32!BaseThreadInitThunk+0x14
11 ntdll!RtlUserThreadStart+0x21
T = 115 ;
false.
```

Thread 115 is waiting for a log flush and would be classed as a **Passive Thread** (page 334) by Chapter 18; its trace says that it is waiting inside the handler of an exception that `Session::Parse` raised, and the dump has no record of that exception because the dispatcher found a handler and the process did not stop for it. A thread in this state is the explanation of a hang that a reader of `thread_state/2` would miss.

The predicate sees exception frames that are live. The other form of the pattern, dispatcher frames left on the raw stack by an exception handled and returned from, is execution residue in the sense of Chapter 18 and needs the `raw_stack/3` extension there, with `symbol_like/3` tested against the addresses of these functions, which the schema does not record.

19.14 Handled Exception (User Space)

A **Handled Exception (User Space)** is one for which the dispatcher found a handler that ran: the exception did not bring the process down, and its effect, if any, is in what the handler did. In a trace it shows as application code above the dispatcher's frames, the handler being the topmost non-system frame between the dispatcher and the top. The recognizer finds the dispatch frame and the nearest application frame above it:

```
% handled_exception(TID, Handler): the dispatcher
% on the stack of TID has called a handler, whose
% Module-Function is Handler.
handled_exception(TID, Handler) :-
    dispatch_frame(TID, N),
    aggregate_all(max(H),
                  ( frame(TID, H, M, _, _),
                    H < N,
                    module(M, _, _),
                    \+ system_module(M)
                  ),
                  Top),
    frame(TID, Top, HM, HF, _),
    Handler = HM-HF.

?- handled_exception(T, H).
T = 114,
H = appserver-'Handler::OnFault' ;
T = 115,
H = appserver-'Handler::OnFault' ;
false.

?- handled_exception(T, _),
   hidden_exception(T, _).
T = 115 ;
false.

?- handled_exception(T, _), nested_exceptions(T).
T = 114 ;
false.
```

`aggregate_all(max(H), ...)` finds the deepest application frame above the dispatcher, which is the handler as the dispatcher called it; the frames above the handler are what the handler did. Both threads with a dispatch frame have the same handler, and the two further queries place them: thread 115's handled exception is hidden, since the process did not stop, and thread 114's handler raised a second exception, which is the nesting of Section 19.11. The three patterns describe one mechanism from three sides.

The predicate finds a handler that is still on the stack. An exception handled and returned from leaves no live frames, whether the handler unwound the stack or resumed execution at the faulting instruction; such exceptions are beyond these facts, and the software trace of Part IV, where a handler usually logs, is where they are recorded.

19.15 C++ Exception

A **C++ Exception** is one thrown by the C++ `throw` statement, which the runtime implements as a software exception with the status `0xe06d7363`, the bytes of `msc` with a high byte, raised through `RaiseException`. The exception address is therefore in `kernelbase`, and the thrower is the first application frame below the runtime's throw routine. The predicate is named `cpp_exception/1`, since `++` cannot appear in an unquoted atom:

```
% cpp_exception(TID): TID raised a C++ exception.
cpp_exception(TID) :-
    exception_record(TID, 0xe06d7363, _).

?- cpp_exception(T).
T = 112 ;
T = 114.

?- cpp_exception(112), print_stack_trace(112).
00 kernelbase!RaiseException+0x6e
01 vcruntime140!_CxxThrowException+0x90
02 appserver!Parser::Expect+0x5a
03 appserver!Parser::Parse+0x1c4
04 appserver!Worker::Run+0x120
05 ucrtbase!thread_start<...>+0x42
06 kernel32!BaseThreadInitThunk+0x14
07 ntdll!RtlUserThreadStart+0x21
true ;
false.
```

Thread 112 threw from `Parser::Expect` and nothing caught it; thread 114 threw from its exception handler. The status code identifies the mechanism and not the type of the object thrown, which is in the exception parameters as a pointer to the object and its type descriptor; the schema's `exception_info/3` has keys for access violations only, and the type of the C++ exception is what an analyst recovers next from the parameters. A C++ exception that was caught never produces a dump, and one that is recorded is unhandled or was dumped at first chance; the predicate cannot tell these apart.

19.16 Managed Code Exception

A **Managed Code Exception** is one raised by the .NET runtime on behalf of managed code, as a software exception with the status `0xe0434352`, the bytes `CCR` with a high byte, through the same `RaiseException`. The native trace shows the runtime's throw path and, below it, frames in no module where the JIT-compiled method was executing:

```
% managed_code_exception(TID): TID raised a .NET
% runtime exception.
managed_code_exception(TID) :-
    exception_record(TID, 0xe0434352, _).

?- managed_code_exception(T),
   print_stack_trace(T).
00 kernelbase!RaiseException+0x6e
01 coreclr!RaiseTheExceptionInternalOnly+0x2a8
02 coreclr!IL_Throw+0x1c0
03 <no module>
04 kernel32!BaseThreadInitThunk+0x14
05 ntdll!RtlUserThreadStart+0x21
T = 113.
```

The code says that the exception is the runtime's; which managed exception it is, and which managed method threw it, the native trace does not say. Frame 3 is JIT-compiled code, outside every module and printed without an address because the extension recorded none; the managed call stack is invisible to a native stack walker, and the runtime's own debugger extension reconstructs it from the runtime's data structures. The schema has no representation for a managed frame beyond `none`, and Chapter 15 said that a knowledge base of managed dumps needs one.

19.17 Stored Exception

A **Stored Exception** is an exception record that lies in the process's memory rather than being the one the debugger reports as current: records saved by handlers, by the runtime, or by the dispatcher on the stack, which `.exr` can decode when their addresses are known. They record exceptions that happened earlier, or the current one in a second copy. The schema does not hold them, and the extension `stored_exception/4` records what a parser that searched for them would have found:

```
% past_exception(TID, Code, Addr): a stored record
% of TID describes an exception that is not its
% current one.
past_exception(TID, Code, Addr) :-
    stored_exception(TID, _, Code, Addr),
    \+ exception_record(TID, Code, Addr).

?- stored_exception(T, _, C, A),
   format("~d ~16r at ~16r~n", [T, C, A]), fail.
109 c0000005 at 7ffb8d1031aa
115 c0000005 at 7ff61d11b0c8
false.

?- past_exception(T, C, A), address_module(A, M).
T = 115,
C = 3221225477,
A = 140695026381000,
M = appserver ;
false.
```

```
?- past_exception(115, _, A),
   address_module(A, M), module(M, _Base, _),
   Off is A - _Base,
   format("~a+0x~16r~n", [M, Off]).
appserver+0x1b0c8
A = 140695026381000,
M = appserver,
Off = 110792 ;
false.
```

Two records were found. Thread 109's is a copy of its current exception, as the dispatcher stores one before calling handlers, and `past_exception/3` excludes it; thread 115's is the access violation that its handler is handling, which Section 19.13 could infer only from the frames and which the stored record now places: in `appserver` at offset `0x1b0c8`, in `Session::Parse` by the trace. The stored record is the exception the dump does not report, recovered.

The predicate relies on a parser having found the records, which a debugger does not do by itself; the analyst points `.exr` at an address, and the knowledge base records the result. Nothing in the facts says when a stored exception happened, and a record in a stale stack region can be from an exception handled long before the dump, so that `past_exception/3` reports candidates, as the residue predicates of Chapter 18 did.

19.18 Exception Module

The **Exception Module** is the module containing the instruction at which the exception was raised: the module of the exception address, and the first answer to "whose code faulted".

```
% exception_module(TID, Module): the exception of
% TID was raised in Module, or in none.
exception_module(TID, Module) :-
    exception_record(TID, _, Addr),
    (   address_module(Addr, M)
    ->  Module = M
    ;   Module = none
    ).

?- exception_module(109, M).
M = dbclient.

?- exception_module(T, kernelbase).
T = 112 ;
T = 113 ;
T = 114 ;
false.

?- exception_module(T, none).
T = 110 ;
T = 117 ;
false.
```

The access violation of the shared dump was raised in `dbclient`. The three software exceptions were raised in `kernelbase`, since `RaiseException` is where every such exception is raised, and that is the pattern's limit: the exception module of a thrown exception is the runtime's, not the thrower's, and the module that matters is the one below it on the stack. The two threads in no module are the null pointer call and the wild code. The next pattern looks past the raising module.

19.19 Problem Module

The **Problem Module** is the module most likely responsible for a fault: the topmost frame on the exception stack trace that belongs to the application rather than to the system or a runtime, on the reasoning that system code faults because of what it was given. The list of system modules is a fact of the chapter, and the recognizer takes the application frame with the smallest number:

```
% problem_frame(TID, Frame): Frame is the topmost
% frame of TID in a module that is not a system
% module.
problem_frame(TID, frame(TID, N, M, F, O)) :-
    exception_record(TID, _, _),
    aggregate_all(min(N0),
                  ( frame(TID, N0, M0, _, _),
                    module(M0, _, _),
                    \+ system_module(M0)
                  ),
                  N),
    frame(TID, N, M, F, O).

% problem_module(TID, Module): Module is the
% module of the problem frame of TID.
problem_module(TID, Module) :-
    problem_frame(TID, frame(_, _, Module, _, _)).

?- problem_module(T, M), exception_module(T, EM),
   M \== EM.
T = 110,
M = appserver,
EM = none ;
T = 112,
M = appserver,
EM = kernelbase ;
T = 114,
M = appserver,
EM = kernelbase ;
T = 117,
M = appserver,
EM = none ;
false.

?- problem_module(113, M).
false.
```

```
?- problem_module(109, M),
   module_info(M, company, C),
   module_info(M, version, V).
M = dbclient,
C = 'Example Corp',
V = '3.1.0.7' ;
false.
```

The problem module differs from the exception module exactly where it should: for the thrown exceptions it names the thrower, and for the two faults outside every module it names the caller. The managed exception of thread 113 has no problem module among its native frames, which is correct, since the responsible code is managed and the predicate's `module(M0, _, _)` test excludes the `none` frame. For the shared dump's fault the problem module is `dbclient`, and the `module_info/3` facts give its vendor and the version that Chapter 22 finds does not match its neighbors.

The reasoning behind the pattern is a heuristic and the predicate makes it explicit. System code does fault on its own, when a kernel or runtime bug is the cause, and a third-party module above the application's frames may be the real culprit when it passed bad data down; the `system_module/1` list is the analyst's division of the modules into suspects and non-suspects, and changing it changes the answer.

19.20 Crash Signature

A **Crash Signature** is a canonical description of a fault short enough to compare across dumps: the kind of exception and the problem frame's module, function, and offset, which together identify the crash the way a bucket in a crash-reporting system does. Two dumps with the same signature are the same crash for most purposes, and a set of dumps is sorted by signature before any is read in detail.

```
% crash_signature(TID, Signature): Signature is
% the canonical term sig(Kind, Module, Function,
% Offset) of the fault of TID.
crash_signature(TID, sig(Kind, M, F, O)) :-
    exception_record(TID, Code, _),
    (   status_name(Code, Kind)
    ->  true
    ;   Kind = Code
    ),
    (   problem_frame(TID, frame(_, _, M, F, O))
    ->  true
    ;   M = none,
        F = 'Unknown',
        O = 0
    ).
```

```
% print_signature(TID): the signature of TID in
% the debugger's notation.
print_signature(TID) :-
    crash_signature(TID, sig(Kind, M, F, O)),
    format("~d ~a ~a!~a+0x~16r~n",
           [TID, Kind, M, F, O]).

?- crash_signature(109, sig(K, M, F, O)).
K = access_violation,
M = dbclient,
F = 'Buffer::Copy',
O = 26.

?- forall(exception_record(T, _, _),
          print_signature(T)).
109 access_violation dbclient!Buffer::Copy+0x1a
110 access_violation appserver!Plugin::Invoke+0x3c
111 divide_by_zero reportengine!Stats::Mean+0x40
112 cpp_exception appserver!Parser::Expect+0x5a
113 clr_exception none!Unknown+0x0
114 cpp_exception appserver!Handler::Report+0x66
116 access_violation dbclient!Row::Get+0x32
117 access_violation appserver!Parser::Parse+0x1c4
119 stack_overflow reportengine!Node::Visit+0x12
true.

?- crash_signature(109, _S),
   crash_signature(T, _S), T \== 109.
false.
```

The nine faults have nine signatures, and the last query, which looks for another thread with thread 109's signature, finds none: the extension's threads are different crashes, as they were designed to be, and across the knowledge bases of several dumps the same query is how duplicates are found. The signature of a stack overflow is the recursing function and that of a thrown exception the thrower, which are the right identities for those crashes; the signature of the managed exception is empty, since the identity of a managed crash is in the managed frames.

The signature is canonical only to the extent its parts are stable. The offset changes with every build of the module, so that two dumps of the same crash from two versions have different signatures, and a signature without the offset merges crashes at different places in a large function; the predicate fixes one choice that an analyst can change. A signature is also only as good as the problem frame, and a fault that corrupted the stack gives a signature that means nothing; Section 19.8 is the check to run first.

Chapter 20. Synchronization Patterns

The patterns of this chapter concern threads that wait for one another: the chains of waits that run from a blocked thread through the owner of the object it waits for to whatever that owner waits for in turn, the cycles in those chains that are deadlocks, the queues that form on one contended object, the state of the objects themselves, and the time and priority facts that say how long the waiting has gone on and why some threads never got their turn. The shared dump has three critical sections and three threads waiting on two of them, which is enough for the critical section patterns and the deadlock; the other kinds of object are supplied by extensions.

The chapter starts a fresh toplevel and consults `dump1.pl`. Its central device is the **wait-for graph**, whose vertices are threads and whose edges run from a waiting thread to the thread that owns the object it waits for; a wait chain is a path in that graph and a deadlock is a cycle. Chapter 10 built the graph as a ugraph and Chapter 11 closed it with tabling; here both formulations of the cycle test are given in full, the recursive walk with a visited list and the tabled transitive closure, and the chapter says what each guarantees.

```
:- [dump1].

% threshold(Name, Value): the thresholds.
threshold(queue_length, 2).
threshold(contention, 2).
threshold(swarm_size, 4).
threshold(spiking_share, 25).

% system_module(M): M belongs to the operating
% system or a runtime, not to the application.
system_module(ntdll).
system_module(kernelbase).
system_module(kernel32).
system_module(ucrtbase).
system_module(msvcp140).

% pattern(Name, Artifact, Chapter): the patterns
% this chapter recognizes with a predicate.
pattern('Wait Chain (Critical Sections)',
        memory_dump, 20).
pattern('Wait Chain (Mutex Objects)',
    memory_dump, 20).
pattern('Wait Chain (Thread Objects)',
    memory_dump, 20).
pattern('Deadlock (Critical Sections)',
    memory_dump, 20).
pattern('Deadlock (Mixed Objects)',
    memory_dump, 20).
pattern('Blocked Queue', memory_dump, 20).
pattern('High Contention (Critical Sections)',
        memory_dump, 20).
pattern('Critical Section Corruption',
    memory_dump, 20).
pattern('Swarm of Shared Locks', memory_dump, 20).
pattern('Waiting Thread Time', memory_dump, 20).
```

```
pattern('Thread Starvation (Normal Priority)',
        memory_dump, 20).
pattern('Blocking Module', memory_dump, 20).
```

The extension this chapter needs most is other kinds of synchronization object. The schema's `waits_for/3` has the kind `critical_section` only, because `!cs` is the one command whose output the parser of Chapter 15 turned into wait facts; mutexes and thread handles are waited for through `WaitForSingleObject`, and the object a thread waits for is found from `!handle` and the thread's parameters. The extension adds `mutant/3`, the kernel's name for a mutex object, with its handle, owner and name, and `waits_for_object/3`, which has the shape of `waits_for/3` and holds the waits of the kinds the shared file lacks. `wait_fact/4` relates a wait to the fact that records it, in whichever of the two relations it lies, so that an explainer can cite the fact rather than the view; the view `waits/3` presents both relations as one. The thresholds are `threshold/2` facts and the patterns are registered with `pattern/3`, as Chapter 17 prescribes. The threads 120 to 124 are not in `dump1.pl`, and none of the predicates below consults `thread/2` for them.

```
% The two extension relations are dynamic, so that
% the boundary checks of Section 20.5 can add
% facts and take them away again.
:- dynamic mutant/3, waits_for_object/3.

% mutant(Handle, OwnerTID, Name): a mutex object
% with its owner. An extension of the schema.
mutant(0x1a8, 120, 'AppServer.CacheMutex').
mutant(0x1b4, 123, 'AppServer.LogMutex').

% waits_for_object(TID, Kind, Object): as
% waits_for/3, for Kinds mutant and thread. An
% extension of the schema.
waits_for_object(120, thread, 103).
waits_for_object(121, mutant, 0x1a8).
waits_for_object(122, thread, 121).
waits_for_object(123, thread, 124).
waits_for_object(124, mutant, 0x1b4).

% wait_fact(TID, Kind, Object, Fact): Fact is the
% fact, of the shared file or of the extension,
% that records that TID waits for Object of Kind.
wait_fact(T, K, O, waits_for(T, K, O)) :-
    waits_for(T, K, O).
wait_fact(T, K, O, waits_for_object(T, K, O)) :-
    waits_for_object(T, K, O).

% waits(TID, Kind, Object): TID waits for Object
% of Kind, by the shared facts or the extension.
waits(TID, Kind, Object) :-
    wait_fact(TID, Kind, Object, _).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, waits(_, _, _), N).
N = 8.
```

```
?- aggregate_all(set(K), waits(_, K, _), Kinds).
Kinds = [critical_section, mutant, thread].
```

Eight threads wait for objects of three kinds. Thread 120 holds the cache mutex and waits for thread 103 to finish, which it never will; thread 121 waits for the mutex; 122 waits for 121; and threads 123 and 124 wait for each other through a mutex and a thread handle. The rest of the chapter derives these sentences from the facts.

20.1 Wait Chain (Critical Sections)

A **Wait Chain (Critical Sections)** is a sequence of threads in which each waits to enter a critical section owned by the next: the first thread is blocked, the second holds what the first needs and is itself blocked, and the chain ends at a thread that is not waiting for a critical section, which is the thread to look at, or closes on itself, which is the deadlock of Section 20.4. The debugger's `!cs` and `!locks` give the owners and `~*k` the waiters, and the analyst follows the chain by hand.

The edge of the wait-for graph is one clause per kind of object, since each kind records its owner differently: a critical section in `critical_section/4`, a mutex in `mutant/3`, and a thread object in the wait fact itself, since the thread waited for is the owner. An edge carries the facts that make it, the wait fact and the ownership fact, so that the explainer of Section 20.4 can cite them; `wait_edge/3` is the edge without its facts. `edge_of/4` selects edges by kind, with the atom `any` for every kind, and hands back the edge it selected as a pair `Kind-Facts`, which is what the walks of Sections 20.4 and 20.5 keep as their witness; `edge_of/3` drops the pair. The chain follows edges from a waiting thread as far as it can without visiting a thread twice:

```
% wait_edge(Kind, Waiter, Owner, Facts): Waiter
% waits for an object of Kind that Owner holds;
% Facts are the wait fact and the ownership fact
% that say so.
wait_edge(critical_section, W, O,
          [Wait, critical_section(A, O, L, R)]) :-
    wait_fact(W, critical_section, A, Wait),
    critical_section(A, O, L, R),
    O =\= 0.
wait_edge(mutant, W, O,
          [Wait, mutant(H, O, N)]) :-
    wait_fact(W, mutant, H, Wait),
    mutant(H, O, N),
    O =\= 0.
wait_edge(thread, W, O, [Wait]) :-
    wait_fact(W, thread, O, Wait).

% wait_edge(Kind, Waiter, Owner): the edge without
% its facts.
wait_edge(Kind, W, O) :-
    wait_edge(Kind, W, O, _).
```

```
% edge_of(+Kind, ?W, ?O, -Edge): a wait edge of
% Kind, or of any kind when Kind is the atom any;
% Edge is K-Facts, the kind of the edge and the
% facts that make it.
edge_of(Kind, W, O, K-Facts) :-
    must_be(oneof([critical_section, mutant,
                   thread, any]), Kind),
    (   Kind == any
    ->  wait_edge(K, W, O, Facts)
    ;   K = Kind,
        wait_edge(K, W, O, Facts)
    ).

% edge_of(+Kind, ?W, ?O): the edge without its
% kind and facts.
edge_of(Kind, W, O) :-
    edge_of(Kind, W, O, _).

% wait_chain(+Kind, -Chain): Chain is the path of
% Kind edges from a waiting thread, followed until
% the next thread would repeat one in the chain.
wait_chain(Kind, [T | Rest]) :-
    edge_of(Kind, T, _),
    follow(Kind, T, [T], Rest).

follow(Kind, T, Visited, Chain) :-
    (   edge_of(Kind, T, Next),
        \+ memberchk(Next, Visited)
    ->  Chain = [Next | Rest],
        follow(Kind, Next, [Next | Visited], Rest)
    ;   Chain = []
    ).

% print_chain(Chain): the threads of Chain joined
% by arrows.
print_chain(Chain) :-
    atomic_list_concat(Chain, ' -> ', Text),
    format("~a~n", [Text]).

?- wait_edge(critical_section, W, O).
W = 101,
O = 103 ;
W = 103,
O = 104 ;
W = 104,
O = 103 ;
false.

?- wait_edge(critical_section, 103, O, _Facts),
   forall(member(F, _Facts), ( print(F), nl )).
waits_for(103,critical_section,140695026508096)
critical_section(140695026508096,104,1,1)
O = 104 ;
false.
```

```
?- wait_chain(critical_section, C),
   print_chain(C), fail.
101 -> 103 -> 104
103 -> 104
104 -> 103
false.

?- catch(wait_chain(mutex, _),
   error(type_error(_, Kind), _), true).
Kind = mutex.
```

The `O =\= 0` in the first two clauses excludes an unowned object, which blocks nobody; a thread recorded as waiting for an unowned critical section is itself a finding, treated in Section 20.8. The second query shows the facts behind one edge: the wait of thread 103 as the shared file records it and the section it waits for, with its owner and counts. `must_be/2` fixes the mode of `edge_of/4`: the kind is an input, and a kind the chapter does not know raises an error naming the admissible values rather than failing silently, as Chapter 7 recommended for lookups. Under the kind `any` the pair `K-Facts` records which kind the edge turned out to have, since the caller did not say. The chain follows one edge per thread, since a thread waits for one object at a time, and the if-then-else commits to it; the visited list, tested with `memberchk/2`, is what makes `follow/4` terminate on the cycle of threads 103 and 104, whose chains end when the next thread is already in the chain.

The three chains of the shared dump are the whole synchronization story of its critical sections: thread 101 waits for 103, which waits for 104, which waits for 103. The chain from 101 ends at 104 because 103 is already in it, and the two chains of length two are the cycle seen from either end; Section 20.4 recognizes it as such. What the predicate does not say is why each owner holds its section, which is in the owner's stack trace, or how long the chain has existed, which Section 20.11 treats; it gives the shape, and the analyst reads the frames.

20.2 Wait Chain (Mutex Objects)

A **Wait Chain (Mutex Objects)** is the same structure through mutexes: a thread waits in `WaitForSingleObject` on a mutex handle, the mutex has an owner, and the owner waits in turn. The debugger shows the handle with `!handle` and the owner in the object's data, and the analyst joins the two by hand; the extension facts record the result, and the kind argument selects the edges:

```
?- wait_chain(mutant, C), print_chain(C), fail.
121 -> 120
124 -> 123
false.

?- wait_chain(mutant, C), last(C, Owner),
   mutant(H, Owner, Name), format("~16r~n", [H]).
1a8
C = [121, 120],
Owner = 120,
H = 424,
Name = 'AppServer.CacheMutex' ;
```

```
1b4
C = [124, 123],
Owner = 123,
H = 436,
Name = 'AppServer.LogMutex'.
```

Each mutex chain has two threads, because the owners wait for objects of another kind, and a chain of one kind stops where the kind changes. Thread 121 waits for the cache mutex, which thread 120 holds; what 120 waits for is a thread object, and the chain of this kind ends there. The single-kind chain is what each debugger command shows, and the limitation is the command's; the `any` kind of `edge_of/3` removes it:

```
?- wait_chain(any, [121 | Rest]),
   print_chain([121 | Rest]).
121 -> 120 -> 103 -> 104
Rest = [120, 103, 104] ;
false.
```

Thread 121 waits for a mutex held by 120, which waits for thread 103 to exit, which waits for a critical section held by 104, which waits for one held by 103. The chain across kinds reaches the deadlock, and that is the explanation of why 121 is blocked, which no single-kind chain gives. A mutex abandoned by a thread that exited while holding it has no owner in the dump, and `wait_edge/3` has no edge for it; the waiters of an abandoned mutex are released by the system with a distinct status, so that a thread still waiting on one is a different finding, which the extension does not record.

20.3 Wait Chain (Thread Objects)

A **Wait Chain (Thread Objects)** runs through waits on threads themselves: a thread waits for another to exit, with `WaitForSingleObject` on a thread handle, as a thread that started a worker and joins it, and the thread waited for is blocked in turn. The owner of a thread object is the thread, so the edge is the wait fact:

```
?- wait_chain(thread, C), print_chain(C), fail.
120 -> 103
122 -> 121
123 -> 124
false.

?- wait_chain(any, [122 | Rest]),
   print_chain([122 | Rest]).
122 -> 121 -> 120 -> 103 -> 104
Rest = [121, 120, 103, 104] ;
false.
```

Thread 120 joins thread 103, a worker that will never exit; 122 joins 121, which is blocked on the mutex 120 holds; and the chain of any kind from 122 is the longest in the knowledge base, five threads and three kinds of object, ending in the deadlock. Thread 123 waits for 124, and Section 20.5 says what 124 waits for.

A thread that waits for a thread to exit waits for everything that thread waits for, which is why these chains are long; and a thread object that is signaled, because the thread has exited, releases its waiters, so that a thread still waiting on an exited thread is impossible unless the dump caught the moment, and `wait_edge/3` would then name an owner that `thread/2` does not know. The predicate assumes that the thread waited for still exists, and the schema check that it does, `forall(waits(_, thread, T), thread(T, _))`, fails on the extension by design, since threads 121 and 124 are not in `dump1.pl`.

20.4 Deadlock (Critical Sections)

A **Deadlock (Critical Sections)** is a cycle in the wait-for graph: each thread of the cycle waits for a critical section held by the next, and the last for one held by the first, so that none can proceed and no external event can release them. It is the one synchronization pattern whose recognition is a proof, since the facts about ownership and waiting entail the cycle, and the two formulations here are two ways of searching for it.

The first is a recursive walk from each waiting thread along edges until the walk returns to its start, with a visited list to stop it from circling a cycle that does not contain the start. The cycle is reported from its smallest thread so that each is reported once. The walk keeps a **witness**: the edge it followed from each thread, as `edge_of/4` selected it, with the kind and the two facts that make it, so that the explainer later in this section and the naming of Section 20.5 rest on the edges the walk used and on no others. `deadlock_witness/3` is the walk with its witness, and the enumerator `deadlock/2` drops the witness, as `double_free/1` of Section 21.7 drops the pair of operations its witness predicate found:

```
% deadlock_witness(+Kind, -Cycle, -Edges): Cycle
% is a list of threads each waiting, through an
% object of Kind, for the next, and the last for
% the first, the first the smallest; Edges holds,
% in the same order, the edge each thread was
% followed along: its kind and the facts that make
% it, as edge_of/4 selected them.
deadlock_witness(Kind, [T | Path],
                 [Edge | Edges]) :-
    edge_of(Kind, T, Next, Edge),
    cycle_from(Kind, T, Next, [T], Path, Edges),
    min_list([T | Path], T).

% cycle_from(Kind, Start, Current, Visited, Path,
% Edges): Path leads from Current back to Start
% along Kind edges without repeating a thread, and
% Edges are the edges followed.
cycle_from(Kind, Start, Current, Visited,
           Path, Edges) :-
    (   Current == Start
    ->  Path = [],
        Edges = []
    ;   \+ memberchk(Current, Visited),
        Path = [Current | Rest],
        Edges = [Edge | Rest1],
```

```
        edge_of(Kind, Current, Next, Edge),
        cycle_from(Kind, Start, Next,
                   [Current | Visited],
                   Rest, Rest1)
    ).

% deadlock(+Kind, -Cycle): Cycle is a cycle of
% Kind edges, from its smallest thread; the
% witness is dropped.
deadlock(Kind, Cycle) :-
    deadlock_witness(Kind, Cycle, _).

?- deadlock(critical_section, Cycle).
Cycle = [103, 104] ;
false.

?- deadlock(mutant, Cycle).
false.

?- deadlock(critical_section, C),
   forall(member(T, C), waits_for(T, _, _)).
C = [103, 104] ;
false.

?- deadlock_witness(critical_section, C, _Edges),
   pairs_keys(_Edges, Kinds),
   pairs_values(_Edges, _Lists),
   maplist(length, _Lists, Lengths).
C = [103, 104],
Kinds = [critical_section, critical_section],
Lengths = [2, 2] ;
false.
```

Threads 103 and 104 deadlock on the two critical sections, and no cycle runs through mutexes alone. The walk from 101 reaches 103 and 104 and then finds 103 again, which is in the visited list and is not the start, so the walk fails; the walk from 103 returns to 103 in two steps and succeeds; the walk from 104 finds the same cycle and `min_list/2` rejects it, since 104 is not its smallest member. Without the visited list the walk from 101 would circle 103 and 104 forever, and the test on `Current == Start` alone would not save it; the list is the termination argument, and it bounds the walk by the number of threads. The witness of the one cycle is two edges, both critical sections, each made of two facts; `pairs_keys/2` and `pairs_values/2` from `library(pairs)` take the kinds and the fact lists apart. The third query is the composition Chapter 23 builds on: every thread of a deadlock is a **Blocked Thread (Software)** (page 336) of Chapter 18.

A thread waits for one object at a time, so a thread has one edge and the walk from it has one way to go; a knowledge base that records two waits for one thread, a wait fact in `waits_for/3` and another in `waits_for_object/3`, gives that thread **parallel edges**, and the walk then has a choice. A specific kind constrains every edge of the walk, so under `critical_section` the mutex edge is never taken; under `any` the walk takes each parallel edge in turn, and reports the cycle once for each selection, with the edges it used. A caller that wants each cycle once under `any` wraps the goal in `distinct/1` of Chapter 17, as a recognizer that reaches one instance by several derivations is wrapped there. Section 20.5 checks both cases on an extension of the facts.

The second formulation is the transitive closure of Chapter 11. `reaches(Kind, A, B)` holds when a path of `Kind` edges leads from `A` to `B`, and a thread is in a deadlock when it reaches itself. Written with left recursion, the definition is the natural one and does not terminate under plain SLD resolution on a cyclic graph, because the first subgoal of the second clause is the predicate itself; tabling makes it terminate, by recording each call and its answers and completing a call that recurs into itself from the answers found so far:

```
:- table reaches/3.

% reaches(+Kind, ?A, ?B): a path of Kind edges
% leads from thread A to thread B.
reaches(Kind, A, B) :-
    edge_of(Kind, A, B).
reaches(Kind, A, B) :-
    reaches(Kind, A, C),
    edge_of(Kind, C, B).

% in_deadlock(+Kind, ?TID): TID lies on a cycle of
% Kind edges.
in_deadlock(Kind, TID) :-
    reaches(Kind, TID, TID).

?- in_deadlock(critical_section, 103).
true.

?- in_deadlock(critical_section, 101).
false.

?- aggregate_all(set(T),
   in_deadlock(critical_section, T), Ts).
Ts = [103, 104].

?- aggregate_all(set(T),
   reaches(critical_section, 101, T), Ts).
Ts = [103, 104].

?- aggregate_all(set(T), reaches(any, 122, T),
   Ts).
Ts = [103, 104, 120, 121].
```

The tabled predicate answers membership questions directly and reaches every thread downstream of a given one in one call; thread 101 reaches the two deadlocked threads and nothing else, and thread 122 reaches four threads across three kinds of object. The order in which a tabled predicate produces its answers is that of its answer table and not of the clauses, which is why the transcripts collect them into sets. The directive `table/1` must precede the clauses it applies to, and the tables are kept until `abolish_all_tables/0` or until a fact they depend on changes under incremental tabling, which these static facts do not.

The two formulations recognize the same deadlocks and differ in what they report. The walk gives the cycle as a list, which is what a report prints, and the witness of its edges, which is what the evidence form needs; the closure gives membership and reachability, which is what a composed pattern asks. Neither says which thread acquired its section first, which the dump does not record,

nor whether the two acquisitions were in a consistent order elsewhere in the program; and a thread waiting for a critical section whose owner has exited, so that the section can never be released, is as stuck as a deadlocked one and is not a cycle, since `thread/2` has no owner for the edge to reach. That configuration is the limit of a graph formulation, and Section 20.8 catches it as a corrupt or orphaned section.

The explainer form is built from the witness and from nothing else: for each edge the walk followed, the wait fact of the thread, as `waits_for/3` or `waits_for_object/3` records it, and the ownership fact of the object, `critical_section/4` or `mutant/3`, both as `wait_edge/4` handed them to the walk. An explainer that found the cycle first and then collected every edge between its adjacent threads would cite edges the walk never used, and would cite a mutex edge in the evidence of a critical section deadlock whenever the facts held one; the witness is what rules that out. The pattern's name is not fixed in the clause but selected from the set of kinds of the selected edges, since `deadlock_witness/3` with the kind `any` finds cycles of every kind. A cycle whose every edge is a critical section is a Deadlock (Critical Sections). A cycle whose selected edges have two kinds or more is a **Deadlock (Mixed Objects)** (page 394), the other deadlock name this chapter uses, and Section 20.5 recognizes it under that definition. A cycle of one kind that is not critical sections, every edge a mutex or every edge a thread object, has no variant name in the catalog lists this book uses, and is reported under the family name `'Deadlock'` with the kind in the instance term, `cycle(Kind, Cycle)`, so that a report says what kind of object the cycle runs through; under a variant name the instance is the cycle itself.

```
% cycle_kinds(Edges, Kinds): Kinds is the set of
% kinds of the edges Edges of a witness.
cycle_kinds(Edges, Kinds) :-
    pairs_keys(Edges, Kinds0),
    sort(Kinds0, Kinds).

% deadlock_name(Kinds, Name): the catalog name of
% a deadlock whose selected edges have the set of
% kinds Kinds: the critical section variant, the
% mixed variant for two kinds or more, and the
% family name for one kind of another object.
deadlock_name([critical_section],
              'Deadlock (Critical Sections)').
deadlock_name([Kind], 'Deadlock') :-
    Kind \== critical_section.
deadlock_name([_, _ | _],
              'Deadlock (Mixed Objects)').

% deadlock(+Kind, -Cycle, -Evidence): as the test
% above; the evidence holds the facts of the edges
% the walk selected, the wait fact and the
% ownership fact of each, and the pattern is named
% from the kinds of those edges. Under a variant
% name the instance is the cycle; under the family
% name it is cycle(Kind, Cycle), so that the kind
% of object is reported.
deadlock(Kind, Cycle, Evidence) :-
    deadlock_witness(Kind, Cycle, Edges),
    cycle_kinds(Edges, Kinds),
```

```
    deadlock_name(Kinds, Name),
    (   Kinds = [One],
        One \== critical_section
    ->  Instance = cycle(One, Cycle)
    ;   Instance = Cycle
    ),
    pairs_values(Edges, Lists),
    append(Lists, Facts),
    Evidence = evidence(Name, Instance, Facts).

?- deadlock(critical_section, C,
   evidence(Name, _, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
waits_for(103,critical_section,140695026508096)
critical_section(140695026508096,104,1,1)
waits_for(104,critical_section,140695026507968)
critical_section(140695026507968,103,2,1)
C = [103, 104],
Name = 'Deadlock (Critical Sections)' ;
false.

?- deadlock(any, C, evidence(Name, _, _)).
C = [103, 104],
Name = 'Deadlock (Critical Sections)' ;
C = [123, 124],
Name = 'Deadlock (Mixed Objects)' ;
false.

?- deadlock(any, [123, 124], evidence(_, _, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
waits_for_object(123,thread,124)
waits_for_object(124,mutant,436)
mutant(436,123,'AppServer.LogMutex')
true ;
false.
```

`pairs_keys/2` takes the kinds off the witness and `sort/2` reduces them to a set; `pairs_values/2` takes the fact lists, and `append/2` joins them into one, in the order of the walk. The four facts of the first cycle are the deadlock as an analyst would write it up: 103 waits for the section at `0x7ff6_1d13_a140`, which 104 owns, and 104 waits for the section at `0x7ff6_1d13_a0c0`, which 103 owns. The second cycle, which Section 20.5 treats, has three facts, since a wait for a thread object names its owner in the wait fact itself, and its name is selected by its two kinds. Every element is a fact of the knowledge base and not a derived view such as `waits/3`: a reader who checks one finds it in `dump1.pl` or in the extension, which is what the contract of Section 17.3 asks of evidence; and every element was consulted by the walk that found the cycle, which is what makes the list a witness and not a reconstruction.

The evidence is under the contract of Section 17.3, and the contract has three parts to check. The first two, that every term is ground and every term holds, are `ground/1` and `maplist(call, Facts)`. The third, that the facts are sufficient, is a re-derivation of the cycle from the list and from nothing else: `fact_edge/3` reads an edge off the list, with `member/2` in place of the knowledge base, and `cycle_entailed/2` asks that every edge of the closed cycle be read off that way.

```
% fact_edge(Facts, W, O): the list Facts alone
% says that W waits for an object that O holds.
fact_edge(Facts, W, O) :-
    member(waits_for(W, critical_section, A),
           Facts),
    member(critical_section(A, O, _, _), Facts).
fact_edge(Facts, W, O) :-
    member(waits_for_object(W, mutant, H), Facts),
    member(mutant(H, O, _), Facts).
fact_edge(Facts, W, O) :-
    member(waits_for_object(W, thread, O), Facts).

% cycle_entailed(Facts, Cycle): Facts alone entail
% that each thread of Cycle waits for an object
% the next holds, and the last for the first.
cycle_entailed(Facts, Cycle) :-
    Cycle = [First | _],
    append(Cycle, [First], Closed),
    forall(nextto(W, O, Closed),
           fact_edge(Facts, W, O)).

?- forall(deadlock(any, _, evidence(_, C, _Fs)),
          ( ground(_Fs),
            maplist(call, _Fs),
            cycle_entailed(_Fs, C) )).
true.

?- _Fs = [ waits_for(101, critical_section,
                     0x7ff6_1d13_a0c0),
           critical_section(0x7ff6_1d13_a0c0,
                            103, 2, 1),
           waits_for(103, critical_section,
                     0x7ff6_1d13_a140),
           critical_section(0x7ff6_1d13_a140,
                            104, 1, 1) ],
   maplist(call, _Fs).
true.

?- _Fs = [ waits_for(101, critical_section,
                     0x7ff6_1d13_a0c0),
           critical_section(0x7ff6_1d13_a0c0,
                            103, 2, 1),
           waits_for(103, critical_section,
                     0x7ff6_1d13_a140),
           critical_section(0x7ff6_1d13_a140,
                            104, 1, 1) ],
   cycle_entailed(_Fs, [103, 104]).
false.
```

The first query is the acceptance check for both cycles: ground, holding, and sufficient. The second and third are a boundary case. The list holds four facts of the dump, all true, two of them about thread 101, which is blocked behind the deadlock and not in it; `maplist(call, _Fs)` accepts the list, and `cycle_entailed/2` rejects it for the cycle `[103, 104]`, because no member of the list says that 104 waits for anything. True but unrelated facts are not evidence, and the re-derivation is the test that tells them from the facts that are.

20.5 Deadlock (Mixed Objects)

A **Deadlock (Mixed Objects)** is a cycle whose edges pass through objects of more than one kind: a thread holds a mutex and waits for a thread, which waits for the mutex, and no single debugger command shows the cycle, since each command knows one kind of object. The recognizer is the walk of Section 20.4 over edges of any kind, with the kinds of the edges the walk selected, taken off its witness by `cycle_kinds/2`, required to number at least two:

```
% deadlock_mixed(-Cycle, -Kinds): Cycle is a cycle
% whose selected edges have more than one kind,
% listed in Kinds.
deadlock_mixed(Cycle, Kinds) :-
    deadlock_witness(any, Cycle, Edges),
    cycle_kinds(Edges, Kinds),
    Kinds = [_, _ | _].

?- deadlock_witness(any, Cycle, _Edges),
   cycle_kinds(_Edges, Kinds).
Cycle = [103, 104],
Kinds = [critical_section] ;
Cycle = [123, 124],
Kinds = [mutant, thread] ;
false.

?- deadlock_mixed(Cycle, Kinds).
Cycle = [123, 124],
Kinds = [mutant, thread] ;
false.
```

Over all kinds the knowledge base has two cycles: the critical section deadlock of the shared dump and the extension's pair, in which thread 123 holds the log mutex and waits for thread 124 to exit while 124 waits for the log mutex. The second is mixed and the first is not, and `sort/2` on the kinds, which removes duplicates, is what makes the count a count of kinds rather than of edges. The explainer of Section 20.4 makes the same distinction when it names the pattern, from the same witness, and its evidence for the mixed cycle is the three facts shown there.

The predicate recognizes a mixed cycle among the kinds the knowledge base has edges for, which are three. Events, semaphores, and the other waitable objects have no owner in the operating system's sense, and a thread waiting on an event waits for whichever thread will set it, which the dump does not record and the analyst infers from the program; such a wait cannot be an edge, and a deadlock through an event that nobody will set is invisible to any graph built from the dump alone. The pattern's full extent is beyond the facts, and the predicate's answer is a lower bound.

20.5.1 Two Boundary Cases

Two configurations the extension does not have check that the explainer and the name rest on the witness. The first is a pair of parallel edges: thread 103, which waits for a critical section that 104 holds, is also recorded as waiting for a mutex that 104 holds. Under the kind `critical_section` the

cycle, its name, and its four facts must be those of Section 20.4, with the mutex facts nowhere in them; under `any` the same cycle is found once through each edge, and the selection through the mutex is a mixed cycle, named and witnessed as such. The facts are added with `assertz/1` and taken away again with `retractall/1`, which the dynamic declaration of the two relations permits:

```
?- assertz(waits_for_object(103, mutant, 1)),
   assertz(mutant(1, 104, a)).
true.

?- deadlock(critical_section, C,
   evidence(Name, I, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
waits_for(103,critical_section,140695026508096)
critical_section(140695026508096,104,1,1)
waits_for(104,critical_section,140695026507968)
critical_section(140695026507968,103,2,1)
C = I, I = [103, 104],
Name = 'Deadlock (Critical Sections)' ;
false.

?- deadlock(critical_section, _,
            evidence(_, _, _Fs)),
   member(F, _Fs), F \= waits_for(_, _, _),
   F \= critical_section(_, _, _, _).
false.

?- deadlock(any, [103, 104],
            evidence(Name, _, _Fs)),
   length(_Fs, N), _Fs = [_First | _],
   functor(_First, Relation, _).
Name = 'Deadlock (Critical Sections)',
N = 4,
Relation = waits_for ;
Name = 'Deadlock (Mixed Objects)',
N = 4,
Relation = waits_for_object ;
false.

?- retractall(waits_for_object(103, mutant, 1)),
   retractall(mutant(1, 104, a)).
true.
```

The second configuration is a cycle of one kind other than critical sections: two threads, 201 and 202, each holding a mutex the other waits for. The walk under `mutant` finds it; the kinds of its witness are one, and that one is not `critical_section`, so the name is the family name and the instance carries the kind; and it is not mixed, by `deadlock_mixed/2` and by the explainer alike:

```
?- assertz(waits_for_object(201, mutant, 1)),
   assertz(waits_for_object(202, mutant, 2)),
   assertz(mutant(1, 202, a)),
   assertz(mutant(2, 201, b)).
true.
```

```
?- deadlock(mutant, C, evidence(Name, I, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
waits_for_object(201,mutant,1)
mutant(1,202,a)
waits_for_object(202,mutant,2)
mutant(2,201,b)
C = [201, 202],
Name = 'Deadlock',
I = cycle(mutant, [201, 202]) ;
false.

?- deadlock_mixed(C, Kinds).
C = [123, 124],
Kinds = [mutant, thread] ;
false.

?- deadlock(any, C, evidence(Name, _, _)).
C = [103, 104],
Name = 'Deadlock (Critical Sections)' ;
C = [201, 202],
Name = 'Deadlock' ;
C = [123, 124],
Name = 'Deadlock (Mixed Objects)' ;
false.

?- retractall(waits_for_object(201, mutant, _)),
   retractall(waits_for_object(202, mutant, _)),
   retractall(mutant(1, 202, a)),
   retractall(mutant(2, 201, b)),
   aggregate_all(count, waits(_, _, _), N).
N = 8.
```

The mutex cycle is reported under `'Deadlock'` with `cycle(mutant, [201, 202])` as its instance, and the mixed recognizer still finds the one mixed cycle. The last query restores the extension to its eight waits, so that the rest of the chapter reads the facts it began with.

20.6 Blocked Queue

A **Blocked Queue** is several threads waiting for the same object: a queue has formed on one lock, and its length measures how much of the process the lock's owner is holding up. Chapter 9 grouped the waiters by object with `bagof/3`; the recognizer adds the threshold:

```
% blocked_queue(?Kind, ?Object, -Waiters): at
% least the threshold many threads wait for Object
% of Kind; Waiters lists them.
blocked_queue(Kind, Object, Waiters) :-
    bagof(W, waits(W, Kind, Object), Waiters),
    length(Waiters, N),
    threshold(queue_length, Min),
    N >= Min.
```

```
?- blocked_queue(K, A, Ws), format("~16r~n", [A]).
7ff61d13a0c0
K = critical_section,
A = 140695026507968,
Ws = [101, 104] ;
false.

?- blocked_queue(_, A, _),
   critical_section(A, O, _, _), waits(O, _, _).
A = 140695026507968,
O = 103 ;
false.
```

The one queue is on the first critical section, with two waiters, and its owner is itself waiting: the queue is behind the deadlock, and will not move. The second query is the composition that says so, and Chapter 23 orders such findings by it, since a queue behind a deadlock is explained by the deadlock and a queue behind a running owner is a different problem, contention, which the next pattern measures from the object's side.

The threshold of two is as low as a queue can be and suits a dump with ten threads; a server under load has queues of two on every lock at every moment, and the pattern there begins at a length that stands out from the rest, which the analyst sets. The queue's order, which thread will get the lock next, is not in the facts, and the operating system does not promise it either.

20.7 High Contention (Critical Sections)

High Contention (Critical Sections) is a critical section that many threads try to enter at once, which shows in the section's own data. In this schema the third argument of `critical_section/4` is `WaiterCount`, fixed in Chapter 15 as the number of threads waiting for the section. It is a value normalized at import: the parser produces it from what the debugger displayed, which is the structure's `LockCount` field or a count decoded from it, depending on the debugger and the version of its extensions, and Section 15.4 gives the decoding for the field's later encoding. The shared data is synthetic, and its waiter counts are normalized by assumption, as Chapter 15 states. The recognizer compares the count with a threshold. The measure is independent of the stack traces, which is its value: the two can be compared.

```
% high_contention(?Addr, -WaiterCount): the
% critical section at Addr reports at least the
% threshold many waiting threads.
high_contention(Addr, WaiterCount) :-
    critical_section(Addr, _, WaiterCount, _),
    threshold(contention, Min),
    WaiterCount >= Min.

?- high_contention(A, N), format("~16r~n", [A]).
7ff61d13a0c0
A = 140695026507968,
N = 2 ;
false.
```

```
?- forall(critical_section(A, _, L, _),
          ( aggregate_all(count,
                waits_for(_, critical_section, A),
                W),
            format("~16r ~d ~d~n", [A, L, W]),
            L =:= W )).
7ff61d13a0c0 2 2
7ff61d13a140 1 1
7ff61d13a1c0 0 0
true.
```

The contended section is the one with the queue, and the second query is the agreement between the two relations, printed before it is tested: for every section, the waiter count recorded for the object, 2, 1, and 0 in `dump1.pl`, equals the number of `waits_for/3` facts for that section, which the parser took from the stack traces. In a knowledge base built from a real dump the two readings come from different commands, and their agreement is a check that the import normalized the count correctly and that every waiter's stack showed the wait. In the shared data it is a check of internal consistency only: the fixture's waiter counts and its wait facts were written to agree, and the agreement does not validate the import convention, which only a real fragment and its decoding can. Section 20.8 turns a disagreement into a finding.

The pattern's intent is contention over time, many threads taking turns on a hot lock and each waiting a little, and the dump shows one instant of it; a section with a long queue at one moment may be hot or may be held once for too long, and only the owner's state distinguishes them. Here the owner is deadlocked, and the contention is a consequence rather than a cause. A count across several dumps, which Chapter 21 does for memory, would make the distinction, and a software trace of lock acquisitions, which Part IV would treat as a **Thread of Activity** (page 537) per lock, would make it precisely.

20.8 Critical Section Corruption

Critical Section Corruption is a critical section whose fields do not describe a possible state: an owner that is not a thread of the process, a waiter count with no owner, a recursion count with no owner, a count that disagrees with the threads actually waiting. Memory corruption produces such states, and so does a thread that exited while holding the section, which leaves an owner that no longer exists. The recognizer is a set of consistency rules over the fields, each naming the inconsistency it finds:

```
% critical_section_corruption(?Addr, -Reason): the
% critical section at Addr is in an impossible
% state for Reason.
critical_section_corruption(Addr, Reason) :-
    critical_section(Addr, Owner, Waiters, Rec),
    inconsistent(Addr, Owner, Waiters, Rec,
                 Reason).

% inconsistent(Addr, Owner, Waiters, Rec, Reason):
% the fields of a critical section violate the
% rule named by Reason.
```

```
inconsistent(_, 0, _, Rec, Reason) :-
    Rec > 0,
    Reason = recursion_without_owner.
inconsistent(_, 0, Waiters, _, Reason) :-
    Waiters > 0,
    Reason = waiters_without_owner.
inconsistent(_, Owner, _, 0, Reason) :-
    Owner =\= 0,
    Reason = owner_without_recursion.
inconsistent(_, Owner, _, _, Reason) :-
    Owner =\= 0,
    \+ thread(Owner, _),
    Reason = owner_not_a_thread(Owner).
inconsistent(Addr, _, Waiters, _, Reason) :-
    aggregate_all(count,
        waits(_, critical_section, Addr), N),
    N =\= Waiters,
    Reason = waiter_count_mismatch(Waiters, N).

?- critical_section_corruption(A, R).
false.

?- inconsistent(0x7ff6_1d13_a200, 0, 3, 0, R).
R = waiters_without_owner ;
R = waiter_count_mismatch(3, 0).

?- inconsistent(0x7ff6_1d13_a200, 777, 1, 1, R).
R = owner_not_a_thread(777) ;
R = waiter_count_mismatch(1, 0).
```

The three sections of the shared dump are consistent, and the two constructed tuples show the rules firing: a section with waiters and no owner violates two rules, and a section owned by a thread the dump does not have, the orphaned section of Section 20.4, violates one of them and the count rule as well, since nobody in the dump waits for the invented address. Several rules can fire for one section and each is reported, which is the right behavior for a diagnostic check.

The rules encode what this schema's fields mean. `WaiterCount` is the normalized number of waiting threads, as Chapter 15 defines it, so `waiters_without_owner` says that nobody waits for a section nobody holds, and `waiter_count_mismatch` compares the count with the `waits_for/3` facts, two readings of the dump that the debugger produced by different commands; Section 20.7 checked that the fixture's values agree, which is a consistency check within the knowledge base. The `LockCount` field of the operating system's structure, whose encoding depends on the Windows version, is not in the schema; Section 15.4 gives the decoding for one of its encodings, and a rule about the field itself would have to be written against a parser that stored it. A mismatch found by the last rule therefore has three readings, a corrupt section, a parser that recorded the field undecoded or decoded it under the wrong encoding, and a waiter whose stack trace does not show the wait, and the coverage records of Chapter 15 are where the analyst looks first. The rules also catch only inconsistencies among recorded facts: a corrupt section whose fields happen to form a possible state passes them, and a debug-build section with its debug information block, which the debugger checks more deeply, is not in the schema.

20.9 Swarm of Shared Locks

A **Swarm of Shared Locks** is a large number of threads waiting on reader-writer locks held in shared mode, with stack traces alike down to the acquisition and no owner to follow, since a lock held by many readers has no single owner the debugger reports; the swarm itself is the finding, and the writer that is waiting for all the readers to leave, or the reader that never leaves, is found by other means. The schema has no facts for such locks, and the following extension records the threads waiting on each; the pattern is treated in prose with the one predicate the facts allow, a count:

```
% shared_lock_wait(TID, Addr): TID waits on the
% reader-writer lock at Addr. An extension of the
% schema.
shared_lock_wait(125, 0x7ff6_1d13_b000).
shared_lock_wait(126, 0x7ff6_1d13_b000).
shared_lock_wait(127, 0x7ff6_1d13_b000).
shared_lock_wait(128, 0x7ff6_1d13_b000).
shared_lock_wait(129, 0x7ff6_1d13_b000).
shared_lock_wait(130, 0x7ff6_1d13_b040).

% swarm_of_shared_locks(?Addr, -TIDs): at least
% the threshold many threads wait on the lock at
% Addr.
swarm_of_shared_locks(Addr, TIDs) :-
    bagof(T, shared_lock_wait(T, Addr), TIDs),
    length(TIDs, N),
    threshold(swarm_size, Min),
    N >= Min.

?- swarm_of_shared_locks(A, Ts),
   format("~16r~n", [A]).
7ff61d13b000
A = 140695026511872,
Ts = [125, 126, 127, 128, 129] ;
false.
```

Five threads wait on one lock and one on another, and the threshold separates the swarm from the single waiter. The count is all the facts support, and the difference from **Blocked Queue** (page 396) is in what cannot be done next: a queue has an owner and a wait chain, and a swarm has neither, so that the wait-for graph has no edges here and the deadlock predicates have nothing to find even when the lock will never be released. The analyst looks instead for the thread that holds the lock exclusively, whose stack trace shows the acquisition in exclusive mode, or for a reader that acquired the lock and is blocked elsewhere; both are stack trace searches of the kind Chapter 18 wrote, over frames the schema would record if the dump had them.

20.10 Dispatch Level Spin

Dispatch Level Spin is a kernel pattern: a processor stuck at dispatch level, the **interrupt request level** at which the kernel's scheduler runs and thread switching is suspended, spinning on a spin lock or in a loop, so that the processor does nothing else and a system with one processor hangs. It is

recognized in a kernel or complete memory dump from the processor's current interrupt request level and the stack of the thread it was running, and `!running` and the per-processor data are where the debugger shows it.

The pattern is treated in prose because the schema is a user-mode schema. A process memory dump has no processors, no interrupt request levels, and no spin locks that the process can see; the kernel-mode schema extension that Chapter 15 discussed, with a process dimension and per-processor facts, is where a `dispatch_level_spin/1` predicate would be written, as a test on a processor's recorded level and the top frames of its current thread. What a user-mode dump of the same system would show is nothing at all, since a system spinning at dispatch level does not write user-mode dumps; the pattern is one of the reasons a kernel dump is taken.

20.11 Waiting Thread Time

Waiting Thread Time is how long each waiting thread has been waiting: the threads that have waited longest are the oldest part of a hang, and two threads whose wait times are nearly equal entered their waits together, which for a deadlock dates it. The time is available in kernel dumps from the thread's wait time field; a process memory dump does not record it, and the extension `wait_time/2` records a wait time in milliseconds per waiting thread, as a parser of a kernel dump of the same system would have supplied it.

```
% wait_time(TID, Ms): TID has been waiting for Ms
% milliseconds. An extension of the schema.
wait_time(100, 20).
wait_time(101, 9800).
wait_time(103, 12400).
wait_time(104, 12380).
wait_time(105, 61000).
wait_time(106, 59500).
wait_time(108, 4900).

% waiting_thread_time(-TID, -Ms): the waiting
% threads with their wait times, longest first.
waiting_thread_time(TID, Ms) :-
    order_by([desc(Ms)], wait_time(TID, Ms)).

?- forall(waiting_thread_time(T, Ms),
          format("~d~t~6|~t~D~8+ ms~n", [T, Ms])).
105     61,000 ms
106     59,500 ms
103     12,400 ms
104     12,380 ms
101      9,800 ms
108      4,900 ms
100         20 ms
true.

?- waiting_thread_time(T, _), waits(T, _, _).
T = 103 ;
T = 104 ;
T = 101 ;
```

```
false.
```

The longest waits belong to the two idle pool threads, which have waited a minute for work and are the least interesting threads in the dump; the pattern is read together with the thread's class, and the threads that have waited long while blocked on an object are the second query, in order. Threads 103 and 104 have waited within 20 ms of each other, which is the deadlock's time of formation, and thread 101 joined the queue two and a half seconds later; the main thread's wait of 20 ms is a message loop doing its job.

`order_by/2` of Chapter 9 sorts the solutions, and the second query keeps the order because `waits/3` is a test on each. What the facts do not say is what the times are relative to, the moment of the dump or the moment the debugger read them; and a thread that woke and waited again has a short wait time and may have been blocked, on and off, for as long as any. The time measures the current wait, not the thread's history.

20.12 Thread Starvation (Normal Priority)

Thread Starvation (Normal Priority) is a thread at normal priority that cannot get a processor because threads at higher priority consume them: it is ready to run and never scheduled, and from the inside of the process it looks hung although nothing blocks it. The symptom in a dump is a spiking thread at raised priority next to normal threads with no CPU time, and the schema lacks priorities; the extension `priority/2` records them.

```
% priority(TID, Level): the scheduling priority of
% TID. An extension of the schema.
priority(102, above_normal).
priority(107, above_normal).
priority(100, normal).
priority(101, normal).
priority(103, normal).
priority(104, normal).
priority(105, normal).
priority(106, normal).
priority(108, normal).
priority(109, normal).

% higher_priority(P, Q): level P is above level Q.
higher_priority(above_normal, normal).
higher_priority(high, normal).
higher_priority(high, above_normal).

% cpu_share(TID, Share): the CPU time of TID as a
% percentage of the total, as in Chapter 18.
cpu_share(TID, Share) :-
    thread_time(TID, U, K),
    Ms is U + K,
    aggregate_all(sum(U1 + K1),
                  thread_time(_, U1, K1), Total),
    Total > 0,
    Share is 100.0 * Ms / Total.
```

```
% spiking_thread(TID): as in Chapter 18.
spiking_thread(TID) :-
    cpu_share(TID, Share),
    threshold(spiking_share, Min),
    Share >= Min.

% thread_starvation(-TID, -By): TID is a normal
% priority thread not blocked on a recorded object
% and not a pool thread, while By spikes at a
% higher priority.
thread_starvation(TID, By) :-
    priority(TID, normal),
    thread_state(TID, waiting(user_request)),
    \+ waits(TID, _, _),
    \+ frame(TID, _, ntdll, 'TppWorkerThread', _),
    spiking_thread(By),
    priority(By, P),
    higher_priority(P, normal).

?- thread_starvation(T, By).
T = 100,
By = 102 ;
T = 100,
By = 107 ;
false.
```

The main thread is the candidate: at normal priority, waiting on nothing the dump records, not a pool thread, with 160 ms of CPU time against the two above-normal threads' 13.8 seconds. The two answers name the two threads that would starve it.

The predicate is a heuristic twice over and should be read as one. A user-mode dump does not record that a thread is ready rather than waiting, so the test for "not blocked" is the negation of what the schema does know, and the main thread here is in a wait on objects the schema does not list; it may be starved or merely idle, and the facts cannot tell. The pattern proper is identified in a kernel dump, where the thread's state and the ready queues are visible, and the user-mode recognizer finds the configuration in which starvation is possible. The priority levels are the operating system's named classes, and the extension would need the numeric priorities, with their dynamic boosts, to compare threads within one class.

20.13 Blocking Module

The **Blocking Module** is the module whose code holds a lock that others wait for: when a chain of waits ends at an owner, the analyst asks which component the owner is executing in while it holds the lock, since that component is where the hold is too long or the ordering is wrong. The recognizer takes the owner of a critical section and the topmost frame of its stack trace outside the `system_module/1` list:

```
% blocking_module(?Addr, -Owner, -Module): Owner
% holds the critical section at Addr and is
% executing in Module, its topmost frame outside
```

```
% the system modules.
blocking_module(Addr, Owner, Module) :-
    critical_section(Addr, Owner, _, _),
    Owner =\= 0,
    aggregate_all(min(N),
                  ( frame(Owner, N, M, _, _),
                    \+ system_module(M)
                  ),
                  Top),
    frame(Owner, Top, Module, _, _).

?- blocking_module(A, O, M),
   format("~16r~n", [A]).
7ff61d13a0c0
A = 140695026507968,
O = 103,
M = dbclient ;
7ff61d13a140
A = 140695026508096,
O = 104,
M = appserver ;
false.

?- blocking_module(_, O, M), frame(O, N, M, F, _),
   \+ ( frame(O, N1, M, _, _), N1 < N ).
O = 103,
M = dbclient,
N = 4,
F = 'Connection::Send' ;
O = 104,
M = appserver,
N = 4,
F = 'Session::Lookup' ;
false.
```

The first section is held by a thread executing in the database client, inside `Connection::Send`, and the second by a thread in the application's `Session::Lookup`; the second query names the functions. Read with the deadlock, the shape is two components taking the same two locks in opposite orders: thread 103 holds the first section and asks for the second from the database client's send routine, while thread 104 holds the second and asks for the first from the application's lookup. Which data each section guards, the frames suggest and the facts do not say. The negation in the second query selects the frame with the smallest number in the module, which is what `aggregate_all(min(N), ...)` did inside the predicate; it is written out to show the frame.

The module in which the owner is executing is not necessarily the module that acquired the lock, which may be further down the owner's stack and which the facts do not mark; `RtlEnterCriticalSection` appears on a waiter's stack and not on an owner's, since the owner's acquisition returned. The predicate names the component holding the lock at the moment of the dump, and the frames below the named one are where the acquisition is to be looked for.

Chapter 21. Memory, Heaps, Handles, and Leaks

The patterns of Chapters 18 to 20 concern what the threads of a process were doing. This chapter turns to what the process held: its heaps, its handles, the words on its stacks, and the values its functions were given. A process memory dump records these as numbers, and the debugger derives summaries from them, the heap summary of `!heap -s`, the handle counts of `!handle`, the parameters it reconstructs for each frame. The schema of Chapter 15 keeps the summaries as `heap/4`, `handle_count/3`, and `exception_info/3`, and the recognizers of this chapter begin from those facts. Where a count or a sum is needed, `aggregate_all/3` of Chapter 9 provides it; where a series is needed, the pairs and `keysort/2` of the same chapter.

Several patterns of the chapter need data that `dump1.pl` does not hold. A leak is growth, which one dump cannot show; heap corruption is found by a verification pass the schema does not record; a double free is an event in a history of heap operations; a value is referenced by the words of a raw stack. For each, the chapter defines a small **extension of the schema** as facts under a predicate of its own, `heap_at/5`, `heap_check/3`, `heap_op/4`, `raw_stack/3`, `frame_param/4`, says so where the facts are introduced, and chooses invented values that are consistent with the shared dump. A reader who has such data from a debugger replaces the facts; the recognizers do not change.

Three conventions of Chapter 17 are used throughout. A recognizer is named by the snake_case of its pattern. Where the catalog name carries a variant in parentheses that the schema cannot distinguish from any other, as every heap in a process memory dump is a process heap, the variant is dropped from the predicate name; where a chapter treats two variants of one name, the variant is the first argument. Thresholds are facts of `threshold/2`, never numbers in clause bodies. The explainer form of a recognizer has one more argument than the enumerator form and binds it to a term `evidence(Pattern, Instance, Facts)` whose last argument lists the facts the recognition rests on.

```
:- [dump1].

% pattern(Name, Artifact, Chapter): the catalog
% entries treated in this chapter.
pattern('Insufficient Memory (Committed Memory)',
    memory_dump, 21).
pattern('Handle Leak', memory_dump, 21).
pattern('Insufficient Memory (Handle Leak)',
    memory_dump, 21).
pattern('Memory Leak (Process Heap)',
    memory_dump, 21).
pattern('Memory Fluctuation', memory_dump, 21).
pattern(
    'Dynamic Memory Corruption (Process Heap)',
    memory_dump, 21).
pattern('Double Free (Process Heap)',
    memory_dump, 21).
pattern('Invalid Handle (General)',
    memory_dump, 21).
pattern('Paged Out Data', memory_dump, 21).
pattern('Shared Buffer Overwrite',
```

```
    memory_dump, 21).
pattern('Value References', memory_dump, 21).
pattern('Regular Data', memory_dump, 21).
pattern('Data Correlation (Function Parameters)',
    memory_dump, 21).

% threshold(Name, Value): the analyst's limits.
threshold(committed_ratio, 0.85).
threshold(large_heap_kb, 1048576).
threshold(committed_total_kb, 1572864).
threshold(handle_type_count, 5000).
threshold(handle_total_count, 10000).
threshold(leak_growth_kb, 65536).

% print_evidence(Evidence): the pattern name, the
% instance, and one line per supporting fact.
print_evidence(evidence(Name, Instance, Facts)) :-
    format("~a~n  instance: ~q~n",
           [Name, Instance]),
    forall(member(Fact, Facts),
           print_fact(Fact)).

% print_fact(Fact): Fact on one indented line, as
% the toplevel would print it.
print_fact(Fact) :-
    Options = [ quoted(true),
                spacing(next_argument)
              ],
    format("  ~W~n", [Fact, Options]).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, pattern(_, _, 21), N).
N = 13.

?- threshold(committed_ratio, R).
R = 0.85.
```

The flag hides the bindings of variables whose names begin with an underscore, as in Chapter 9; several queries below bind an address or an evidence term only to print it in a readable form, since the toplevel prints integers in decimal and prints a long term on one line however long it is. `print_evidence/1` prints the facts one per line with the `~W` directive of `format/2`, which takes a term and a list of `write_term/2` options; `quoted(true)` gives atoms the quotes they need to be read back, and `spacing(next_argument)` puts a space after each comma, as the toplevel does.

21.1 Insufficient Memory (Committed Memory)

The pattern names a process that has committed nearly all the memory it can commit: its heaps are full to their reservations, or the sum of its commitments approaches the limit of its address space or of the system. The symptom in the application is an allocation failure, usually reported far from the code that consumed the memory, and the debugger shows the condition in the heap summary and in `!address -summary`.

The schema holds one summary line per heap, `heap(Addr, ReservedKb, CommittedKb, FreeKb).` Two readings of the pattern follow. The first is per heap: the committed part is a large fraction of the reservation, and the reservation is itself large, since a small heap committed to its limit is the normal state of a small heap. The second is per process: the sum of committed memory over all heaps exceeds what the analyst expects of this application. Both fractions and sums are the analyst's limits, and both are `threshold/2` facts. The second variant of the pattern treated in this chapter, **Insufficient Memory (Handle Leak)** (page 410), is the third clause of the same predicate, because the clauses of a predicate must be defined together and SWI-Prolog treats a predicate defined again by a later file as redefined, discarding the earlier clauses with a warning. The third clause is read in Section 21.3.

```
% committed_ratio(Heap, Ratio): Ratio is the
% committed part of Heap over its reservation.
committed_ratio(Heap, Ratio) :-
    heap(Heap, Reserved, Committed, _),
    Reserved > 0,
    Ratio is Committed / Reserved.

% insufficient_memory(Variant, Instance): the two
% variants of the pattern treated in this chapter.
insufficient_memory(committed_memory, heap(H)) :-
    committed_ratio(H, Ratio),
    threshold(committed_ratio, MinRatio),
    Ratio >= MinRatio,
    heap(H, Reserved, _, _),
    threshold(large_heap_kb, MinKb),
    Reserved >= MinKb.
insufficient_memory(committed_memory, process) :-
    aggregate_all(sum(C), heap(_, _, C, _),
                  Total),
    threshold(committed_total_kb, MaxKb),
    Total >= MaxKb.
insufficient_memory(handle_leak, process(PID)) :-
    process(PID, _),
    aggregate_all(sum(C), handle_count(PID, _, C),
                  Total),
    threshold(handle_total_count, Max),
    Total > Max.

?- forall(committed_ratio(H, R),
          format("~16r ~3f~n", [H, R])).
210a0000000 0.875
210a0010000 0.250
210a2000000 0.938
true.

?- insufficient_memory(committed_memory, I).
I = heap(2270427086848) ;
I = process.

?- aggregate_all(sum(C), heap(_, _, C, _), T).
T = 1896704.
```

Two of the three heaps are committed beyond 85 percent of their reservations, and the first query shows why the ratio alone is not the pattern: the third heap is 94 percent committed, but it reserves 64 MB, below the floor of 1 GB in `threshold(large_heap_kb, _)`, and is not recognized. The first heap is recognized by both the ratio and the floor, and the process as a whole is recognized because its 1.8 GB of committed heap memory exceeds the 1.5 GB limit. The `/` of `committed_ratio/2` divides two integers; SWI-Prolog yields an integer when the division is exact and a float otherwise, which is SWI-Prolog's behavior when the flags `iso` and `prefer_rationals` are both `false`, their defaults; in ISO Prolog the result of `/` on two integers is implementation defined, and a program that needs a float divides floats.

The explainer form attaches the facts. The variant's catalog name is given by a fact, `variant_name/2`, so that the explainer and the report of Chapter 23 spell it the same way:

```
% variant_name(Variant, Name): the catalog name of
% a variant of Insufficient Memory.
variant_name(committed_memory,
    'Insufficient Memory (Committed Memory)').
variant_name(handle_leak,
    'Insufficient Memory (Handle Leak)').

% insufficient_memory(Variant, Instance, Ev): the
% explainer form for heap instances: the heap and
% the two thresholds the first clause compared it
% with.
insufficient_memory(committed_memory, I, E) :-
    I = heap(H),
    insufficient_memory(committed_memory, I),
    heap(H, R, C, F),
    threshold(committed_ratio, Min),
    threshold(large_heap_kb, MinKb),
    variant_name(committed_memory, Name),
    Facts = [ heap(H, R, C, F),
              threshold(committed_ratio, Min),
              threshold(large_heap_kb, MinKb)
            ],
    E = evidence(Name, I, Facts).

?- forall(insufficient_memory(committed_memory,
                              _, E),
          print_evidence(E)).
Insufficient Memory (Committed Memory)
  instance: heap(2270427086848)
  heap(2270427086848, 2097152, 1835008, 24576)
  threshold(committed_ratio, 0.85)
  threshold(large_heap_kb, 1048576)
true.
```

The explainer's first subgoal is the enumerator form of the same predicate; the explainer adds facts to a recognition rather than recognizing again, which keeps the two forms in agreement by construction. The `I = heap(H)` before it restricts the clause to heap instances, since the `process` instance has no single heap to cite. The evidence lists both thresholds, because the first clause compared the heap with both: the ratio and the reservation are computed from the `heap/4` fact,

and a reader with the fact and the two limits can repeat the two comparisons, which is the sufficiency the contract of Section 17.3 asks for; with the ratio limit alone the list would be true and incomplete, since nothing in it says what a large heap is.

The predicate does not capture what the dump does not contain. A process memory dump holds the heaps' own accounting, not the system's commit limit nor the private memory the process committed outside its heaps, so the per-process clause is a statement about heap memory, not about the commit charge the system saw. Nor does a full heap say whether it is full of live data or of garbage: the `FreeKb` column, 24 MB free inside 1.8 GB committed, says only that the heap manager has little to hand out without committing more. Whether the data is needed is a question about the program, and Section 21.4 asks the dumps it can be asked of.

21.2 Handle Leak

The pattern names a process holding far more handles of some type than its function requires, which an analyst reads from the per-type counts of `!handle` and attributes to handles that were opened and never closed. The schema has `handle_count(PID, Type, Count)`, and the recognizer compares the count of each type with a limit:

```
% handle_leak(Type): the count of handles of Type
% exceeds the analyst's limit for one type.
handle_leak(Type) :-
    handle_count(_, Type, Count),
    threshold(handle_type_count, Max),
    Count > Max.

% handle_share(Type, Share): the share of the
% process's handles that are of Type.
handle_share(Type, Share) :-
    handle_count(PID, Type, Count),
    aggregate_all(sum(C), handle_count(PID, _, C),
                  Total),
    Share is Count / Total.

?- handle_leak(Type).
Type = event ;
false.

?- handle_share(event, S), P is round(100 * S).
S = 0.9940554821664465,
P = 99.
```

```
?- forall(order_by([desc(C)],
                   handle_count(_, T, C)),
          format("~a~t~10|~t~D~8+~n", [T, C])).
event       12,040
file            38
key             14
thread          10
section          6
mutant           4
true.
```

The event handles are 99 percent of all handles of the process, and the trailing `false.` of the first query is `handle_count/3` trying its remaining five facts, none of which exceeds the limit. The table is the ordered view of `order_by/2` from Chapter 9, and it says what the count alone does not: the other types are in the dozens, which is the normal state of a server with ten threads, and the one type is three orders of magnitude out of line with the rest.

The formalization stops at the count. A leak is a process over time, and one dump holds one moment of it; twelve thousand event handles could be a leak or a design that keeps one event per connection in a server with twelve thousand connections. The dump does not say which. The remedy is the same as for memory, a series of dumps and a fact with a dump identifier, `handle_count_at/4`, and the predicates of Section 21.4 transfer to it unchanged. The schema also drops what `!handle` shows per handle, the object name and the granted access, which is often what identifies the code that opened them.

21.3 Insufficient Memory (Handle Leak)

The pattern is the memory consequence of the previous one: a handle leak severe enough that the resources handles consume, the process's handle table and the kernel memory behind each object, become the limiting factor. It is the third clause of `insufficient_memory/2` in Section 21.1, which compares the total count over all types with `threshold(handle_total_count, _)` rather than any one type with its own limit.

```
?- insufficient_memory(handle_leak, I).
I = process(4120).

?- insufficient_memory(V, I).
V = committed_memory,
I = heap(2270427086848) ;
V = committed_memory,
I = process ;
V = handle_leak,
I = process(4120).

?- insufficient_memory(handle_leak, process(P)),
   handle_leak(T).
P = 4120,
T = event ;
false.
```

With the variant unbound, the predicate enumerates its three instances, which is the shape a report wants; with both arguments bound it is a test. The last query composes the two patterns: the process whose handle total is excessive and the type that accounts for it, which together say more than either alone. The `.` after the single answer of the first query is the toplevel reporting that `process/2` has one fact and `threshold/2` with its first argument bound is selected by first-argument indexing, so no choice point remains.

The limits of Section 21.2 apply, and one more. The threshold is the analyst's judgment of what is plausible for this application, not an operating system limit: a process is bounded in the handles it may hold, and each costs kernel memory, but a dump of the process does not contain the system-wide figures that say how close the limit was. The pattern as recognized here is therefore a statement about the process's own count. A kernel memory dump, which records the system's pool usage and the handle table of every process, would let the consequence be checked rather than inferred; Section 22.18 touches on such dumps.

21.4 Memory Leak (Process Heap)

The pattern names heap memory that grows for the life of the process because what is allocated is not freed. One dump shows a size; a leak is a rate, and the smallest artifact that shows it is two dumps of the same process taken some time apart. The following facts extend the schema with a series of three snapshots of the heaps of `dump1.pl`'s process: `dump1a` and `dump1b` are two dumps an operator took while the service was slow, and `dump1` is the dump of this book, taken last. `heap_at/5` is `heap/4` with a dump identifier in front, and the clause for `dump1` is a rule that reads the shared facts rather than a copy of them. `dump_time/2` gives each dump's time in seconds from the first.

```
% heap_at(Dump, Heap, ReservedKb, CommittedKb,
% FreeKb): heap/4 of a dump in a series. An
% extension of the schema.
heap_at(dump1a, 0x0000_0210_a000_0000,
        2097152, 1703936, 40960).
heap_at(dump1a, 0x0000_0210_a001_0000,
        1024, 256, 128).
heap_at(dump1a, 0x0000_0210_a200_0000,
        65536, 59392, 2048).
heap_at(dump1b, 0x0000_0210_a000_0000,
        2097152, 1769472, 32768).
heap_at(dump1b, 0x0000_0210_a001_0000,
        1024, 256, 128).
heap_at(dump1b, 0x0000_0210_a200_0000,
        65536, 32768, 4096).
heap_at(dump1, Heap, R, C, F) :-
    heap(Heap, R, C, F).

% dump_time(Dump, Seconds): when each dump of the
% series was written, from the first.
dump_time(dump1a, 0).
dump_time(dump1b, 600).
dump_time(dump1, 1200).
```

```
% committed_series(Heap, Series): Series is the
% list of Time-CommittedKb pairs of Heap in time
% order.
committed_series(Heap, Series) :-
    findall(T-C,
            ( heap_at(D, Heap, _, C, _),
              dump_time(D, T)
            ),
            Pairs),
    keysort(Pairs, Series).

?- committed_series(0x0000_0210_a000_0000, S).
S = [0-1703936, 600-1769472, 1200-1835008].

?- committed_series(0x0000_0210_a200_0000, S).
S = [0-59392, 600-32768, 1200-61440].
```

`keysort/2` orders the pairs by time, so the series does not depend on the order of the facts; the dump identifiers are gone from it, which is right, since the pattern is about the numbers in time order and not about which file they came from. The recognizer asks whether the series rises at every step and whether the total rise is large enough to matter:

```
% strictly_increasing(Series): every value is
% greater than the one before it.
strictly_increasing([_-C0 | Pairs]) :-
    increasing_from(Pairs, C0).

% increasing_from(Pairs, Prev): the values of
% Pairs increase, starting above Prev.
increasing_from([], _).
increasing_from([_-C | Pairs], Prev) :-
    Prev < C,
    increasing_from(Pairs, C).

% memory_leak(Heap): the committed size of Heap
% grows at every snapshot, by at least the leak
% threshold over the series.
memory_leak(Heap) :-
    heap(Heap, _, _, _),
    committed_series(Heap, Series),
    Series = [_-First, _ | _],
    strictly_increasing(Series),
    last(Series, _-Last),
    threshold(leak_growth_kb, MinGrowth),
    Last - First >= MinGrowth.

% leak_rate(Heap, KbPerMinute): the mean growth of
% Heap over the series.
leak_rate(Heap, KbPerMinute) :-
    committed_series(Heap, Series),
    Series = [T0-C0 | _],
    last(Series, T1-C1),
    T1 > T0,
    KbPerMinute is (C1 - C0) / (T1 - T0) * 60.
```

```
?- memory_leak(H), format("~16r~n", [H]).
210a0000000
H = 2270427086848 ;
false.

?- leak_rate(0x0000_0210_a000_0000, Rate).
Rate = 6553.6.
```

The first heap grew by 64 MB in each ten-minute interval, 6.4 MB a minute, and is recognized; the second heap is flat and the third falls and rises, so `strictly_increasing/1` fails for both, and the `false.` after the answer is `heap/4` trying them. The pattern `Series = [_-First, _ | _]` in `memory_leak/1` both takes the first value and requires at least two snapshots, since a series of one cannot grow. `last/2` takes the final pair. `increasing_from/2` carries the previous value as an argument and distinguishes its two clauses by `[]` against a nonempty list, which first-argument indexing tells apart, so the recursion leaves no choice point; a pair of clauses for `[_]` and `[_, _ | _]` would say the same thing and leave one, since both heads have the list functor `'[|]'/2` and the indexing looks no deeper.

Three points are not a trend, and the predicate does not pretend otherwise: it recognizes monotone growth over the snapshots it has. Monotone growth is consistent with a leak and equally consistent with a cache filling to its designed size, a startup phase, or a workload that grew; the series must span the steady state of the process for the distinction to be visible, and a fourth and fifth snapshot that flatten the curve would withdraw the recognition. The predicate also says which heap grows and not which code grows it; that needs the allocation stack traces of a heap tracing facility, which the schema does not hold, and Section 21.7 shows the shape such a record takes.

21.5 Memory Fluctuation

The pattern names memory use that rises and falls between observations, which is what a leak does not do. Its diagnostic value is negative and positive at once: a reading that is high in one dump and low in the next is not a leak, and a heap that breathes with the workload, filling during a report build and emptying after it, tells the analyst that the application's memory behavior is periodic. The recognizer classifies each step of a series as a rise, a fall, or no change, and asks for at least one rise and one fall:

```
% directions(Series, Dirs): Dirs classifies each
% step of Series as up, down, or flat.
directions([_-C0 | Pairs], Dirs) :-
    directions_from(Pairs, C0, Dirs).

directions_from([], _, []).
directions_from([_-C | Pairs], Prev, [D | Ds]) :-
    compare(Order, C, Prev),
    direction(Order, D),
    directions_from(Pairs, C, Ds).
```

```
direction(<, down).
direction(=, flat).
direction(>, up).

% memory_fluctuation(Heap): the committed size of
% Heap both rose and fell over the series.
memory_fluctuation(Heap) :-
    heap(Heap, _, _, _),
    committed_series(Heap, Series),
    directions(Series, Dirs),
    memberchk(up, Dirs),
    memberchk(down, Dirs).

?- committed_series(0x0000_0210_a200_0000, S),
   directions(S, D).
S = [0-59392, 600-32768, 1200-61440],
D = [down, up].

?- committed_series(0x0000_0210_a001_0000, S),
   directions(S, D).
S = [0-256, 600-256, 1200-256],
D = [flat, flat].

?- memory_fluctuation(H), format("~16r~n", [H]).
210a2000000
H = 2270460641280.
```

`compare/3` yields the order of the two values as one of the atoms `<`, `=`, and `>`, and `direction/2` maps them to words; the two-step series of the third heap is a fall followed by a rise, which is the smallest fluctuation three snapshots can show. `memberchk/2` is `member/2` that stops at the first match, which is the right choice when the question is whether an element occurs and not where.

The limit of three snapshots is sharper here than in the previous section. Three points show at most one reversal, so the predicate cannot tell a fluctuation from a single dip, and it cannot see a period; more snapshots give both. A subtler failure is a sawtooth whose troughs rise, which is a fluctuation on top of a leak. `memory_leak/1` misses it, because the series is not monotone, and `memory_fluctuation/1` recognizes only the fluctuation. A series long enough to hold several troughs would let a refinement compare the local minima, and that refinement would be a leak recognizer that is robust to periodic use; it is not written here, because the data for it is not.

21.6 Dynamic Memory Corruption (Process Heap)

The pattern names damage to the heap’s own bookkeeping, the block headers and free lists that the heap manager keeps between the blocks it hands out, by an application that wrote past the end of a block or into a block it had freed. The heap manager finds the damage later, when it walks the damaged structures to serve an unrelated request, so the thread that crashes in heap code is rarely the one that did the writing. The debugger’s heap verification reports, for each block it examines, that the block passed or what was wrong with it; the schema does not record this, and `heap_check/3` extends it with one fact per examined block.

```
% heap_check(Heap, Block, Finding): the result of
% the debugger's heap verification for Block of
% Heap: ok, or the kind of damage found. An
% extension of the schema.
heap_check(0x0000_0210_a200_0000,
           0x0000_0210_a204_1000, ok).
heap_check(0x0000_0210_a200_0000,
           0x0000_0210_a204_1040, bad_size).
heap_check(0x0000_0210_a200_0000,
           0x0000_0210_a204_1080, bad_prev_size).
heap_check(0x0000_0210_a200_0000,
           0x0000_0210_a204_10c0, ok).
heap_check(0x0000_0210_a001_0000,
           0x0000_0210_a001_1000, ok).

% corrupt_entry(Heap, Block, Finding): a block of
% Heap that the verification did not pass.
corrupt_entry(Heap, Block, Finding) :-
    heap_check(Heap, Block, Finding),
    Finding \== ok.

% verified(Heap): the verification examined at
% least one block of Heap.
verified(Heap) :-
    heap(Heap, _, _, _),
    once(heap_check(Heap, _, _)).

% dynamic_memory_corruption(Heap): at least one
% block of Heap did not pass the verification.
dynamic_memory_corruption(Heap) :-
    heap(Heap, _, _, _),
    aggregate_all(count,
                  corrupt_entry(Heap, _, _), N),
    N > 0.


?- dynamic_memory_corruption(H),
   format("~16r~n", [H]).
210a2000000
H = 2270460641280.

?- forall(corrupt_entry(_, B, F),
          format("~16r ~a~n", [B, F])).
210a2041040 bad_size
210a2041080 bad_prev_size
true.

?- heap(H, _, _, _), \+ verified(H),
   format("~16r~n", [H]).
210a0000000
H = 2270427086848 ;
false.
```

The third heap, the one that fluctuates, has two damaged blocks, and it is the last `heap/4` fact, which is why this answer and the one of Section 21.5 end in `.` where the enumeration of Section 21.4 ended in `false.`: no fact remains to try. The findings are adjacent: the block at `0x...1040` has a size field the verification rejected, and the next block's header records a previous size that does not match it. The two findings are consistent with a single overwrite across the boundary between the two blocks,

which is what an overrun of the first block would produce, and consistent with other histories too; the verification reports the state of the headers, not how they came to be in it.

The third query is the closed-world caution of Chapter 17 made concrete. The first heap has no `heap_check/3` facts, which means the verification did not examine it, not that it is sound, and `dynamic_memory_corruption/1` fails for it for lack of evidence rather than because of evidence. `verified/1` exists so that a report can say which heaps were examined; the report of Chapter 23 says it. The `once/1` in it makes the test deterministic, since a heap with many examined blocks would otherwise leave a choice point for each.

21.7 Double Free (Process Heap)

The pattern names a block freed twice with no allocation of it between the two frees. The second free hands the heap manager a header it has already processed, and the manager either detects the state and raises an exception or breaks into the debugger, or does not and corrupts its free lists, which is one way the previous pattern arises. A dump shows the second free, in the stack trace of the thread that performed it; the first free is in the past and is visible only if a heap tracing facility logged it. `heap_op/4` extends the schema with such a log, one fact per recorded operation in sequence. The log below has eight operations: the block at `0x...1040` is allocated, freed twice, then allocated again and freed once more, so that the history holds both a double free and, after it, the normal life of a block.

```
% heap_op(Seq, Op, Block, TID): the Seq-th
% recorded heap operation, alloc or free of Block
% by thread TID. An extension of the schema: the
% log a heap tracing facility can provide.
heap_op(1, alloc, 0x0000_0210_a204_1040, 102).
heap_op(2, alloc, 0x0000_0210_a204_1080, 102).
heap_op(3, free, 0x0000_0210_a204_1040, 103).
heap_op(4, alloc, 0x0000_0210_a204_10c0, 102).
heap_op(5, free, 0x0000_0210_a204_1040, 104).
heap_op(6, free, 0x0000_0210_a204_1080, 102).
heap_op(7, alloc, 0x0000_0210_a204_1040, 102).
heap_op(8, free, 0x0000_0210_a204_1040, 102).

% log_complete(Relation): the analyst's assumption
% that the extension relation records every
% operation of its kind, with no gaps.
log_complete(heap_op/4).

% no_alloc_between(Block, N1, N2): the log records
% no allocation of Block after operation N1 and
% before operation N2.
no_alloc_between(Block, N1, N2) :-
    \+ ( heap_op(N3, alloc, Block, _),
         N1 < N3,
         N3 < N2
       ).
```

```
% double_free_witness(Block, N1, N2): Block was
% freed at N1 and again at N2, N1 < N2, with no
% allocation of Block between them, under the
% assumption that the log is complete.
double_free_witness(Block, N1, N2) :-
    log_complete(heap_op/4),
    heap_op(N1, free, Block, _),
    heap_op(N2, free, Block, _),
    N1 < N2,
    no_alloc_between(Block, N1, N2).

% double_free(Block): some pair of frees of Block
% is a witness.
double_free(Block) :-
    double_free_witness(Block, _, _).

% double_free(Block, Evidence): explainer form,
% built from the witness that was recognized.
double_free(Block, E) :-
    double_free_witness(Block, N1, N2),
    heap_op(N1, free, Block, T1),
    heap_op(N2, free, Block, T2),
    Facts = [ log_complete(heap_op/4),
              heap_op(N1, free, Block, T1),
              heap_op(N2, free, Block, T2),
              no_alloc_between(Block, N1, N2)
            ],
    E = evidence('Double Free (Process Heap)',
                 Block, Facts).

?- double_free(B), format("~16r~n", [B]).
210a2041040
B = 2270460907584 ;
false.

?- forall(double_free(_, E), print_evidence(E)).
Double Free (Process Heap)
  instance: 2270460907584
  log_complete(heap_op/4)
  heap_op(3, free, 2270460907584, 103)
  heap_op(5, free, 2270460907584, 104)
  no_alloc_between(2270460907584, 3, 5)
true.

?- double_free(B), corrupt_entry(_, B, F).
B = 2270460907584,
F = bad_size ;
false.
```

The recognition is in one place, `double_free_witness/3`, which binds the block and the two sequence numbers that make it a double free; the enumerator form discards the numbers and the explainer form keeps them and looks up the two operations by them, which is deterministic because the sequence number is the key of the log. The explainer therefore cites the pair that was recognized and no other. An explainer written the other way, calling `double_free/1` for the block and then selecting two frees of it afresh, would be true of the block and wrong about the evidence: on the log above it would also cite operations 5 and 8, a free and the free after a legitimate reallocation,

because the block-level test says nothing about which pair proved it. The witness predicate is what prevents that, and the check below exercises it.

The negation in `no_alloc_between/3` is the interval condition. Two frees of one block are a double free only if the block was not allocated again between them, and `\+` succeeds iff the goal inside it has no solution under the current bindings, which here are `Block`, `N1`, and `N2`. The reading is safe because every variable the negated goal shares with the clause is bound when it is called; `N3` is local to it. The condition is read under the **completeness assumption** for the log: `\+` treats what is not recorded as not having happened, which is the closed-world assumption of Chapter 7, and it is only as good as the recording. The assumption is therefore a fact of the chapter, `log_complete/1`, in the manner of `threshold/2`: the witness clause consults it, so a knowledge base whose analyst has not granted it recognizes no double free, and the explainer cites it, so a reader of the report who does not grant it knows which finding falls. The evidence list holds every goal the clause consulted, in the order of the clause and including the interval condition, and each is a goal that succeeds, as Chapter 17 requires.

The block at `0x...1040` was freed by thread 103 at the third operation and again by thread 104 at the fifth, with no allocation between, and it is the block whose header the verification of Section 21.6 rejected: the two extensions agree, which is what an analyst looks for when two sources of evidence are available. That threads 103 and 104 are the deadlocked pair of Chapter 20 is a coincidence the data was chosen to have; two threads releasing one buffer under a lock that was meant to serialize them is one story that fits all three facts, and the facts do not single it out. The seventh and eighth operations are the block handed out again and freed by its new owner. The frees at 5 and 8 have the allocation at 7 between them, and so do the frees at 3 and 8; neither pair is a witness, and the explainer never cites either. The following is a check of that boundary case:

```
?- double_free_witness(B, N1, N2).
B = 2270460907584,
N1 = 3,
N2 = 5 ;
false.

?- double_free_witness(_, 5, 8).
false.

?- double_free(_, evidence(_, _, Fs)),
   member(heap_op(8, _, _, _), Fs).
false.

?- forall(double_free(_, evidence(_, _, Fs)),
          maplist(call, Fs)).
true.
```

The one witness is the pair 3 and 5; the pair 5 and 8 is not one; no evidence term mentions operation 8; and every leaf of every evidence term holds. A block freed three times in a row has three witnesses, one per pair of frees with no allocation between, and `double_free/1` answers once for each; a report that wants the block once wraps the goal in `distinct/2`, which Section 21.11 introduces.

Without a log at all, the pattern is recognized from the dump in the way Chapter 19 recognizes exceptions, by the heap manager's exception or breakpoint in the stack trace of the second free, and the first free remains an inference. With a log that has gaps, `log_complete/1` is not a fact the analyst can state, and the recognizer is silent by design; a weaker recognizer that reported two frees with no recorded allocation between them as a suspected double free would need the suspicion in its pattern name and its evidence, which this chapter does not define.

21.8 Invalid Handle (General)

The pattern names the use of a handle value that does not refer to an object in the process's handle table: a handle closed already, never opened, or belonging to another process. The process sees it as the status `STATUS_INVALID_HANDLE`, `0xc0000008`, when the system raises it as an exception, or as the Win32 error `ERROR_INVALID_HANDLE`, 6, when the API returns failure and the thread's last error records it. The schema holds both: `exception_record/3` for a raised status and `last_error/2` for the recorded code.

```
% invalid_handle(TID): thread TID raised the
% invalid-handle status or recorded the
% invalid-handle error.
invalid_handle(TID) :-
    thread(TID, _),
    (   exception_record(TID, 0xc0000008, _)
    ;   last_error(TID, 0x6)
    ).

?- invalid_handle(T).
false.

?- last_error(T, C), C =\= 0,
   format("~16r~n", [C]).
3e6
T = 109,
C = 998.
```

The dump has no instance, and the predicate says so. The only thread with a nonzero last error is thread 109, whose code `0x3e6` is `ERROR_NOACCESS`, the Win32 rendering of the access violation that Chapter 19 treats; it is not a handle error. A recognizer that finds nothing in the shared data still earns its place: a report that lists what was looked for and not found is more useful than one that lists only what was found, and the report of Chapter 23 does both.

The last error is a weak witness. It is one value per thread, overwritten by every API call that sets it, and it records the most recent failure the thread saw, which may be unrelated to the thread's present state; a `0` says that the last call that set it succeeded, not that no call failed. The exception code is a strong witness but a rare one, since most API functions return failure rather than raise. And neither can see the handle that is valid but wrong, open and pointing at an object other than the one the code believes: that is a semantic error the handle table does not record, and no predicate over this schema finds it.

21.9 Paged Out Data

The pattern names memory the dump refers to but does not contain. The debugger shows such memory as question marks, and a stack trace, a heap walk, or a structure dump that runs into it stops there. In a kernel or complete memory dump the usual cause is that the pages were paged out to disk when the dump was written, which gives the pattern its name; in a process memory dump, a page can also be missing because the dump kind did not include it. From the dump alone, the fact is unavailability, and the cause is an inference, which is why the extension below is named for the fact and the recognizer for the pattern:

```
% unavailable(Base, Size): a range of the address
% space whose contents the dump does not hold.
% An extension of the schema.
unavailable(0x0000_0210_a300_0000, 0x100000).
unavailable(0x0000_7ffb_7a01_0000, 0x4000).

% paged_out_data(Addr): the contents at Addr are
% not in the dump.
paged_out_data(Addr) :-
    unavailable(Base, Size),
    Base =< Addr,
    Addr < Base + Size.

?- paged_out_data(0x0000_0210_a300_8000).
true ;
false.

?- paged_out_data(0x0000_0210_a204_1040).
false.

?- module(cachelib, Base, _),
   paged_out_data(Base + 0x10000).
Base = 140718060339200.
```

The first range is a megabyte inside the heap region, and the second is 16 KB of the image of `cachelib`, the module without symbols; a return address into those pages would resolve to no instruction. The `;` after the first answer is the second `unavailable/2` fact waiting to be tried, and the `false.` is its failure; a test that should be deterministic wraps the goal in `once/1`. The third query passes an unevaluated expression, and the arithmetic comparison operators evaluate both sides, so `Base + 0x10000` is compared as a number; a predicate that stored the argument rather than comparing it would need `is/2` first.

What the predicate cannot do is what the pattern is about: it cannot read the data. Every recognizer in this book that walks a structure, from a stack trace to a heap, assumes the facts are complete, and an unavailable range is a place where the facts end without the structure ending. A report should say where that happened, and `unavailable/2` lets it; the analyst then knows that the absence of a frame or a block below that point is an absence of evidence.

21.10 Shared Buffer Overwrite

The pattern names a buffer shared between threads or between processes, through a mapped section or through a pointer passed across a boundary, that one party wrote while another was reading, so that the reader found contents that do not match the structure it expected. In a dump it appears as data that is well formed by the writer's layout and malformed by the reader's, and as two or more agents holding the same address. The schema's evidence is indirect: `handle_count(4120, section, 6)` says the process has six section handles, which is the mechanism of sharing between processes, and the parameters of Section 21.13 say which addresses two frames hold in common. Neither says what was overwritten. A recognizer would need the expected layout of the buffer, which is knowledge about the program and not content of the dump, and the chapter leaves the pattern in prose for that reason.

21.11 Value References

The pattern names the search for a value, an address, a handle, a magic number, in every place the dump holds it: the places are the value's context, and a value that appears on a thread's stack and in a synchronization object, or as a parameter and as a heap block, is explained by the second occurrence in a way the first alone does not give. The debugger searches memory for the value; the schema holds the summaries, and the raw words of a stack are an extension. `raw_stack/3` gives a few words of thread 109's stack:

```
% raw_stack(TID, Addr, Word): the 8-byte Word at
% stack address Addr of thread TID. An extension
% of the schema.
raw_stack(109, 0xe4_1a7f_f6a0, 0x210_a204_1040).
raw_stack(109, 0xe4_1a7f_f6a8, 0x7ffb_8d10_3e8c).
raw_stack(109, 0xe4_1a7f_f6b0, 0x0).
raw_stack(109, 0xe4_1a7f_f6b8, 0x18).
raw_stack(109, 0xe4_1a7f_f6c0, 0x7ff6_1d13_a0c0).
raw_stack(109, 0xe4_1a7f_f6c8, 0x7ffb_b2a1_2f10).
raw_stack(109, 0xe4_1a7f_f6d0,
          0xcdcd_cdcd_cdcd_cdcd).

% reference(Value, Where): Value occurs at Where.
reference(Value, stack(TID, Addr)) :-
    raw_stack(TID, Addr, Value).
reference(Value, critical_section) :-
    critical_section(Value, _, _, _).
reference(Value, exception_target(TID)) :-
    exception_info(TID, target, Value).
reference(Value, heap_block) :-
    distinct(Value, heap_op(_, _, Value, _)).
```

```
% value_references(Value, Refs): Refs are the
% places Value occurs, at least two of them.
value_references(Value, Refs) :-
    distinct(Value, reference(Value, _)),
    findall(Where, reference(Value, Where), Refs),
    Refs = [_, _ | _].

% print_references: every value with at least two
% references, one line per reference.
print_references :-
    forall(value_references(V, Refs),
           ( format("~16r~n", [V]),
             forall(member(R, Refs),
                    format("  ~w~n", [R]))
           )).

?- print_references.
210a2041040
  stack(109,979697137312)
  heap_block
18
  stack(109,979697137336)
  exception_target(109)
7ff61d13a0c0
  stack(109,979697137344)
  critical_section
true.
```

Three values on the stack of the faulting thread are referenced elsewhere: the double-freed block of Section 21.7, the faulting target address `0x18` of the exception record, and the address of the first critical section of the deadlock in Chapter 20. The first two give the thread's context in the terms of other patterns, and the third says that the address of the lock two other threads are waiting for was in this thread's hands, as a local variable or an argument, which the stack trace of frames alone does not show.

The first subgoal of `value_references/2` deserves attention. `reference/2` with both arguments unbound enumerates every occurrence of every value, and the same value occurs several times; `distinct/2` from `library(solution_sequences)` passes each value once, and the `findall/3` that follows runs with `Value` bound and collects that value's places. Without `distinct/2` the predicate would answer once per occurrence, with the same list of references each time. The `heap_block` clause uses `distinct/2` for the same reason at the level of a single value, since a block appears in several operations of the log. `~w` prints the terms without spaces after commas; `~q` would print the same here, and `~p` would go through `portray/1`, which this book does not define.

The formalization searches the facts it has, and the facts are a few words of one stack. A search of the whole address space is a different operation in scale, and a value that is small, such as `0x18`, occurs by chance in memory of any size, so the pattern is only as discriminating as the value is distinctive. The predicate also treats a word as a value and nothing else: a pointer into the middle of a structure, which the debugger's symbol-aware search would attribute to the structure, is here a number that matches or does not.

21.12 Regular Data

The pattern names data whose bytes show a visible regularity, the same byte repeated, a run of ascending values, readable text, which an analyst recognizes at a glance as a fill pattern, a string, or an array rather than as pointers and sizes. The regularities are cheap to test on words. A word made of one byte repeated is equal to itself shifted by a byte in its low seven bytes, and the test is one comparison of two bitwise operations:

```
% repeated_byte(Word): the 8-byte Word is one byte
% value repeated eight times.
repeated_byte(Word) :-
    Word >> 8 =:= Word /\ 0x00ff_ffff_ffff_ffff.

% regular_data(TID, Addr): the stack word of TID
% at Addr is a nonzero repeated byte.
regular_data(TID, Addr) :-
    raw_stack(TID, Addr, Word),
    Word =\= 0,
    repeated_byte(Word).

?- repeated_byte(0x4141_4141_4141_4141).
true.

?- repeated_byte(0x4141_4141_4141_4142).
false.

?- regular_data(T, A), raw_stack(T, A, W),
   format("~16r~n", [W]).
cdcdcdcdcdcdcdcd
T = 109,
A = 979697137360,
W = 14829735431805717965.
```

The word of `0xcd` bytes is the fill value some debug allocators write into memory they hand out before the program has written anything there, and its presence on the stack says that a local variable or a copied block was never initialized. Zero is excluded from `regular_data/2` because a zero word is the most regular datum there is and the least informative; `repeated_byte/1` accepts it, since the arithmetic is indifferent. SWI-Prolog's integers are unbounded, so the mask and the shift are exact on 64-bit words; in an ISO system with bounded integers the word `0xcdcd_cdcd_cdcd_cdcd` would overflow a signed 64-bit integer, and the toplevel's decimal rendering of it above is a number that fits only because the integers are unbounded.

The pattern as the catalog describes it is wider than a byte repeated: text, counters, and vtable-like sequences of nearby addresses are regular too, and each has its own test. The chapter gives the one that fits in a line and leaves the others as the shape to follow. The interpretation is the analyst's in every case: `0xcd` bytes are a fill pattern only if a debug allocator was in use, and a dump does not say which allocator built the heap.

21.13 Data Correlation (Function Parameters)

The pattern names the relation between the parameters the debugger reconstructs for the frames of a stack trace and other data of the dump: the faulting address that is a parameter plus a small offset, the block that is a parameter of two frames, the handle that is a parameter and also in the handle table. The correlation is what turns a number in a frame into a role in the failure. The schema has no parameters, and `frame_param/4` extends it with the ones the debugger showed for thread 109:

```
% frame_param(TID, N, I, Value): the I-th
% parameter of frame N of thread TID as the
% debugger reconstructs it. An extension of the
% schema.
frame_param(109, 0, 1, 0x0).
frame_param(109, 0, 2, 0x0000_0210_a204_1040).
frame_param(109, 0, 3, 0x400).
frame_param(109, 1, 1, 0x0000_0210_a003_2a00).
frame_param(109, 1, 2, 0x0000_0210_a204_1040).
frame_param(109, 2, 1, 0x0000_0210_a003_2a00).

% data_correlation(TID, Param, Relation): the
% value of Param, a parameter of a frame of TID,
% is related to another datum of the dump.
data_correlation(TID, param(N, I), Relation) :-
    exception_info(TID, target, Target),
    frame_param(TID, N, I, Base),
    Offset is Target - Base,
    0 =< Offset,
    Offset < 0x1000,
    Relation = fault_at_offset(Offset).
data_correlation(TID, param(N, I), Relation) :-
    frame_param(TID, N, I, Block),
    corrupt_entry(_, Block, Finding),
    Relation = corrupt_block(Finding).
data_correlation(TID, param(N, I), Relation) :-
    frame_param(TID, N, I, Value),
    frame_param(TID, N2, I2, Value),
    N2 > N,
    Relation = shared_with(param(N2, I2)).

?- data_correlation(109, P, R).
P = param(0, 1),
R = fault_at_offset(24) ;
P = param(0, 2),
R = corrupt_block(bad_size) ;
P = param(1, 2),
R = corrupt_block(bad_size) ;
P = param(0, 2),
R = shared_with(param(1, 2)) ;
P = param(1, 1),
R = shared_with(param(2, 1)) ;
false.
```

```
?- frame(109, 0, M, F, _).
M = dbclient,
F = 'Buffer::Copy'.
```

The first correlation is the **NULL Pointer (Data)** (page 360) of Chapter 19 seen from the parameters: the first parameter of `Buffer::Copy`, which for a method is the object itself, is zero, and the fault read address `0x18`, so the function read a field at offset 24 of an object that was not there. The second and third say that the second parameter of the same frame and of its caller is the block that the heap verification rejected and the log showed freed twice, so the copy was reading from a block that no longer belonged to anyone. The last two are the propagation of values down the call chain, which is what a parameter shared by a frame and its caller means, and which confirms that the debugger's reconstruction of the two frames is consistent.

The predicate writes each relation as a separate clause with the relation term built last, so that a new kind of correlation is a new clause and the enumerator shape `data_correlation(+TID, -Param, -Relation)` is kept. The three relations are the ones the shared data can show. The catalog's pattern is open ended, since any datum of the dump can correlate with a parameter, and the formalization is a list of the correlations someone thought to write down. It is also only as good as the parameters, which the debugger reconstructs from registers and stack slots under assumptions about the calling convention that are not always met; a parameter that is wrong correlates with nothing, or with the wrong thing, and the dump does not mark which parameters it is sure of.

Chapter 22. Modules, Versions, and Environment

A process is its threads and its memory, and it is also the code that was loaded into it and the environment it was started in. The patterns of this chapter read the module list of a process memory dump, which the debugger prints with `lm` and `lmv`, for what it says about versions, vendors, symbols, and hooks; they read the environment block for what changed between the machine where the application works and the machine where it failed; and they look, in kernel and complete memory dumps, at the list of processes rather than the list of modules. The schema of Chapter 15 holds the module list as `module/3` and `module_info/3`, the hooks as `hooked/3`, and the last error codes as `last_error/2`; the environment, the executable regions outside modules, the process properties, and the processes of a system dump are extensions, each introduced where it is used.

Two kinds of knowledge meet in this chapter, and the predicates keep them apart. The dump says which modules were loaded and what their version resources said; the analyst knows which modules the operating system supplies, which the product deploys, and what the environment should have been. The second kind is encoded as facts too, `system_module/1`, `expected_module/1`, `expected_env/2`, and every recognizer that compares the dump with expectation says so in its definition. A pattern recognized by such a comparison is a statement about the difference between the dump and the analyst's model, and it is wrong exactly when the model is.

```
:- [dump1].

% pattern(Name, Artifact, Chapter): the catalog
% entries treated in this chapter.
pattern('Module Variety', memory_dump, 22).
pattern('Not My Version', memory_dump, 22).
pattern('Unknown Component', memory_dump, 22).
pattern('No Component Symbols', memory_dump, 22).
pattern('Missing Component (General)',
    memory_dump, 22).
pattern('Hooked Functions (User Space)',
    memory_dump, 22).
pattern('Hooked Modules', memory_dump, 22).
pattern('Hidden Module', memory_dump, 22).
pattern('Changed Environment', memory_dump, 22).
pattern('Virtualized Process (WOW64)',
    memory_dump, 22).
pattern('Deviant Module', memory_dump, 22).
pattern('Same Vendor', memory_dump, 22).
pattern('Top Module', memory_dump, 22).
pattern('Origin Module', memory_dump, 22).
pattern('Module Hint', memory_dump, 22).
pattern('Historical Information',
    memory_dump, 22).
pattern('Last Error Collection', memory_dump, 22).
pattern('Zombie Processes', memory_dump, 22).
pattern('Special Process', memory_dump, 22).
pattern('Frozen Process', memory_dump, 22).
pattern('Busy System', memory_dump, 22).
```

```
pattern('Lateral Damage', memory_dump, 22).
pattern('Ubiquitous Component', memory_dump, 22).

% threshold(Name, Value): the analyst's limits.
threshold(ubiquity, 0.7).

% system_module(Module): a module supplied by the
% operating system or the C runtime. An extension
% of the schema: the analyst's knowledge.
system_module(ntdll).
system_module(kernelbase).
system_module(kernel32).
system_module(ucrtbase).
system_module(msvcp140).

% vendor(Module, Vendor): the company recorded for
% Module; system for a system module without one;
% unrecorded otherwise.
vendor(Module, Vendor) :-
    module(Module, _, _),
    (   module_info(Module, company, Company)
    ->  Vendor = Company
    ;   system_module(Module)
    ->  Vendor = system
    ;   Vendor = unrecorded
    ).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- aggregate_all(count, pattern(_, _, 22), N).
N = 23.

?- vendor(dbclient, V).
V = 'Example Corp'.

?- vendor(ntdll, V).
V = system.
```

`vendor/2` is the chapter's view of the `company` key. The shared dump records no company for the five system modules, which is how a pipeline that takes the company from the version resource leaves a module whose resource it did not read, and the chained if-then-else of Chapter 7 fills the gap from the analyst's list and names the remaining case `unrecorded` rather than leaving it to fail. The `module(Module, _, _)` in front makes the predicate enumerate modules when its first argument is unbound.

22.1 Module Variety

The pattern names the number and diversity of the modules loaded into a process, which an analyst reads from the length of the `lm` listing before reading any line of it: a process with three hundred modules from forty vendors has more places for a fault to come from than a process with thirty from two. Chapter 9 counted the modules. This chapter adds the share of them that are not system modules, and the set of vendors:

```
% module_variety(Count): the number of modules
% loaded in the process.
module_variety(Count) :-
    aggregate_all(count, module(_, _, _), Count).

% third_party_ratio(Ratio): the share of loaded
% modules that are not system modules.
third_party_ratio(Ratio) :-
    module_variety(Total),
    aggregate_all(count,
                  ( module(M, _, _),
                    \+ system_module(M)
                  ),
                  Others),
    Ratio is Others / Total.

% vendor_variety(Vendors): the set of vendors of
% the loaded modules.
vendor_variety(Vendors) :-
    aggregate_all(set(V), vendor(_, V), Vendors).
```

```
?- module_variety(N), third_party_ratio(R).
N = 10,
R = 0.5.

?- vendor_variety(_Vs),
   forall(member(V, _Vs), format("~a~n", [V])).
CacheWorks
Example Corp
Unknown
system
true.

?- aggregate(count, M^vendor(M, V), N).
V = 'CacheWorks',
N = 1 ;
V = 'Example Corp',
N = 3 ;
V = 'Unknown',
N = 1 ;
V = system,
N = 5.
```

Ten modules, half of them outside the system, from three recorded vendors and one unrecorded. The last query is the `aggregate/3` of Chapter 9 grouping by the free variable `V`, and it is the summary an analyst forms at a glance from `lmv`: the application's own vendor has three modules, a cache library vendor has one, and one module claims no vendor at all, which Section 22.3 takes up. The ratio is a number between 0 and 1, and `aggregate_all/3` with `count` yields 0 for a process with only system modules, so the division is safe whenever the process has any module at all.

The count depends on what counts as a module and as a system module. The list `system_module/1` is the analyst's and is short; a real list would run to hundreds of names, or would be replaced by the `company` key when the pipeline records it for every module. The variety says nothing about which

modules matter: a process may load a hundred modules and fault in one, and the patterns that follow are the ones that pick it out.

22.2 Not My Version

The pattern names a module whose version is not the one the analyst, or the application, expects: a component of the product shipped at a different release than the rest, a library updated on one machine and not another. The version resources in the dump are the evidence, and the comparison is between modules of the same vendor, since a version number means nothing across vendors. The executable's version is taken as the reference.

Version numbers are atoms in the schema, `'3.2.1.0'`, and atoms compare as text: `'3.10.0.0'` precedes `'3.9.0.0'` in the standard order of atoms, because the character `1` precedes `9`. A version is compared as the list of its numbers, which `atomic_list_concat/3` in its splitting mode and `atom_number/2` produce, and lists of integers compare in the standard order of terms element by element, with a shorter list preceding its extensions:

```
% version_parts(Version, Parts): the numbers of a
% dotted version atom, as a list.
version_parts(Version, Parts) :-
    atomic_list_concat(Texts, '.', Version),
    maplist(atom_number, Texts, Parts).

% main_module(Module): the module of the process's
% executable, by its file name.
main_module(Module) :-
    process(_, Executable),
    file_name_extension(Base, _, Executable),
    downcase_atom(Base, Module),
    module(Module, _, _).

% version_of(Module, Parts): the version of Module
% as a list of numbers; a module has one version.
version_of(Module, Parts) :-
    once(module_info(Module, version, Version)),
    version_parts(Version, Parts).

% not_my_version(Module): Module is from the same
% vendor as the executable but has a different
% version.
not_my_version(Module) :-
    main_module(Main),
    vendor(Main, Vendor),
    vendor(Module, Vendor),
    Module \== Main,
    version_of(Main, MainParts),
    version_of(Module, Parts),
    MainParts \== Parts.
```

```
% older_than(M1, M2): the version of M1 precedes
% that of M2.
older_than(M1, M2) :-
    version_of(M1, P1),
    version_of(M2, P2),
    P1 @< P2.

?- version_parts('3.1.0.7', P).
P = [3, 1, 0, 7].

?- '3.10.0.0' @< '3.9.0.0'.
true.

?- version_parts('3.10.0.0', A),
   version_parts('3.9.0.0', B), A @< B.
false.

?- main_module(M).
M = appserver.

?- not_my_version(M).
M = dbclient ;
false.

?- older_than(dbclient, appserver).
true.
```

The database client is at 3.1.0.7 while the executable and the report engine are at 3.2.1.0, and it is older. The `once/1` in `version_of/2` states that a module has one version fact and makes the lookup deterministic; without it `module_info/3`, which has three facts for `dbclient`, would leave a choice point behind every version comparison, and the `true.` of the last query would be a `true ;` followed by `false.`. `main_module/1` derives the module name from the executable's file name with `file_name_extension/3`, which splits `'AppServer.exe'` into its base and extension, and `downcase_atom/2`, which gives the lowercase atom the schema uses for module names; the final `module/3` subgoal checks that the result is a loaded module, since a process whose executable is not in the module list is a **Truncated Dump** (page 452) of Chapter 23. The second and third queries are the point of `version_parts/2`: as atoms the versions compare wrongly, as lists of numbers they compare rightly.

The predicate compares with the executable, which is one choice of reference and not the only one. A product whose components are versioned independently has no single expected version, and the comparison would then be with a manifest, as `expected_module/1` is in Section 22.5. The version resource is also what the module says of itself; a module rebuilt without updating its resource carries the old number, and the dump cannot tell. The catalog's pattern includes the analyst's realization that the dump is of a build other than the one being debugged, which no predicate decides.

22.3 Unknown Component

The pattern names a module about which the dump says nothing useful: no vendor, no description, often no symbols, a file name that identifies nothing. The schema's convention for unknowns, from

Chapter 15, admits two renderings: a `company` value of `'Unknown'` when the debugger reported the module and could say nothing about its vendor, and no `company` fact at all when nothing was reported; `vendor/2` maps the second case to `unrecorded`. Both are the pattern:

```
% unknown_component(Module): the dump records no
% vendor for Module, or records it as unknown.
unknown_component(Module) :-
    vendor(Module, Vendor),
    memberchk(Vendor, ['Unknown', unrecorded]).

?- unknown_component(M).
M = thirdparty_hook.

?- unknown_component(M), module_info(M, K, V).
M = thirdparty_hook,
K = version,
V = '1.0.0.1' ;
M = thirdparty_hook,
K = symbols,
V = none ;
M = thirdparty_hook,
K = company,
V = 'Unknown'.
```

One module, with a version, no symbols, and an unknown company, and it is the last `module/3` fact, so the answer ends in `.`. The second query lists everything the dump records about it, which is the analyst's next move: an unknown component is identified, if at all, from its file name, its version, its load address, and what it does, and Sections 22.6 and 22.15 show that this one hooks the file API and says so in its name.

The predicate takes the dump's word. A module with a company of `'Unknown'` may be well known to the analyst, and a module with a recorded company may be unknown in every sense that matters; the pattern's name is about the analyst's knowledge, and the predicate approximates it by the dump's. A `system_module/1` list that is incomplete produces `unrecorded` for a system module the pipeline did not describe, and a recognizer that mattered would want that list complete before trusting the second case.

22.4 No Component Symbols

The pattern names a module for which the debugger has no symbol file, so that frames in it show the module name with an offset, or the nearest exported name with a large offset, in place of a function. The schema's `symbols` key has three values: `pdb` for a full symbol file, `export` for the module's export table only, and `none`. Both of the last two are the pattern, in different degrees:

```
% no_component_symbols(Module): the debugger had
% no symbol file for Module.
no_component_symbols(Module) :-
    module_info(Module, symbols, Kind),
    Kind \== pdb.
```

```
?- no_component_symbols(M).
M = dbclient ;
M = cachelib ;
M = thirdparty_hook.

?- frame(T, N, M, 'Unknown', O).
T = 101,
N = 4,
M = cachelib,
O = 10768.

?- forall(frame(_, _, M, 'Unknown', _),
          no_component_symbols(M)).
true.

?- no_component_symbols(M),
   \+ frame(_, _, M, 'Unknown', _).
M = dbclient ;
M = thirdparty_hook.
```

Three modules lack symbol files, and one of them, the cache library, has a frame on the stack trace of thread 101 that the debugger could not name: its function is `'Unknown'` and its offset, `0x2a10`, is from the module base. The `forall/2` confirms that every unnamed frame belongs to a module without symbols, which is the consistency a dump should have. The last query gives the other two: the database client, whose export symbols name its frames on threads 103, 104, and 109 but may name them wrongly, and the hooking module, which has no frame at all.

The export case is the subtle one. An exported name is correct for the function it names and wrong for every function that follows it in the module without being exported, and the debugger shows the latter as the nearest export plus a large offset. The schema records offsets, and the **Coincidental Symbolic Information** (page 347) pattern of Chapter 18 is the check: a frame in a module with `export` symbols and a large offset is a frame whose name should not be trusted. The predicate here says which modules that check applies to, and does not perform it.

22.5 Missing Component (General)

The pattern names a module that should be in the process and is not: a plug-in that failed to load, a library a configuration disabled, a component an installer omitted. Absence is not in the dump. It is in the difference between the module list and a list of what should be there, and the second list is knowledge about the product, encoded here as `expected_module/1`:

```
% expected_module(Module): the modules the product
% deploys. An extension of the schema: knowledge
% about the product, not content of the dump.
expected_module(appserver).
expected_module(reportengine).
expected_module(dbclient).
expected_module(cachelib).
expected_module(authplugin).
```

```
% missing_component(Module): an expected module is
% not loaded.
missing_component(Module) :-
    expected_module(Module),
    \+ module(Module, _, _).

?- missing_component(M).
M = authplugin.

?- expected_module(M), module(M, _, _),
   format("~a~n", [M]), fail.
appserver
reportengine
dbclient
cachelib
false.
```

The authentication plug-in is expected and absent. The second query is the failure-driven loop of Chapter 7 printing the expected modules that are present, and its `false.` is the loop's natural end rather than a finding. The negation is safe because `Module` is bound by `expected_module/1` before `\+` is reached.

The predicate finds absence relative to the manifest and nothing else. A module that was loaded and later unloaded leaves a trace the debugger shows in a separate list, the catalog's Unloaded Module, which the schema does not carry; a module loaded at a path other than the expected one is present and is a different pattern. And the manifest is a claim about the deployment, so the recognition is only as good as the claim: a plug-in that is optional on this installation is not missing, however the manifest reads.

22.6 Hooked Functions (User Space)

The pattern names functions whose code in memory does not match their code on disk because another module patched them, to intercept calls to the system's file, registry, or network API. The debugger finds the patches by comparing memory with the image, and the schema records each as `hooked(Function, Module, Hooker)`. The recognizer groups the hooks by the module that was hooked and the module that did the hooking:

```
% hooked_functions(Module, Hooker, Functions): the
% functions of Module that Hooker hooks.
hooked_functions(Module, Hooker, Functions) :-
    bagof(Function,
          hooked(Function, Module, Hooker),
          Functions).

?- hooked_functions(M, H, Fs).
M = kernelbase,
H = thirdparty_hook,
Fs = ['CreateFileW', 'ReadFile'].
```

```
?- hooked_functions(kernelbase, _, Fs),
   length(Fs, N).
Fs = ['CreateFileW', 'ReadFile'],
N = 2.
```

Two functions of `kernelbase`, the ones that open and read files, are hooked by the unknown component of Section 22.3. `bagof/3` groups by its free variables, here the hooked module and the hooker, which is the grouping wanted; with `findall/3` the three arguments would have to be enumerated and grouped by hand, and with a dump in which several modules hook several others, `bagof/3` produces one answer per pair on backtracking.

The predicate is a view of `hooked/3`, and `hooked/3` is only as complete as the debugger's comparison. The comparison covers the modules whose images the debugger could find and the regions it was asked to check, and a hook installed by rewriting an import table rather than the function's code is a different mechanism that the comparison does not see. The pattern also does not say that a hook is wrong: security products, profilers, and compatibility layers hook the same functions for a living, and the analyst's judgment turns on who the hooker is.

22.7 Hooked Modules

The pattern names the modules on either side of a hook: those whose functions have been patched, and those that did the patching. Both are projections of `hooked/3`, and `distinct/2` from `library(solution_sequences)` makes each module appear once however many functions are involved:

```
% hooked_modules(Module): some function of Module
% is hooked.
hooked_modules(Module) :-
    distinct(Module, hooked(_, Module, _)).

% hooking_module(Hooker): Hooker hooks a function
% of some module.
hooking_module(Hooker) :-
    distinct(Hooker, hooked(_, _, Hooker)).

?- hooked_modules(M).
M = kernelbase ;
false.

?- hooking_module(H), unknown_component(H),
   no_component_symbols(H).
H = thirdparty_hook ;
false.
```

The second query is three patterns composed: a module that hooks the file API, that no vendor claims, and that ships without symbols. Each is a weak signal and the conjunction is not, and the conjunction is one line, which is the argument of Chapter 17 for patterns as predicates. The `false.` after each answer is `distinct/2` finding that the remaining solutions of its goal are repeats.

The limits of the previous section carry over. In addition, `hooking_module/1` names the module the debugger attributed the patch to, usually by the address the patched code jumps to, and a hook that jumps through memory no module owns, the **Hidden Module** (page 435) of the next section, has no name to give.

22.8 Hidden Module

The pattern names executable code in the address space that no entry of the module list accounts for: a module unlinked from the loader's lists to escape enumeration, or code written into allocated memory and never a module at all. `lm` does not show it; `!address` shows the region, and comparing the two is the recognition. The schema has the module list and not the regions, and `code_region/2` extends it with the executable regions:

```
% code_region(Base, Size): a region of executable
% memory in the address space. An extension of the
% schema.
code_region(0x0000_7ffb_b2a0_0000, 0x1f8000).
code_region(0x0000_7ff6_1d10_0000, 0x3c000).
code_region(0x0000_0210_c000_0000, 0x8000).

% hidden_module(Base, Size): an executable region
% that no loaded module covers.
hidden_module(Base, Size) :-
    code_region(Base, Size),
    \+ ( module(_, ModBase, ModSize),
         ModBase =< Base,
         Base + Size =< ModBase + ModSize
       ).

?- hidden_module(B, S),
   format("~16r ~16r~n", [B, S]).
210c0000000 8000
B = 2270963957760,
S = 32768.

?- code_region(B, _), module(M, B, _).
B = 140719010349056,
M = ntdll ;
B = 140695026270208,
M = appserver ;
false.
```

The first two regions are the images of `ntdll` and the executable, and the module list covers them; the third is 32 KB of executable memory in the heap range that nothing covers. The negated goal asks whether some module contains the region entirely, and the region is hidden when none does. The second query shows that the covered regions begin exactly at module bases, which is the case for an image mapped by the loader; a region that begins inside a module or straddles two is also uncovered by the test as written, and would deserve a finding of its own.

The predicate depends on `code_region/2`, which the shared dump does not have and which a debugger provides at the cost of walking the whole address space. It also cannot say what the hidden code is. Just-in-time compiled code, the catalog's JIT Code, trampolines of a legitimate hooking library, and injected code all look alike at this level: executable memory, no module. The analyst disassembles; the predicate points.

22.9 Changed Environment

The pattern names a difference between the environment a process was started in and the environment the analyst assumes: a variable set that enables a profiler or a compatibility layer, a path that points at a staging directory, a variable removed that the application reads. The environment block is in a process memory dump, and the debugger can print it; the schema does not record it, and `env/2` extends it, with `expected_env/2` for the baseline. The value `absent` in the baseline says that the variable must not be set:

```
% env(Name, Value): the environment block of the
% process. An extension of the schema.
env('COMPUTERNAME', 'APP01').
env('APP_ENV', staging).
env('COR_ENABLE_PROFILING', '1').
env('TMP', 'C:\\Windows\\Temp').

% expected_env(Name, Value): the baseline the
% analyst compares with; absent means the variable
% must not be set. An extension of the schema.
expected_env('APP_ENV', production).
expected_env('APP_HOME', 'C:\\App').
expected_env('COR_ENABLE_PROFILING', absent).
expected_env('TMP', 'C:\\Windows\\Temp').

% changed_environment(Name, Change): the variable
% Name differs from the baseline: changed, added,
% or removed.
changed_environment(Name, changed(Old, New)) :-
    expected_env(Name, Old),
    Old \== absent,
    env(Name, New),
    New \== Old.
changed_environment(Name, added(Value)) :-
    expected_env(Name, absent),
    env(Name, Value).
changed_environment(Name, removed(Old)) :-
    expected_env(Name, Old),
    Old \== absent,
    \+ env(Name, _).

?- changed_environment(N, C).
N = 'APP_ENV',
C = changed(production, staging) ;
N = 'COR_ENABLE_PROFILING',
C = added('1') ;
N = 'APP_HOME',
C = removed('C:\\App') ;
```

```
false.

?- env('TMP', V), expected_env('TMP', V).
V = 'C:\\Windows\\Temp'.
```

Three changes: the application was started in a staging configuration, a variable that asks the .NET runtime to load a profiler was set, and the application's home directory variable was not. Each is one clause, and the three clauses are exclusive by construction, since a baseline value is either `absent` or not and a variable is either set or not. The backslashes are doubled in the source because `\` begins an escape sequence in a quoted atom, and the toplevel prints them doubled for the same reason; the atom itself holds one backslash at each place.

The pattern is a comparison, and the baseline is the analyst's. A variable that is not in `expected_env/2` is not reported however strange its value, which is the closed-world assumption applied to the baseline rather than the dump; a complete baseline is as long as an environment, and the predicate is best used with a baseline that lists the variables the application reads. The dump also holds the environment at process start, as the loader copied it, not the environment of the shell that started the process, and a variable the process set for itself afterward is in the block too.

22.10 Virtualized Process (WOW64)

The pattern names a 32-bit process running on a 64-bit system under the WOW64 layer, whose dump shows two worlds at once: 32-bit code and stacks, and the 64-bit modules `wow64.dll`, `wow64win.dll`, and `wow64cpu.dll` that translate between them. The debugger must be told which world to show, and stack traces taken in the wrong one are meaningless. The schema records nothing about the process's architecture; the addresses of `dump1.pl` are 64-bit, which is evidence but not a record, and `process_info/2` extends the schema with the properties the debugger reports:

```
% process_info(Key, Value): properties of the
% process. An extension of the schema.
process_info(wow64, false).
process_info(pointer_size, 8).

% virtualized_process: the process runs 32-bit
% code under the system's WOW64 layer, by the
% debugger's report or by the layer's modules.
virtualized_process :-
    (   process_info(wow64, true)
    ;   module(wow64, _, _)
    ).

?- virtualized_process.
false.

?- process_info(pointer_size, S),
   module(_, Base, _), Base >= 1 << 32.
S = 8,
Base = 140719010349056 ;
```

```
S = 8,
Base = 140718977843200 ;
...
```

The process is not virtualized, by the debugger's report and by the absence of the layer's modules; the second query shows what the addresses alone suggest, that every module is loaded above the 4 GB that a 32-bit address can reach, which is consistent with a 64-bit process and would not be conclusive on its own, since a WOW64 process maps the 64-bit `ntdll` high too. `1 << 32` is the shift of Chapter 6 and evaluates inside the comparison. The disjunction in `virtualized_process/0` is the two sources of evidence; the predicate succeeds once per source that holds, and a report that wants a Boolean wraps it in `once/1`.

The pattern is otherwise prose in this chapter, because the shared dump cannot show it. In a WOW64 dump, every `frame/5` fact belongs to one of two stacks per thread, and the schema would need a sixth argument to say which; the module list would hold two of several system modules, with the same name and different bases, which `module/3` as a relation allows and the predicates above, which treat the name as a key, do not expect.

22.11 Deviant Module

The pattern names a module that stands apart from its companions in some recorded property: the one component of a vendor at another version, the one built without symbols, the one with a different timestamp. The recognition is relative to a group, here the modules of one vendor, and to a property, here any key of `module_info/3`:

```
% deviant_module(Module, Key): among at least
% three modules of one vendor, Module alone has
% its value of Key.
deviant_module(Module, Key) :-
    vendor(Module, Vendor),
    aggregate_all(count, vendor(_, Vendor), N),
    N >= 3,
    module_info(Module, Key, Value),
    forall(( vendor(Other, Vendor),
             Other \== Module
           ),
           ( module_info(Other, Key, OtherValue),
             OtherValue \== Value
           )).

?- deviant_module(M, K).
M = dbclient,
K = version ;
M = dbclient,
K = symbols ;
false.
```

```
?- deviant_module(M, K), module_info(M, K, V).
M = dbclient,
K = version,
V = '3.1.0.7' ;
M = dbclient,
K = symbols,
V = export ;
false.
```

The database client deviates twice from the other two modules of its vendor, in version and in symbols, and both are findings of earlier sections seen as one. The `forall/2` says that every other module of the vendor has a value for the key and that the value differs; a vendor with two modules has no majority and is excluded by `N >= 3`. The system modules are a group of five with identical `symbols` and no `version`, so none deviates.

The formalization chooses one grouping and one notion of deviance. Grouping by vendor is natural for a product's components and useless for a vendor with one module; grouping by load path or by timestamp needs facts the schema lacks. Deviance as "alone with its value" is strict: of five modules, two at one version and three at another, none is deviant by this definition, where an analyst would call the two the deviants. A definition by majority, with `aggregate/3` counting the values, is a few lines more and is left as the shape to follow.

22.12 Same Vendor

The pattern names a stack trace whose non-system frames all belong to modules of one vendor, so that the responsibility for the thread's state, whatever it is, lies with that vendor's code. The recognizer collects the non-system modules of a thread and checks that they share a vendor:

```
% stack_module(TID, Module): a non-system module
% with a frame on the stack trace of TID.
stack_module(TID, Module) :-
    frame(TID, _, Module, _, _),
    \+ system_module(Module).

% same_vendor(TID, Vendor, Modules): the two or
% more non-system modules on the stack trace of
% TID all come from Vendor.
same_vendor(TID, Vendor, Modules) :-
    thread(TID, _),
    setof(M, stack_module(TID, M), Modules),
    Modules = [_, _ | _],
    maplist(from_vendor(Vendor), Modules).

from_vendor(Vendor, Module) :-
    vendor(Module, Vendor).


?- same_vendor(T, V, Ms).
T = 102,
V = 'Example Corp',
Ms = [appserver, reportengine] ;
```

```
T = 103,
V = 'Example Corp',
Ms = [appserver, dbclient] ;
T = 104,
V = 'Example Corp',
Ms = [appserver, dbclient] ;
T = 107,
V = 'Example Corp',
Ms = [appserver, reportengine] ;
T = 109,
V = 'Example Corp',
Ms = [appserver, dbclient].

?- setof(M, stack_module(101, M), Ms).
Ms = [appserver, cachelib].
```

Five threads run entirely in the product's own code, among them both spiking threads, both halves of the deadlock, and the faulting thread; thread 101 does not, because the cache library from another vendor sits in its stack, and threads with one or no non-system module are excluded by the pattern `[_, _ | _]`. `setof/3` gives the modules as a set, and `maplist/2` with the closure `from_vendor(Vendor)` binds `Vendor` at the first module and requires every following module to match it, which is the "all the same" test in one call.

The vendor is the version resource's, with the caveats of Section 22.3, and the system modules are excluded by the analyst's list. The pattern as an analyst uses it also weighs where in the stack the vendor's frames are: a stack whose top frame is the vendor's is a stronger attribution than one where the vendor's code is a caller several frames down, and the predicate does not distinguish them.

22.13 Top Module

The pattern names the module of the top frame of a stack trace, the code that was executing or waiting when the dump was written. For a faulting thread it is where the fault was raised; for a running thread it is where the time is spent; for a waiting thread it is almost always `ntdll`, and the informative module is the first one below the system frames, which Chapter 18 treats under **Stack Trace** (page 329). The predicate is one fact lookup:

```
% top_module(TID, Module): the module of the top
% frame of TID.
top_module(TID, Module) :-
    frame(TID, 0, Module, _, _).

?- aggregate_all(count, top_module(_, ntdll), N).
N = 7.

?- top_module(T, M), \+ system_module(M).
T = 102,
M = reportengine ;
T = 107,
M = reportengine ;
T = 109,
M = dbclient.
```

Seven of ten threads have `ntdll` on top, which is what waiting looks like, and the three that do not are the two running threads in the report engine and the faulting thread in the database client. The indexing of Chapter 13 makes the second query cheap: with the first argument unbound and the second bound to `0`, SWI-Prolog's just-in-time indexing selects the clauses by the second argument, and the answers come without trying the other 68 frames.

The limit is the one the pattern's definition carries: the top frame is the most recent and not the most responsible. The spiking thread's top module is the report engine because `Table::Find` is where the loop spends its time, which is informative; the faulting thread's top module is where the exception was raised, which is informative too, but a fault raised inside a system function on behalf of a bad argument from the caller would name the system module, and the **Exception Module** (page 378) and **Problem Module** (page 379) patterns of Chapter 19 exist to make that distinction.

22.14 Origin Module

The pattern names the module that started a thread's activity, which in a stack trace is the deepest frame that is not the system's thread start code: the module whose function the thread was created to run. It attributes a thread to a component when the top of the stack is in shared code.

```
% origin_module(TID, Module): the module of the
% deepest non-system frame of TID.
origin_module(TID, Module) :-
    thread(TID, _),
    aggregate_all(max(N, M),
                  ( frame(TID, N, M, _, _),
                    \+ system_module(M)
                  ),
                  max(_, Module)).

?- origin_module(T, M).
T = 100,
M = appserver ;
T = 101,
M = appserver ;
T = 102,
M = appserver ;
T = 103,
M = appserver ;
T = 104,
M = dbclient ;
T = 107,
M = appserver ;
T = 108,
M = appserver ;
T = 109,
M = appserver.

?- frame(104, N, dbclient, F, _).
N = 5,
F = 'Connection::OnReply' ;
```

```
N = 6,
F = 'Reader::Loop'.
```

Eight threads originate in the application; the two idle pool threads have no non-system frame and `max/2` of nothing fails, so they are absent from the answers; and thread 104 originates in the database client, whose `Reader::Loop` is the deepest application frame, with the application's `Session::Lookup` above it as a callback. That thread is half of the deadlock of Chapter 20, and knowing that it is the database client's thread, not the application's, says which component's locking discipline is in question. The `max(N, M)` specification of `aggregate_all/3` with a witness, from Chapter 9, yields the module of the frame with the largest number.

The predicate takes "deepest non-system" as the definition and is wrong where a non-system module is in the thread's start path for other reasons: a thread pool implemented in the application would make every pooled thread originate in it, whatever work it was doing, and a thread created by a system API with a callback into application code originates, by this test, in the application, which is usually what is wanted but not always.

22.15 Module Hint

The pattern names the information carried by a module's name alone: a name that contains `hook`, `inject`, `guard`, or a vendor's abbreviation tells an analyst what the module is for before any symbol is loaded. The recognizer is a table of words and `sub_atom/5` searching for them in module names:

```
% hint_word(Word, Hint): a word in a module name
% and what it suggests.
hint_word(hook, hooking).
hint_word(inject, injection).
hint_word(guard, protection).
hint_word(shim, compatibility).
hint_word(cache, caching).

% module_hint(Module, Hint): the name of Module
% contains a hint word.
module_hint(Module, Hint) :-
    module(Module, _, _),
    hint_word(Word, Hint),
    sub_atom(Module, _, _, _, Word).

?- module_hint(M, H).
M = cachelib,
H = caching ;
M = thirdparty_hook,
H = hooking ;
false.
```

Two names carry hints, and the second is the hooking module of Section 22.6, whose name said what the hook check confirmed. `sub_atom/5` with the subatom bound and the positions unbound enumerates the places the word occurs, which for these names is one place each, and the `false.` is its search for a second occurrence.

The table is tiny and the matching is naive: `cache` in `cachelib` is a hint and `cache` in `apachecore` would be noise. The pattern is a prompt for the analyst's recognition, which a table of words approximates poorly and a longer table approximates less poorly; its value in a predicate is as one more weak signal in conjunctions such as that of Section 22.7, not as a finding of its own.

22.16 Historical Information

The pattern names what the analyst brings to a dump from outside it: previous dumps of the same application, the record of which versions were deployed when, the change log between the version in the dump and the version that worked. It is the pattern behind **Not My Version** (page 429), where the history is the expected version, and behind the series of Chapter 21, where the history is the earlier dumps. The schema has no place for it, because a dump has none; a knowledge base that grows with the analyst's work does, and Chapter 11's dynamic database and `annotate/2` are the mechanism. A fact `release(Module, Version, Date)` beside `module_info/3`, maintained from the product's release records, would let `not_my_version/1` say not only that a version differs but by how many releases and since when; it is not written here, since the data is the product's, not the dump's.

22.17 Last Error Collection

The pattern names the per-thread last error values read together: each thread records the Win32 error code of the last API call that set one, and the collection over all threads shows which threads met failures and of what kind, including failures the code ignored. The schema has `last_error/2` for the threads the debugger reported, and the codes are numbers that a small table names:

```
% win32_error(Code, Name): the names of some Win32
% error codes.
win32_error(0x0, 'ERROR_SUCCESS').
win32_error(0x2, 'ERROR_FILE_NOT_FOUND').
win32_error(0x5, 'ERROR_ACCESS_DENIED').
win32_error(0x6, 'ERROR_INVALID_HANDLE').
win32_error(0x8, 'ERROR_NOT_ENOUGH_MEMORY').
win32_error(0x57, 'ERROR_INVALID_PARAMETER').
win32_error(0x7e, 'ERROR_MOD_NOT_FOUND').
win32_error(0x3e6, 'ERROR_NOACCESS').

% error_name(Code, Name): the name of Code, or
% unknown.
error_name(Code, Name) :-
    (   win32_error(Code, Name)
    ->  true
    ;   Name = unknown
    ).
```

```
% last_error_collection(Collection): the TID-Code
% pairs of every thread with a recorded last
% error, by thread.
last_error_collection(Collection) :-
    findall(TID-Code, last_error(TID, Code),
            Pairs),
    keysort(Pairs, Collection).

% print_last_errors: one line per recorded thread.
print_last_errors :-
    last_error_collection(Collection),
    forall(member(TID-Code, Collection),
           ( error_name(Code, Name),
             format("~d~t~6|~16r~t~12|~a~n",
                    [TID, Code, Name])
           )).

?- print_last_errors.
100   0     ERROR_SUCCESS
101   0     ERROR_SUCCESS
102   0     ERROR_SUCCESS
103   0     ERROR_SUCCESS
104   0     ERROR_SUCCESS
109   3e6   ERROR_NOACCESS
true.

?- last_error_collection(_C),
   member(T-C, _C), C =\= 0.
T = 109,
C = 998.

?- thread(T, _), \+ last_error(T, _).
T = 105 ;
T = 106 ;
T = 107 ;
T = 108 ;
false.
```

Six threads have a recorded last error, and one of them is nonzero: thread 109's `ERROR_NOACCESS`, which is the Win32 rendering of its access violation and adds nothing to the exception record. Four threads have no record, which is a different thing from a record of zero, and the third query lists them: the schema leaves out a fact rather than inventing a value, as Chapter 15 prescribes, and a report that printed `0` for them would be claiming something the dump did not say.

The collection's limits are those of Section 21.8: one value per thread, overwritten by every call that sets it, recording the most recent failure rather than the important one. Its diagnostic use is in the aggregate: a dump in which many threads show `ERROR_NOT_ENOUGH_MEMORY` or `ERROR_ACCESS_DENIED` says something no single thread's value says, and `aggregate/3` grouping by code would count them. The naming table is the analyst's and incomplete; `unknown` is the honest name for a code it lacks.

22.18 Zombie Processes, Special Process, Frozen Process, and Busy System

Four patterns of the catalog concern kernel and complete memory dumps, where the artifact holds many processes and the first list the analyst reads is the process list rather than the module list. The schema of this book describes one process; Chapter 15 discussed its extension by a process dimension and did not use it. The following facts are a small instance of such an extension, one `sys_process/4` fact per process with its name, thread count, and state, and one `processor/2` fact per processor with the process whose thread was running on it:

```
% sys_process(PID, Name, Threads, State): a
% process of a kernel or complete memory dump,
% with its thread count and state. An extension of
% the schema for dumps that hold many processes.
sys_process(4, 'System', 180, active).
sys_process(512, 'smss.exe', 2, active).
sys_process(600, 'csrss.exe', 12, active).
sys_process(700, 'lsass.exe', 9, active).
sys_process(4120, 'AppServer.exe', 10, active).
sys_process(5200, 'Updater.exe', 0, exited).
sys_process(5300, 'Monitor.exe', 6, frozen).

% processor(Cpu, PID): the process whose thread
% was running on Cpu, or idle.
processor(0, 4120).
processor(1, 4120).
processor(2, 700).
processor(3, 4).

% zombie_processes(PID): a process that has exited
% but whose process object remains.
zombie_processes(PID) :-
    sys_process(PID, _, 0, exited).

% special_name(Name): a process on which the
% system depends.
special_name('System').
special_name('smss.exe').
special_name('csrss.exe').
special_name('lsass.exe').
special_name('services.exe').
special_name('wininit.exe').

% special_process(PID): a process of the system.
special_process(PID) :-
    sys_process(PID, Name, _, _),
    special_name(Name).

% frozen_process(PID): a process whose threads are
% all frozen.
frozen_process(PID) :-
    sys_process(PID, _, Threads, frozen),
    Threads > 0.
```

```
% busy_system: no processor was idle.
busy_system :-
    \+ processor(_, idle).

?- zombie_processes(P), sys_process(P, N, _, _).
P = 5200,
N = 'Updater.exe'.

?- special_process(P), sys_process(P, N, _, _).
P = 4,
N = 'System' ;
P = 512,
N = 'smss.exe' ;
P = 600,
N = 'csrss.exe' ;
P = 700,
N = 'lsass.exe' ;
false.

?- frozen_process(P).
P = 5300.

?- busy_system.
true.
```

Zombie Processes are processes that have exited but whose process objects remain because something still holds a handle to them; they show in the process list with no threads, and many of them say that some process is leaking process handles. **Special Process** names the processes the system cannot run without, whose failure is a system failure, and the list is the analyst's. **Frozen Process** names a process all of whose threads are frozen, which a debugger does when it attaches and a dump tool does while it writes, so that a frozen process in a dump is usually the dump's own effect and not the problem. **Busy System** names a system all of whose processors were running threads when the dump was written, as against a system that was idle and hung.

The four predicates are tiny because the facts are. A real process list has hundreds of entries with many more properties, and the patterns of this book's Part III that concern a process, from **Spiking Thread** (page 335) to Insufficient Memory, each acquire a process argument when the artifact has processes; the `frame/6` of Chapter 15 is the shape. What a system dump adds that no process dump has is the relation between processes, which is where **Lateral Damage** (page 446) lives.

22.19 Lateral Damage

The pattern names harm done to one component by the failure of another that shares a process, a system, or a resource with it: a plug-in that corrupts the heap of its host, a service in a shared host process that takes the other services down with it, a hook that fails and breaks every caller of the function it hooked. The shared dump has the shape of it. The hooking module of Section 22.6 sits in the path of every `CreateFileW` and `ReadFile` call in the process, so that a fault in it would appear in the stack trace of whichever thread happened to open a file, under the application's frames; and the double free of Chapter 21 by one thread damaged the heap that another thread's copy read from.

Neither is recognized by a predicate here, because the pattern is a claim about causation across components, and the predicates of this book recognize states, not causes. What they can do is list the shared things, the hooked functions, the shared heaps, the sections, so that the analyst knows where lateral damage could have traveled.

22.20 Ubiquitous Component

The pattern names a component that is present nearly everywhere in an artifact: a module on the stack trace of almost every thread, or, in a complete memory dump, loaded into almost every process. A ubiquitous component is either the system's, and uninteresting, or a third party's, and interesting precisely because it is in a position to affect everything. The recognizer measures the share of threads on whose stack trace a module appears against a threshold:

```
% on_stack_of(Module, TID): Module has a frame on
% the stack trace of TID.
on_stack_of(Module, TID) :-
    distinct(TID, frame(TID, _, Module, _, _)).

% ubiquitous_component(Module): Module is on the
% stack traces of at least the ubiquity share of
% the threads.
ubiquitous_component(Module) :-
    module(Module, _, _),
    aggregate_all(count, thread(_, _), Threads),
    aggregate_all(count, on_stack_of(Module, _),
                  OnStack),
    threshold(ubiquity, Share),
    OnStack / Threads >= Share.

% thread_count(Module, N): Module is on N stack
% traces.
thread_count(Module, N) :-
    module(Module, _, _),
    aggregate_all(count, on_stack_of(Module, _),
                  N).

% print_ubiquity: every module with the number of
% threads it appears on, most first.
print_ubiquity :-
    forall(order_by([desc(N)],
                    thread_count(M, N)),
           format("~a~t~17|~t~d~3+~n", [M, N])).
```

```
?- print_ubiquity.
ntdll             10
kernel32          10
appserver          8
ucrtbase           6
dbclient           3
kernelbase         2
reportengine       2
cachelib           1
msvcp140           0
thirdparty_hook    0
true.

?- ubiquitous_component(M).
M = ntdll ;
M = kernel32 ;
M = appserver ;
false.

?- ubiquitous_component(M), \+ system_module(M).
M = appserver ;
false.
```

Three modules are on at least seven of the ten stack traces: the two system modules that every thread starts and waits in, and the application itself, which is ubiquitous because it is the application. The table says the rest: the C runtime on six threads, the database client on three, and two modules on no stack at all, among them the hooking module, which is ubiquitous in a sense the stack traces do not measure, since it stands in the path of every file operation without appearing on any stack unless a thread was inside its hook when the dump was written.

The share is over threads, which is the dimension a process dump has. The catalog's pattern is at home in complete memory dumps, where the count is over processes and a third-party module in every process is the finding; the `sys_process/4` extension of Section 22.18 would need a module list per process for that, and the predicate would change its first `aggregate_all/3` and nothing else. The threshold is the analyst's, and the system modules have to be read past: a version of the predicate that excludes `system_module/1` reports only the application, as the last query does, and a dump in which it reported anything else would be worth the analyst's attention.

Chapter 23. Dump Quality, Composition, and Reports

Every pattern of Part III has been recognized in a dump taken to be a faithful record of a process at one moment. The first patterns of this chapter question that assumption. A dump may have been written on request rather than by a fault, may hold less than the process had, may contradict itself, may have been taken before the process did anything, may record an exception that was never a failure, or may not load at all. Each of these changes how every other finding is to be read, and a report that lists findings without saying what kind of dump they come from is incomplete.

The chapter then turns from single patterns to their composition. Patterns entail one another: a **Deadlock (Critical Sections)** (page 388) is made of **Blocked Thread (Software)** (page 336) instances, each of which heads a **Wait Chain (Critical Sections)** (page 384). Chapter 17 said that a pattern is a goal in another pattern's body; this chapter composes the recognizers of Chapter 20 as that chapter wrote them and shows the entailment as a query. **Step Dumps** (page 460) follows, comparing the three snapshots of Chapter 21. The chapter ends with `report/0`, a predicate that prints a plain-text analysis report of `dump1.pl`: every pattern instance it recognizes, in order of a severity fact, the instances it suspects, kept apart, the patterns it looked for and did not find, and explicit statements of what the report does not claim.

Each chapter starts a fresh toplevel, and the recognizers of Chapters 18 to 22 are therefore not restated here. They are collected, with the clauses those chapters gave them, in the module `patterns_dump`, the file `patterns_dump.pl` of the shared artifacts, which the chapter loads with `use_module/1` after the dump. The module does not consult `dump1.pl` itself: a file without a module declaration is loaded into one module only, as Chapter 13 said, and the chapters load the dump into `user`, so every reference to a relation of the schema inside the module is qualified with `user:`, and the module's predicates read whatever dump the toplevel has loaded. The thresholds of Chapters 18 to 21 and the analyst's list of system modules are facts of the module, exported, and declared `multifile` and `dynamic`, so that a chapter adds a limit of its own with a module-qualified clause, as the first program block does for the one threshold this chapter needs, and an analysis changes a value with `retract/1` and `assertz/1`. The extension relations that Chapters 20 and 21 define in-chapter, `mutant/3`, `waits_for_object/3`, `heap_at/5`, and `dump_time/2`, are declared dynamic in `user` by the module, so that a knowledge base without them makes the recognizers that read them fail rather than raise an existence error; Section 23.9 supplies two of them.

The dump's own properties, which a debugger reports when it opens the file, are not in the schema of Chapter 15 and are added here as `dump_info/2`, an extension of the schema.

```
:- [dump1].
:- use_module(patterns_dump).
```

```
% pattern(Name, Artifact, Chapter): the catalog
% entries treated in this chapter.
pattern('Manual Dump (Process)', memory_dump, 23).
pattern('Truncated Dump', memory_dump, 23).
pattern('Inconsistent Dump', memory_dump, 23).
pattern('Early Crash Dump', memory_dump, 23).
pattern('False Positive Dump', memory_dump, 23).
pattern('Corrupt Dump', memory_dump, 23).
pattern('No Current Thread', memory_dump, 23).
pattern('Step Dumps', memory_dump, 23).

% dump_info(Key, Value): what the debugger reports
% about the dump file itself. An extension of the
% schema.
dump_info(kind, process).
dump_info(truncated, false).
dump_info(current_thread, 109).
dump_info(process_uptime_s, 1860).

% threshold(Name, Value): the module's thresholds
% are those of Chapters 18 to 21; this chapter
% adds its one limit to the multifile predicate.
patterns_dump:threshold(early_uptime_s, 10).
```

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- dump_info(kind, K), dump_info(truncated, T).
K = process,
T = false.

?- predicate_property(spiking_thread(_),
                      imported_from(M)).
M = patterns_dump.

?- aggregate_all(count, threshold(_, _), N),
   threshold(early_uptime_s, S).
N = 12,
S = 10.

?- threshold(spiking_share, Min),
   threshold(frame_run, Run),
   threshold(overflow_run, Overflow).
Min = 25,
Run = 4,
Overflow = 6.
```

The module's exports are visible in `user` as if defined there, which `predicate_property/2` with `imported_from(M)` shows for `spiking_thread/1`, and the twelve thresholds are the eleven of the pattern chapters and the one added here. The three shown are Chapters 18's and 19's: a **Spiking Thread** (page 335) has at least 25 percent of the process's processor time, a **Frame Regularity** (page 342) is a run of at least four frames, and a deep stack is a suspected **Stack Overflow (User Mode)** (page 366) when a run reaches six, Chapter 19's own policy, which is kept under its own name so that the two chapters' rules stay distinct. Whatever this chapter's report says about those patterns, it says with those chapters' definitions and limits.

23.1 Manual Dump (Process)

The pattern names a process memory dump written because someone asked for it, with a tool or a debugger command, rather than because the process faulted. It is the normal kind of dump for a hang, and the analyst must know it is one: a manual dump has no faulting thread to start from, and the thread the debugger shows first is whichever it broke in on, not one that did anything wrong. Three signs are read from the dump. No thread has an exception record. The main thread, whose stack trace holds the process's entry point, is not faulting. And one thread may carry the frames of the debugger's break-in, `DbgBreakPoint` or `DbgUiRemoteBreakin`, which is how a debugger stopped the process to write the file.

```
% break_in_frame(TID): a frame left by a debugger
% break-in on the stack trace of TID.
break_in_frame(TID) :-
    frame(TID, _, _, Function, _),
    memberchk(Function, [ 'DbgBreakPoint',
                          'DbgUiRemoteBreakin' ]).

% manual_dump(Sign): the dump was written on
% request: no thread has an exception record, or a
% debugger broke in.
manual_dump(no_exception_record) :-
    \+ exception_record(_, _, _).
manual_dump(break_in(TID)) :-
    break_in_frame(TID).

?- main_thread(T).
T = 100 ;
false.

?- main_thread(T), \+ exception_record(T, _, _).
T = 100 ;
false.

?- manual_dump(Sign).
false.

?- exception_record(T, Code, _),
   format("~16r~n", [Code]).
c0000005
T = 109,
Code = 3221225477.
```

The main thread is not faulting, but the dump is not a manual dump: thread 109 has an exception record with the access violation code, and no thread shows a break-in. The dump was written for the exception, by a postmortem debugger or an error reporting rule, and the faulting thread is where analysis starts. `main_thread/1` is Chapter 18's, imported from the module, with its list of entry functions; the `;` after its answer is `thread/2` offering the other nine threads, none of which qualifies, and the `false.` is their failure. `memberchk/2` in `break_in_frame/1` tests membership without leaving a choice point.

The predicate recognizes the signs the stack traces give, and the signs can mislead in both directions. A manual dump of a process that had a first-chance exception in flight has an exception record and is a manual dump nonetheless; a dump written by a tool that injects no thread shows no break-in frames and is manual nonetheless, so `no_exception_record` is the sign that matters and a `break_in(TID)` sign is confirmation when it occurs. Conversely, the dump kind that the debugger reports when it opens the file, which a richer `dump_info/2` would record as the comment the dump writer left in it, is the direct evidence, and the analyst reads that first when it exists.

23.2 Truncated Dump

The pattern names a dump that holds less than the process had: a dump cut short by a disk filling up, a minidump written without the memory the stack traces need, a module list missing the modules that frames refer to. The debugger reports truncation of the file when it can tell; the content shows it when the file is whole but the capture was not. Both are tested:

```
% bottom_frame(TID, Function): the function of the
% deepest frame of TID.
bottom_frame(TID, Function) :-
    thread(TID, _),
    aggregate_all(max(N, F),
                  frame(TID, N, _, F, _),
                  max(_, Function)).

% truncation_sign(Sign): one symptom of a dump
% that holds less than the process had.
truncation_sign(flagged) :-
    dump_info(truncated, true).
truncation_sign(module_missing(Module)) :-
    distinct(Module, frame(_, _, Module, _, _)),
    \+ module(Module, _, _).
truncation_sign(no_frames(TID)) :-
    thread(TID, _),
    \+ frame(TID, _, _, _, _).
truncation_sign(incomplete_stack(TID)) :-
    bottom_frame(TID, Function),
    Function \== 'RtlUserThreadStart'.

% truncated_dump(Signs): the dump shows at least
% one sign of truncation.
truncated_dump(Signs) :-
    findall(Sign, truncation_sign(Sign), Signs),
    Signs \== [].

?- truncated_dump(Signs).
false.

?- forall(thread(T, _),
          bottom_frame(T, 'RtlUserThreadStart')).
true.

?- forall(frame(_, _, M, _, _), module(M, _, _)).
true.
```

The dump shows no sign of truncation: the debugger did not flag the file, every module a frame names is in the module list, every thread has frames, and every stack trace ends in the system's thread start function. The predicate collects the signs with `findall/3` and succeeds with the list when it is nonempty, which is the form a report wants, since the signs say where the dump is incomplete and not only that it is; `truncation_sign/1` alone is the enumerator form.

The signs are symptoms, and each has other causes. A stack trace that ends before `RtlUserThreadStart` is also the **Truncated Stack Trace** (page 346) of Chapter 18, where the cause is a stack walk that failed in a frame without unwind information and the dump is whole; a module that frames name and the list lacks is also what a module unloaded before the dump looks like, when the debugger still knows its name from its list of unloaded modules. The predicate cannot tell these apart from the facts, and it names the sign rather than the cause for that reason. What it also cannot see is what is absent altogether: a minidump that omits the heap gives the schema no `heap/4` facts at all, and nothing in the knowledge base says whether the process had heaps. The census of Section 23.6 is the closest a program comes to noticing.

23.3 Inconsistent Dump

The pattern names a dump whose parts do not agree, usually because the process kept running while the dump was written and different parts of the file describe different moments: a critical section owned by a thread that is not in the thread list, a thread waiting for a lock that is free, a thread state that says exception and no exception record. The schema makes such disagreements checkable, since each is a relation between two kinds of fact, and the checks are written as one clause each of `inconsistency/1`:

```
% inconsistency(Finding): one disagreement between
% two kinds of fact in the dump.
inconsistency(owner_not_a_thread(Addr, Owner)) :-
    critical_section(Addr, Owner, _, _),
    Owner =\= 0,
    \+ thread(Owner, _).
inconsistency(wait_on_unknown_section(TID)) :-
    waits_for(TID, critical_section, Addr),
    \+ critical_section(Addr, _, _, _).
inconsistency(wait_on_free_section(TID)) :-
    waits_for(TID, critical_section, Addr),
    critical_section(Addr, 0, _, _).
inconsistency(state_without_record(TID)) :-
    thread_state(TID, waiting(exception)),
    \+ exception_record(TID, _, _).
inconsistency(record_without_state(TID)) :-
    exception_record(TID, _, _),
    \+ thread_state(TID, waiting(exception)).
inconsistency(fault_outside_top_module(TID)) :-
    exception_record(TID, _, Addr),
    frame(TID, 0, Module, _, _),
    \+ ( module(Module, Base, Size),
         Base =< Addr,
```

```
          Addr < Base + Size
        ).

% inconsistent_dump(Findings): the dump disagrees
% with itself in at least one way.
inconsistent_dump(Findings) :-
    findall(F, inconsistency(F), Findings),
    Findings \== [].

?- inconsistent_dump(F).
false.

?- exception_record(109, _, A),
   frame(109, 0, M, _, _),
   module(M, Base, Size),
   Base =< A, A < Base + Size.
A = 140718380167594,
M = dbclient,
Base = 140718380154880,
Size = 450560.

?- critical_section(Addr, Owner, _, _),
   Owner =\= 0, thread(Owner, _),
   format("~16r~n", [Addr]).
7ff61d13a0c0
Addr = 140695026507968,
Owner = 103 ;
7ff61d13a140
Addr = 140695026508096,
Owner = 104 ;
false.
```

The dump passes the six checks. The second query is the last check run forward: the exception address lies inside the module of the faulting thread’s top frame, which is what a consistent dump shows, and the third lists the owned critical sections with owners that are threads of the dump. Every check is a negation or a comparison over bound facts, and the order of subgoals in each clause binds the variables before `\+` or `=\=` reaches them, which is the safety condition of Chapter 7.

Six checks are the ones the shared schema supports, and a schema with more kinds of fact supports more; the general shape is that every foreign key of the schema, every thread identifier, address, or module name that one fact uses to refer to another, is a check. What the checks do not establish is consistency. A dump that passes them can still describe two moments, if the two moments happen to agree on everything the schema records, and a dump that fails one may be consistent and the schema’s pipeline wrong. The finding is a disagreement between facts; whether the dump or the facts are at fault is the analyst’s to decide, and the catalog’s pattern is about the dump.

23.4 Early Crash Dump

The pattern names a dump of a process that failed before it had done anything: during loading, in a static initializer, in the first lines of its entry point. Such a dump has few threads, few modules, a main thread whose deepest frame is in the loader rather than in the thread start function, and a

process age of seconds. The age is reported by the debugger and kept here in `dump_info/2`; the rest is in the facts:

```
% early_sign(Sign): a symptom that the process had
% barely started when the dump was written.
early_sign(short_uptime(Seconds)) :-
    dump_info(process_uptime_s, Seconds),
    threshold(early_uptime_s, Max),
    Seconds < Max.
early_sign(single_thread) :-
    aggregate_all(count, thread(_, _), 1).
early_sign(thread_in_loader(TID)) :-
    bottom_frame(TID, 'LdrInitializeThunk').

% early_crash_dump(Signs): the dump shows at least
% one sign of an early crash.
early_crash_dump(Signs) :-
    findall(Sign, early_sign(Sign), Signs),
    Signs \== [].


?- early_crash_dump(S).
false.

?- dump_info(process_uptime_s, S), M is S // 60.
S = 1860,
M = 31.

?- main_thread(T), frame(T, 2, appserver, F, _).
T = 100,
F = 'MainLoop' ;
false.
```

The process had run for 31 minutes with ten threads, and its main thread was in its message loop; nothing about the dump is early. The second clause of `early_sign/1` unifies the count of threads with 1 in the result argument of `aggregate_all/3`, which is a test, and fails for ten; the third asks for a thread whose deepest frame is the loader's initialization entry, which a thread shows while the process is still being set up and before its own code has run.

The signs are the ones a user-space dump shows, and the uptime is the strongest. A process that crashes in a static initializer has threads created by the modules loaded so far and modules loaded by its import table, so neither count is reliably small, and the loader frame is only on the stack of the thread doing the initializing. The pattern's value is in what it rules out: a process that failed in its first seconds did not fail because of a leak, a race under load, or a resource exhausted over hours, and the recognizers for those are not worth running on it.

23.5 False Positive Dump

The pattern names a dump written for an exception that was not a failure: a breakpoint the debugger itself raised, a single-step trap, the exception a program raises to give a thread a name, a debug print. A tool configured to write a dump on every exception writes one for each of these, and the

analyst who treats the exception record as a fault analyzes nothing. The codes are in the exception record, and the recognizer compares them with a list of codes that a process raises and survives:

```
% benign_code(Code, Kind): exception codes that a
% process raises and survives when a debugger is
% attached.
benign_code(0x80000003, breakpoint).
benign_code(0x80000004, single_step).
benign_code(0x406d1388, thread_naming).
benign_code(0x40010006, debug_print).

% false_positive_dump(TID, Kind): the exception
% record of TID is of a benign kind and no other
% thread has a record that is not.
false_positive_dump(TID, Kind) :-
    exception_record(TID, Code, _),
    benign_code(Code, Kind),
    \+ ( exception_record(Other, OtherCode, _),
         Other \== TID,
         \+ benign_code(OtherCode, _)
       ).

?- false_positive_dump(T, K).
false.

?- exception_record(_, Code, _),
   Severity is Code >> 30.
Code = 3221225477,
Severity = 3.

?- benign_code(Code, _), Severity is Code >> 30,
   format("~16r ~d~n", [Code, Severity]), fail.
80000003 2
80000004 2
406d1388 1
40010006 1
false.
```

The one exception record has the access violation code, whose severity bits are 3, an error; the benign codes have severity 2, a warning, or 1, informational, which is the classification Chapter 7 made of status codes and the first check an analyst makes on an exception record. The negation in `false_positive_dump/2` guards against a dump with a benign record on one thread and a real one on another, which is not a false positive but a dump with two exception records, the **Nested Exceptions (User Space)** (page 372) or **Multiple Exceptions (User Space)** (page 373) of Chapter 19.

The list is the analyst's and is short. The severity bits are the general test, since every code with severity below 3 is at least suspect as the reason for a dump, but a warning code can be a failure in context: a C++ exception, `0xe06d7363`, has severity 3 and is often handled, and a breakpoint reached by a corrupted jump is not benign. The predicate names the code's kind and leaves the judgment where the catalog leaves it.

23.6 Corrupt Dump

The pattern names a dump file that the debugger cannot read, in whole or in part: a file truncated in the header, a file whose streams point outside it, a file copied badly. A corrupt dump produces no facts, or produces facts up to the point of corruption, and the pipeline of Chapter 15 reports the debugger's errors rather than a knowledge base. The pattern therefore has no recognizer over the schema: by the time there are facts to query, the dump was readable. What a program can do is notice a knowledge base that is implausibly thin, which is the Prolog-side symptom of a dump that loaded partially. A census of the schema's predicates is the check:

```
% census(Counts): the number of facts of each
% predicate of the schema.
census(Counts) :-
    findall(Name/Arity-N,
            ( schema_predicate(Name/Arity),
              functor(Goal, Name, Arity),
              aggregate_all(count, Goal, N)
            ),
            Counts).

schema_predicate(process/2).
schema_predicate(module/3).
schema_predicate(module_info/3).
schema_predicate(thread/2).
schema_predicate(thread_time/3).
schema_predicate(thread_state/2).
schema_predicate(frame/5).
schema_predicate(exception_record/3).
schema_predicate(exception_info/3).
schema_predicate(critical_section/4).
schema_predicate(waits_for/3).
schema_predicate(handle_count/3).
schema_predicate(heap/4).
schema_predicate(last_error/2).
schema_predicate(hooked/3).

% print_census: one line per schema predicate.
print_census :-
    census(Counts),
    forall(member(Pred-N, Counts),
           format("~w~t~20|~t~d~4+~n",
                  [Pred, N])).

?- print_census.
process/2              1
module/3              10
module_info/3         20
thread/2              10
thread_time/3         10
thread_state/2        10
frame/5               78
exception_record/3     1
exception_info/3       2
critical_section/4     3
waits_for/3            3
```

```
handle_count/3          6
heap/4                  3
last_error/2            6
hooked/3                2
true.

?- census(_C), member(P-0, _C).
false.
```

Every predicate of the schema has facts, and the proportions are those of a whole process: as many `thread_time/3` and `thread_state/2` facts as threads, frames for every thread, module information for every module. `functor/3` builds a goal with the predicate's name and arity and unbound arguments, and `aggregate_all/3` counts its solutions; the second query asks whether any predicate has none, which for a dump of a process with heaps and handles would be the sign to look at. The check is weak, since a dump with no exception has one empty predicate legitimately, and it is a check on the pipeline as much as on the dump.

23.7 No Current Thread

The pattern names a dump in which the debugger has no thread to call current: no exception context to switch to, no record of which thread the dump writer was on, so that commands that assume a thread, `k` among them, have nothing to show until the analyst chooses one. It is a property of the dump file rather than of the process, and `dump_info/2` records what the debugger reported:

```
% no_current_thread: the debugger reported no
% current thread when it opened the dump.
no_current_thread :-
    dump_info(current_thread, none).

?- no_current_thread.
false.

?- dump_info(current_thread, T),
   exception_record(T, _, _).
T = 109.
```

The current thread is the faulting one, as it is in a dump written for an exception. A manual dump of Section 23.1 has a current thread too, the one the dump writer used, and the pattern arises with dump formats and tools that record none. Its consequence for a knowledge base is nil, since the facts do not depend on which thread is current, and its consequence for the analyst is procedural: the stack trace collection of Chapter 18 is where to begin, since no single thread presents itself.

23.8 Deadlock (Critical Sections), Blocked Thread (Software), and Wait Chain (Critical Sections)

Chapter 20 treats the three synchronization patterns in full. They return here because they compose, and the composition is the chapter's example of what Chapter 17 called a pattern as a goal in another

pattern's body. A **Blocked Thread (Software)** (page 336) is a thread stopped by a synchronization object that software controls. A **Wait Chain (Critical Sections)** (page 384) is the sequence of threads from a blocked thread through the owner of the section it waits for, and the owner of what that thread waits for, until a thread that waits for nothing or a thread already in the chain. A **Deadlock (Critical Sections)** (page 388) is a wait chain that returns to its start. The three recognizers are Chapter 18's `blocked_thread/1` and Chapter 20's `wait_chain/2` and `deadlock/2`, imported from the module with the clauses those chapters gave them; the first argument of the last two is the kind of synchronization object, `critical_section` here, since the shared dump records waits of no other kind.

```
?- deadlock(critical_section, C).
C = [103, 104] ;
false.

?- wait_chain(critical_section, Chain).
Chain = [101, 103, 104] ;
Chain = [103, 104] ;
Chain = [104, 103] ;
false.

?- deadlock(critical_section, C),
   forall(member(T, C), blocked_thread(T)).
C = [103, 104] ;
false.

?- deadlock(critical_section, C), member(T, C),
   wait_chain(critical_section, [T | Rest]),
   last([T | Rest], Last),
   wait_edge(critical_section, Last, _Back),
   memberchk(_Back, [T | Rest]).
C = [103, 104],
T = 103,
Rest = [104],
Last = 104 ;
C = [103, 104],
T = 104,
Rest = [103],
Last = 103 ;
false.
```

The third and fourth queries are the composition as a check. Every member of the deadlock cycle is a blocked thread, and every member heads a wait chain that closes: Chapter 20's chain stops before a thread it already holds, so a chain closes when the wait edge of its last thread, `wait_edge/3` of that chapter, leads back to a thread in the chain. That is what the entailments say: a deadlock implies blocked threads, and a blocked thread in a deadlock implies a wait chain that closes. The entailment runs one way. Thread 101 is blocked and heads a wait chain that reaches the cycle, and is not in it; its chain is longer than the cycle because the thread waits behind the deadlock rather than in it, which is the difference between a victim and a participant. `deadlock/2` gives each cycle once, from its smallest member, as Chapter 20 explained; without that device `[104, 103]` would be a second answer.

The explainer of Chapter 20 is imported as well, and the contract of Chapter 17 holds for it here as there: the evidence of the cycle is four facts of the dump, each callable, and they alone entail the cycle.

```
?- deadlock(critical_section, C,
              evidence(Name, _, _Fs)),
   ground(_Fs), maplist(call, _Fs),
   length(_Fs, N).
C = [103, 104],
Name = 'Deadlock (Critical Sections)',
N = 4 ;
false.
```

The entailments can be stated as facts, and the report of Section 23.10 prints them beside the patterns they relate:

```
% implies(Pattern, Pattern2): every instance of
% Pattern contains instances of Pattern2.
implies('Deadlock (Critical Sections)',
        'Blocked Thread (Software)').
implies('Blocked Thread (Software)',
        'Wait Chain (Critical Sections)').
```

The predicates are Chapter 20's, and not a restatement: the kind argument selects the edges, the visited list bounds the walk, and `min_list/2` reports each cycle once, exactly as there. The module declares that chapter's extension relations, `mutant/3` and `waits_for_object/3`, dynamic in `user`, so that in this toplevel, which does not define them, `deadlock(any, C)` finds the one cycle of the shared dump and raises no error. The epistemic limit is the one every synchronization pattern has: `waits_for/3` and `critical_section/4` are the debugger's reading of the process's lock structures at the moment of the dump, and a dump that is inconsistent in the sense of Section 23.3 can show a cycle that never existed in any single moment. The checks of that section are the reason to trust this one.

23.9 Step Dumps

The pattern names a sequence of dumps of the same process taken at intervals and read as a sequence, so that the information is in the differences: a thread whose stack trace is the same in every dump is stuck, a heap that grows in every step is leaking, a count that rises and falls is a workload. Chapter 21 introduced three snapshots of the heaps of this process and read a leak and a fluctuation from them; the facts are repeated here, in the two extension relations the module declares and Chapter 21's `memory_leak/1` reads, and the recognizer computes the steps themselves, the change of committed memory from each dump to the next:

```
% heap_at(Dump, Heap, ReservedKb, CommittedKb,
% FreeKb) and dump_time(Dump, Seconds): the series
% of Chapter 21. An extension of the schema.
heap_at(dump1a, 0x0000_0210_a000_0000,
        2097152, 1703936, 40960).
```

```
heap_at(dump1a, 0x0000_0210_a001_0000,
        1024, 256, 128).
heap_at(dump1a, 0x0000_0210_a200_0000,
        65536, 59392, 2048).
heap_at(dump1b, 0x0000_0210_a000_0000,
        2097152, 1769472, 32768).
heap_at(dump1b, 0x0000_0210_a001_0000,
        1024, 256, 128).
heap_at(dump1b, 0x0000_0210_a200_0000,
        65536, 32768, 4096).
heap_at(dump1, Heap, R, C, F) :-
    heap(Heap, R, C, F).

dump_time(dump1a, 0).
dump_time(dump1b, 600).
dump_time(dump1, 1200).

% step_dumps(Heap, Steps): Steps lists, for each
% dump after the first, Time-Delta where Delta is
% the change of committed KB of Heap since the
% previous dump.
step_dumps(Heap, Steps) :-
    heap(Heap, _, _, _),
    findall(T-C,
            ( heap_at(D, Heap, _, C, _),
              dump_time(D, T)
            ),
            Pairs),
    keysort(Pairs, [_-C0 | Series]),
    steps_from(Series, C0, Steps).

steps_from([], _, []).
steps_from([T-C | Series], Prev, [T-D | Steps]) :-
    D is C - Prev,
    steps_from(Series, C, Steps).

% dump_sequence(Dumps): the dumps in time order.
dump_sequence(Dumps) :-
    findall(T-D, dump_time(D, T), Pairs),
    keysort(Pairs, Sorted),
    pairs_values(Sorted, Dumps).

?- dump_sequence(Ds).
Ds = [dump1a, dump1b, dump1].

?- forall(step_dumps(H, Steps),
          format("~16r ~w~n", [H, Steps])).
210a0000000 [600-65536,1200-65536]
210a0010000 [600-0,1200-0]
210a2000000 [600- -26624,1200-28672]
true.

?- memory_leak(H), format("~16r~n", [H]).
210a0000000
H = 2270427086848 ;
false.
```

```
?- step_dumps(H, Steps),
   forall(member(_-D, Steps), D > 0),
   memory_leak(H).
H = 2270427086848,
Steps = [600-65536, 1200-65536] ;
false.
```

The first heap gained 64 MB in each step, the second did not change, and the third lost 26 MB and regained 28, which are the leak and the fluctuation of Chapter 21 as differences rather than as series. `600- -26624` is how SWI-Prolog writes the pair of 600 and a negative number: the space separates the operator from the sign, which is needed for the term to be read back, and the term is the same `-/2` compound as the others. `pairs_values/2` from `library(pairs)` takes the dump identifiers from the sorted pairs. The leak is recognized by Chapter 21's `memory_leak/1`, imported from the module and reading the `heap_at/5` and `dump_time/2` facts above, which asks for a strictly increasing series; the last query is the same condition stated over the differences, every step a rise, and the two agree on the one heap, as a statement about a series and a statement about its differences must.

The pattern is wider than heaps. Step dumps of a hang compare stack traces across the steps, and a thread whose `frame/5` facts are identical in every dump is the finding; the schema would need `frame_at/6` with a dump identifier in front, which is the `heap_at/5` construction applied to frames, and the comparison is `stack_trace/2` on each dump and `==/2` between the results. The shared data has no such series, and the chapter leaves the shape stated. What step dumps cannot show is what happened between the steps; a thread that is in the same function in two dumps may have left and returned, and the interval is the analyst's choice of how much to miss.

23.10 The Analysis Report

A report is the analyst's output, and a report produced by a program is a list of the pattern instances the program recognized, in an order that puts first what matters most, with the instances it can only suspect kept apart, with the patterns it looked for and did not find, and with a statement of its own limits. `report/0` is such a program for `dump1.pl`. It composes the canonical recognizers of Chapters 18 to 22, imported from `patterns_dump` at the start of the chapter with the clauses, helpers, and thresholds those chapters gave them, so that every instance it prints is an instance by the definition of the chapter that treats the pattern, and the recognizer a reader disputes is the one that chapter defined and tested. Nothing is shortened for the shared dump: the recognizers are general, and the report is a composition of them, not a library of its own.

```
?- spiking_thread(T).
T = 102 ;
T = 107 ;
false.

?- frame_regularity(T, Key, N).
T = 107,
Key = reportengine-'Node::Visit',
N = 8 ;
```

```
false.

?- null_pointer_data(T), not_my_version(M).
T = 109,
M = dbclient ;
false.

?- stack_overflow(T, _Finding),
   functor(_Finding, Form, 3).
T = 107,
Form = suspected ;
false.
```

The recognizers the report composes are **Spiking Thread** (page 335), **Blocked Thread (Software)** (page 336), **Frame Regularity** (page 342), and **Main Thread** (page 332) from Chapter 18; **Exception Stack Trace** (page 358), **NULL Pointer (Data)** (page 360), and **Stack Overflow (User Mode)** (page 366) from Chapter 19; **Wait Chain (Critical Sections)** (page 384) and **Deadlock (Critical Sections)** (page 388) from Chapter 20; **Handle Leak** (page 409), **Insufficient Memory (Committed Memory)** (page 406), **Insufficient Memory (Handle Leak)** (page 410), and **Memory Leak (Process Heap)** (page 411) from Chapter 21; and **Not My Version** (page 429), **Unknown Component** (page 430), **No Component Symbols** (page 431), **Hooked Functions (User Space)** (page 433), and **Last Error Collection** (page 443) from Chapter 22, with Suspended Thread as Chapter 1 wrote it. Each is called with the arguments its chapter gave it: the key of a **Frame Regularity** (page 342) is a `Module-Function` pair, the kind of a wait chain is its first argument, and the finding of `stack_overflow/2` on the shared dump is a suspicion term, since no thread has the status code. The dump quality patterns of Sections 23.1 to 23.7 and **Step Dumps** (page 460) are this chapter's own.

The catalog of the report is `instance/2`: one clause per pattern, relating the pattern's name to each of its instances, with the instance as a term whose principal functor says what kind of thing it is. `suspicion/2` does the same for the instances a recognizer reports with a suspicion term, and the two are never merged, as Section 17.3 requires. `severity/2` ranks the patterns, and the rank names are the report's vocabulary for them. The dump quality patterns of Sections 23.1 to 23.5 rank first, because a positive instance of any of them qualifies everything below it.

```
% instance(Pattern, Instance): a recognized
% instance of Pattern in the dump.
instance('Manual Dump (Process)', Sign) :-
    manual_dump(Sign).
instance('Truncated Dump', signs(Signs)) :-
    truncated_dump(Signs).
instance('Inconsistent Dump', findings(Fs)) :-
    inconsistent_dump(Fs).
instance('Early Crash Dump', signs(Signs)) :-
    early_crash_dump(Signs).
instance('False Positive Dump',
         thread(TID, Kind)) :-
    false_positive_dump(TID, Kind).
instance('Deadlock (Critical Sections)',
         threads(Cycle)) :-
    deadlock(critical_section, Cycle).
instance('NULL Pointer (Data)', thread(TID)) :-
```

```
    null_pointer_data(TID).
instance('Stack Overflow (User Mode)',
         thread(TID)) :-
    stack_overflow(TID, evidence(_, TID, _)).
instance('Exception Stack Trace', thread(TID)) :-
    exception_stack_trace(TID, _).
instance('Insufficient Memory (Committed Memory)',
         Inst) :-
    insufficient_memory(committed_memory, Inst).
instance('Insufficient Memory (Handle Leak)',
         Inst) :-
    insufficient_memory(handle_leak, Inst).
instance('Memory Leak (Process Heap)', heap(H)) :-
    memory_leak(H).
instance('Handle Leak', handles(Type)) :-
    handle_leak(Type).
instance('Spiking Thread', thread(TID)) :-
    spiking_thread(TID).
instance('Frame Regularity',
         frames(TID, Key, N)) :-
    frame_regularity(TID, Key, N).
instance('Blocked Thread (Software)',
         thread(TID)) :-
    blocked_thread(TID).
instance('Wait Chain (Critical Sections)',
         chain(Chain)) :-
    wait_chain(critical_section, Chain).
instance('Not My Version', module(M)) :-
    not_my_version(M).
instance('Hooked Functions (User Space)',
         hook(F, M, H)) :-
    hooked_functions(M, H, Fs),
    member(F, Fs).
instance('Unknown Component', module(M)) :-
    unknown_component(M).
instance('No Component Symbols', module(M)) :-
    no_component_symbols(M).
instance('Suspended Thread', thread(TID)) :-
    suspended_thread(TID).
instance('Last Error Collection',
         error(TID, C)) :-
    last_error_collection(Pairs),
    member(TID-C, Pairs),
    C =\= 0.
instance('Main Thread', thread(TID)) :-
    main_thread(TID).
instance('Step Dumps', dumps(Dumps)) :-
    dump_sequence(Dumps),
    Dumps = [_, _ | _].

% suspicion(Pattern, Instance): an instance of
% Pattern that a recognizer reports with a
% suspicion term rather than an evidence term.
suspicion('Stack Overflow (User Mode)',
          thread(TID)) :-
    stack_overflow(TID, suspected(_, TID, _)).

% severity(Pattern, Rank): the order of the
% report; 0 qualifies the dump, 1 is most severe.
```

```
severity('Manual Dump (Process)', 0).
severity('Truncated Dump', 0).
severity('Inconsistent Dump', 0).
severity('Early Crash Dump', 0).
severity('False Positive Dump', 0).
severity('Deadlock (Critical Sections)', 1).
severity('NULL Pointer (Data)', 1).
severity('Stack Overflow (User Mode)', 1).
severity('Exception Stack Trace', 2).
severity('Insufficient Memory (Committed Memory)',
         2).
severity('Insufficient Memory (Handle Leak)', 2).
severity('Memory Leak (Process Heap)', 2).
severity('Handle Leak', 2).
severity('Spiking Thread', 2).
severity('Frame Regularity', 3).
severity('Blocked Thread (Software)', 3).
severity('Wait Chain (Critical Sections)', 3).
severity('Not My Version', 3).
severity('Hooked Functions (User Space)', 3).
severity('Unknown Component', 3).
severity('No Component Symbols', 4).
severity('Suspended Thread', 4).
severity('Last Error Collection', 4).
severity('Main Thread', 4).
severity('Step Dumps', 4).

% rank_name(Rank, Name): the report's word for a
% severity rank.
rank_name(0, 'dump quality').
rank_name(1, critical).
rank_name(2, major).
rank_name(3, minor).
rank_name(4, informational).

?- aggregate_all(count, severity(_, _), N),
   aggregate_all(count,
                 distinct(P, instance(P, _)), R),
   aggregate_all(count,
                 distinct(P, suspicion(P, _)), S).
N = 25,
R = 19,
S = 1.

?- severity(P, 1), instance(P, I).
P = 'Deadlock (Critical Sections)',
I = threads([103, 104]) ;
P = 'NULL Pointer (Data)',
I = thread(109) ;
false.

?- severity(P, 1), suspicion(P, I).
P = 'Stack Overflow (User Mode)',
I = thread(107) ;
false.
```

Twenty-five patterns are looked for, nineteen have instances, and one, **Stack Overflow (User Mode)** (page 366), has a suspected instance and no recognized one: thread 107 is deep and regular and has

no exception record, so the pattern is looked for, not recognized, and suspected, three statements that the report keeps separate. The instances are terms, and a report prints them as text; `describe/3` relates each kind of instance term to a `format/2` control string and its arguments, with a last resort for kinds it does not know, which is how a new pattern is added without touching the printing code:

```
% describe(Instance, Format, Args): how to print
% an instance of each kind.
describe(thread(T), "thread ~d", [T]).
describe(thread(T, Kind), "thread ~d (~a)",
         [T, Kind]).
describe(threads(Ts), "threads ~w", [Ts]).
describe(chain(Chain), "wait chain ~w", [Chain]).
describe(module(M), "module ~a", [M]).
describe(heap(H), "heap 0x~16r", [H]).
describe(process, "the process", []).
describe(process(PID), "process ~d", [PID]).
describe(handles(Type), "~a handles", [Type]).
describe(hook(F, M, H), "~a!~a <- ~a", [M, F, H]).
describe(frames(T, _-F, N),
         "thread ~d: ~d frames of ~a", [T, N, F]).
describe(error(T, C),
         "thread ~d: last error 0x~16r", [T, C]).
describe(dumps(Ds), "dumps ~w", [Ds]).

% print_instance(Instance): one indented line.
print_instance(Instance) :-
    (   describe(Instance, Format, Args)
    ->  format("    "),
        format(Format, Args),
        nl
    ;   format("    ~w~n", [Instance])
    ).

% recognized(Rank, Pattern): Pattern has at least
% one instance.
recognized(Rank, Pattern) :-
    severity(Pattern, Rank),
    once(instance(Pattern, _)).

% report_pattern(Rank, Pattern): the heading, the
% instances, and the entailments of Pattern.
report_pattern(Rank, Pattern) :-
    rank_name(Rank, Word),
    format("[~a] ~a~n", [Word, Pattern]),
    forall(instance(Pattern, Instance),
           print_instance(Instance)),
    forall(implies(Other, Pattern),
           format("    (entailed by ~a)~n",
                  [Other])).


?- report_pattern(1,
                  'Deadlock (Critical Sections)').
[critical] Deadlock (Critical Sections)
    threads [103,104]
true.
```

```
?- report_pattern(3, 'Blocked Thread (Software)').
[minor] Blocked Thread (Software)
    thread 101
    thread 103
    thread 104
    (entailed by Deadlock (Critical Sections))
true.
```

The remaining parts of the report are the suspected instances, the list of patterns looked for and not recognized, the list of what the report did not examine because the knowledge base lacks the facts, and the statements of what it does not claim. The suspicions are printed under their own heading, since a suspicion term is not an instance and the recognized list must not count it; the limit lines are lines of text in a single fact, printed with the `~s` directive that takes a string, and `report/0` prints the parts in order:

```
% not_examined(What): data the report would need
% and the knowledge base does not hold.
not_examined('heap verification results').
not_examined('heap operation logs').
not_examined('raw stack words').
not_examined('function parameters').
not_examined('the environment block').
not_examined('executable regions not in modules').

% limit_lines(Lines): what the report does not
% claim.
limit_lines([
    "- that any recognized pattern is the cause",
    "  of the failure; a pattern is a state the",
    "  dump shows, and causes are inferred;",
    "- that a pattern not recognized is absent;",
    "  the recognizers see the facts the schema",
    "  holds, and the dump holds more;",
    "- that the thresholds are right for this",
    "  process; they encode an analyst's limits;",
    "- that the stack traces are correct; they",
    "  are the debugger's reconstructions;",
    "- anything about time before or after the",
    "  moment the dump was written, except what",
    "  the step dumps show at their moments."
]).

report_header :-
    process(PID, Name),
    format("Analysis report: ~a (PID ~d)~n",
           [Name, PID]),
    aggregate_all(count, thread(_, _), Threads),
    aggregate_all(count, module(_, _, _),
                  Modules),
    dump_info(kind, Kind),
    format("~a memory dump, ", [Kind]),
    format("~d threads, ~d modules~n",
           [Threads, Modules]),
    nl.
```

```
report_recognized :-
    format("Recognized, most severe first:~n"),
    findall(Rank-Pattern,
            recognized(Rank, Pattern),
            Pairs),
    keysort(Pairs, Sorted),
    forall(member(Rank-Pattern, Sorted),
           report_pattern(Rank, Pattern)),
    nl.

report_suspected :-
    format("Suspected, for the analyst:~n"),
    forall(suspicion(Pattern, Instance),
           ( format("[suspected] ~a~n",
                    [Pattern]),
             print_instance(Instance)
           )),
    nl.

report_absent :-
    format("Looked for and not recognized:~n"),
    forall(( severity(Pattern, _),
             \+ instance(Pattern, _)
           ),
           format("    ~a~n", [Pattern])),
    nl.

report_not_examined :-
    format("Not examined, for want of facts:~n"),
    forall(not_examined(What),
           format("    ~a~n", [What])),
    nl.

report_limits :-
    format("This report does not claim~n"),
    limit_lines(Lines),
    forall(member(Line, Lines),
           format("~s~n", [Line])).

% report: the analysis report of the dump.
report :-
    report_header,
    report_recognized,
    report_suspected,
    report_absent,
    report_not_examined,
    report_limits.


?- report.
Analysis report: AppServer.exe (PID 4120)
process memory dump, 10 threads, 10 modules

Recognized, most severe first:
[critical] Deadlock (Critical Sections)
    threads [103,104]
[critical] NULL Pointer (Data)
    thread 109
[major] Exception Stack Trace
    thread 109
```

```
[major] Insufficient Memory (Committed Memory)
    heap 0x210a0000000
    the process
[major] Insufficient Memory (Handle Leak)
    process 4120
[major] Memory Leak (Process Heap)
    heap 0x210a0000000
[major] Handle Leak
    event handles
[major] Spiking Thread
    thread 102
    thread 107
[minor] Frame Regularity
    thread 107: 8 frames of Node::Visit
[minor] Blocked Thread (Software)
    thread 101
    thread 103
    thread 104
    (entailed by Deadlock (Critical Sections))
[minor] Wait Chain (Critical Sections)
    wait chain [101,103,104]
    wait chain [103,104]
    wait chain [104,103]
    (entailed by Blocked Thread (Software))
[minor] Not My Version
    module dbclient
[minor] Hooked Functions (User Space)
    kernelbase!CreateFileW <- thirdparty_hook
    kernelbase!ReadFile <- thirdparty_hook
[minor] Unknown Component
    module thirdparty_hook
[informational] No Component Symbols
    module dbclient
    module cachelib
    module thirdparty_hook
[informational] Suspended Thread
    thread 108
[informational] Last Error Collection
    thread 109: last error 0x3e6
[informational] Main Thread
    thread 100
[informational] Step Dumps
    dumps [dump1a,dump1b,dump1]

Suspected, for the analyst:
[suspected] Stack Overflow (User Mode)
    thread 107

Looked for and not recognized:
    Manual Dump (Process)
    Truncated Dump
    Inconsistent Dump
    Early Crash Dump
    False Positive Dump
    Stack Overflow (User Mode)
```

```
Not examined, for want of facts:
    heap verification results
    heap operation logs
    raw stack words
    function parameters
    the environment block
    executable regions not in modules

This report does not claim
- that any recognized pattern is the cause
  of the failure; a pattern is a state the
  dump shows, and causes are inferred;
- that a pattern not recognized is absent;
  the recognizers see the facts the schema
  holds, and the dump holds more;
- that the thresholds are right for this
  process; they encode an analyst's limits;
- that the stack traces are correct; they
  are the debugger's reconstructions;
- anything about time before or after the
  moment the dump was written, except what
  the step dumps show at their moments.
true.
```

The report is the book's account of `dump1.pl` in one page. The order is the severity facts', and within a rank the order of the facts, since `keysort/2` is stable; the two critical findings are the deadlock and the null pointer dereference, which are the two failures the dump was chosen to contain, and everything below them is context for one or the other. Every instance is an instance by its chapter's definition: the **Insufficient Memory (Committed Memory)** (page 406) of Chapter 21 has two instances, a heap and the process as a whole, and its second variant has one; the wait chains are Chapter 20's, which stop before a thread they already hold. The second part is the one suspicion, thread 107, which the first part does not count and the third part lists as not recognized, since no exception record establishes the overflow; the three statements together are the precise position. The third part also says that the dump is, by the five tests of this chapter, a whole, consistent exception dump of a mature process, which is what licenses the reading of the first part. The fourth names the extensions of Chapters 21 and 22 that were not loaded into this toplevel, so that a reader knows the corruption, the double free, and the hooks' hidden code are findings of other chapters and not absences here.

The last part is the one no program can supply from the facts, and it is written as text because it is the analyst's position rather than a derivation. Each line is a limit the earlier chapters stated beside a predicate: that a recognizer finds a state and not a cause, from Chapter 17 onward; that absence of recognition is not absence, from the closed-world discussions of Chapters 21 and 23; that thresholds are the analyst's, from every `threshold/2` fact; that stack traces are reconstructions, from Chapter 15; and that a dump is a moment, from the series of Chapter 21. The report says them once, where a reader who has read nothing else will see them.

What the report does not do is decide. It does not say that the deadlock caused the hang that the operator saw, that the null pointer dereference is a consequence of the double free that this toplevel does not know about, or that the version mismatch in `dbclient` has anything to do with either; the

trace of Chapter 16 and the patterns of Part IV have more to say about the sequence of events, and Chapter 28 brings the two artifacts together. A program that produced those sentences would be producing the analyst's conclusions without the analyst, and the discipline of this book is that the facts, the recognizers, and the evidence terms are the program's, and the conclusion is signed by a person.

23.10.1 Two Boundary Cases

A report that composes general recognizers is only as general as they are, and two boundary cases check that the composition kept the definitions of the chapters rather than a shape that happens to fit the shared dump. The first is an access violation whose target is in the null page but whose access is `execute`: a call through a null function pointer, which is **NULL Pointer (Code)** (page 361) and not **NULL Pointer (Data)** (page 360), and which a recognizer that tested the code and the target without the access kind would misreport. The second is a stack trace in which one function occurs six times in three separate runs of two: a recognizer that counted occurrences, or added the runs, would report a **Frame Regularity** (page 342) against the threshold of four, and the longest run is two. Neither configuration is in `dump1.pl`, so the checks run in a fresh toplevel on an extension file that includes the shared dump, as Chapter 19 did, and adds two synthetic threads for this section and five more, of the shape Section 19.9 checked, for the two sections that follow:

```
% boundary.pl -- dump1.pl extended with seven
% synthetic threads for the boundary checks of
% Section 23.10. An extension of the schema; every
% value is invented. frame/5 is dynamic, so that
% the directive at the end can add frames.
:- discontiguous thread/2, frame/5,
   exception_record/3, exception_info/3.
:- dynamic frame/5.
:- include(dump1).
:- use_module(patterns_dump).

% 130: an access violation executing at address 0
thread(130, 4120).
exception_record(130, 0xc0000005, 0x0).
exception_info(130, access, execute).
exception_info(130, target, 0x0).

% 131: one function in three separate runs of two
thread(131, 4120).
frame(131, 0, appserver, f, 0x10).
frame(131, 1, appserver, f, 0x10).
frame(131, 2, appserver, g, 0x20).
frame(131, 3, appserver, f, 0x10).
frame(131, 4, appserver, f, 0x10).
frame(131, 5, appserver, g, 0x20).
frame(131, 6, appserver, f, 0x10).
frame(131, 7, appserver, f, 0x10).
```

```
% 132-134: twelve frames with a longest run of
% four, five, and six; 135: eight frames with a
% run of six; 136: three frames and the stack
% overflow status code. The frames are built by
% the directive below, one per function listed.
thread(132, 4120).
thread(133, 4120).
thread(134, 4120).
thread(135, 4120).
thread(136, 4120).
exception_record(136, 0xc00000fd,
                 0x7ff6_1d11_0010).

:- forall(member(TID-Functions,
                 [ 132-[f,f,f,f,g,h,i,j,k,l,m,n],
                   133-[f,f,f,f,f,g,h,i,j,k,l,m],
                   134-[f,f,f,f,f,f,g,h,i,j,k,l],
                   135-[f,f,f,f,f,f,g,h],
                   136-[a,b,c] ]),
          forall(nth0(N, Functions, F),
                 assertz(frame(TID, N, appserver,
                               F, 0x10)))).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- exception_record(130, Code, _),
   exception_info(130, access, Access),
   exception_info(130, target, Target),
   threshold(null_page, Limit).
Code = 3221225477,
Access = execute,
Target = 0,
Limit = 65536 ;
false.

?- null_pointer_data(130).
false.

?- null_pointer_data(T).
T = 109 ;
false.
```

Thread 130 has the access violation code and a target below the null page limit, and `null_pointer_data/1` rejects it, because the clause of Chapter 19 requires an access other than `execute`; the enumerator still finds thread 109 and nothing else. The second case is checked twice, on the run lengths and on the recognizer:

```
?- aggregate_all(count, frame(131, _, _, f, _),
                 N).
N = 6.

?- stack_trace_signature(131, _S), runs(_S, _Rs),
   pairs_values(_Rs, Lengths).
Lengths = [2, 1, 2, 1, 2].
```

```
?- frame_regularity(131, Key, N).
false.

?- retract(threshold(frame_run, 4)),
   assertz(threshold(frame_run, 2)),
   aggregate_all(max(N),
                 frame_regularity(131, _, N),
                 Max).
Max = 2.

?- frame_regularity(131, Key, N).
Key = appserver-f,
N = 2 ;
Key = appserver-f,
N = 2 ;
Key = appserver-f,
N = 2.
```

Six frames name `f`, in runs of two, one, two, one, and two, as `runs/2` of Chapter 18 computes them from the signature; with the threshold at four there is no instance, and when the threshold is lowered to two, which the dynamic `threshold/2` of the module permits with `retract/1` and `assertz/1`, each of the three runs is reported with its own length, never their sum, since `frame_regularity/3` enumerates the runs that reach the threshold and selects no longest. The lowered threshold stays in force until it is restored, which the next section does before anything else; an analysis that changes a threshold states the change beside its findings, as Section 17.4 asked.

23.10.2 The Two Run Thresholds

The report composes Chapter 19's `stack_overflow/2`, and that chapter's rule for a suspected overflow compares a run with `overflow_run`, six, where Chapter 18's **Frame Regularity** (page 342) compares with `frame_run`, four. A module that kept one threshold for both would change which stacks the report suspects: under four, a twelve-frame stack with a run of four would be a suspicion that Chapter 19 rejects. The module keeps the two, and the following checks it on the five synthetic stacks of `boundary.pl`, after restoring the threshold the previous section lowered. `run_report/1` prints, for one thread, its depth, the runs `frame_regularity/3` enumerates, and the functors of the findings of `stack_overflow/2`:

```
% run_report(TID): the depth of TID, the runs that
% reach frame_run, and the forms of its findings.
run_report(TID) :-
    stack_depth(TID, Depth),
    findall(C, frame_regularity(TID, _, C), Runs),
    findall(Form,
            ( stack_overflow(TID, F),
              functor(F, Form, 3)
            ),
            Forms),
    format("~d depth ~d runs ~w ~w~n",
           [TID, Depth, Runs, Forms]).
```

```
?- retract(threshold(frame_run, 2)),
   assertz(threshold(frame_run, 4)),
   threshold(frame_run, R),
   threshold(overflow_run, O).
R = 4,
O = 6.

?- forall(member(T, [107, 132, 133, 134, 135,
                     136]),
          run_report(T)).
107 depth 14 runs [8] [suspected]
132 depth 12 runs [4] []
133 depth 12 runs [5] []
134 depth 12 runs [6] [suspected]
135 depth 8 runs [6] []
136 depth 3 runs [] [evidence]
true.

?- stack_overflow(T, suspected(_, _, _Gs)),
   memberchk(threshold(overflow_run, Min), _Gs).
T = 107,
Min = 6 ;
T = 134,
Min = 6 ;
false.
```

The three twelve-frame stacks are **Frame Regularity** (page 342) instances, and one of them is a suspected overflow, the one whose run reaches six; the eight-frame stack is not deep enough, and the three-frame stack with the status code is an instance by the first clause alone. The grounds of each suspicion name the threshold they were compared with, which is the second one. These are Section 19.9's answers, obtained here from the module's clauses, as the equivalence of the two definitions requires.

23.10.3 The Evidence Contract from an Importing Module

Section 17.3 fixed the execution context of an evidence list: its leaves are callable, as written, from any module that imports the module that defines the recognizer. The claim is about the module's public interface, and it is checked from outside the module, here from `user`, which `boundary.pl` made an importer. `check_explainers/0` runs every explainer the module lists in `explainer/2` on the loaded knowledge base and, for every evidence or suspicion term, checks the three parts of the contract: the leaves are ground; each leaf, called in the module that called `check_explainers/0`, succeeds; and the finding re-derives from its leaves alone. The last check is `sufficient/1`, which calls the explainer again under `derivable/2`, a meta-interpreter in the manner of Chapter 29 that expands the module's clauses, evaluates the first-order built-ins and arithmetic of a closed list, interprets the all-solutions and apply predicates so that the goals they would call are derived rather than run, takes a relation of the dump or a threshold from the list and from nowhere else, and takes a derived goal such as `stack_depth/2` as given when the list holds it; the finding is sufficient when that derivation reconstructs it, name, instance, and list. The vocabulary of the derivation is stated, and checked, at the end of this section. The predicate prints one line per explainer and one per finding that breaks the contract, and fails if there is one:

```
?- explainer(Goal, _), functor(Goal, Name, Arity),
   format("~a/~d~n", [Name, Arity]), fail.
spiking_thread/2
blocked_thread/2
null_pointer_data/2
stack_overflow/2
deadlock/3
insufficient_memory/3
false.

?- check_explainers.
spiking_thread/2: 2 of 2 ok
blocked_thread/2: 3 of 3 ok
null_pointer_data/2: 1 of 1 ok
stack_overflow/2: 3 of 3 ok
deadlock/3: 1 of 1 ok
insufficient_memory/3: 1 of 1 ok
true.

?- spiking_thread(102, evidence(_, _, _Fs)),
   maplist(call, _Fs),
   predicate_property(cpu_time(_, _),
                      imported_from(M)).
M = patterns_dump.
```

Eleven findings of six explainers pass, three of them for **Stack Overflow (User Mode)** (page 366), the status code of thread 136 and the suspicions of threads 107 and 134. The third query is the test of the interface in its plainest form: the evidence of thread 102 lists `cpu_time(102, 9680)` and `total_cpu(14135)`, helpers of Chapter 18, and `maplist(call, _Fs)` runs in `user` because the module exports them; a module that exported the recognizers and kept the helpers private would give the same list and an existence error on its first element, which is the defect Section 17.3 named.

Two evidence lists are easy to leave insufficient in one way: by omitting a threshold the recognizer compared against. The heap instance of **Insufficient Memory (Committed Memory)** (page 406) rests on the ratio limit and on the reservation floor `large_heap_kb`, and **NULL Pointer (Data)** (page 360) rests on the exception record, its parameters, and the null page limit; a list without the second threshold is true in every element and does not entail the recognition. Both lists carry their thresholds, and `check_finding/3`, the test behind `check_explainers/0`, says what happens to a list from which one is taken away:

```
?- insufficient_memory(committed_memory, heap(H),
                       evidence(_, _, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
heap(2270427086848,2097152,1835008,24576)
threshold(committed_ratio,0.85)
threshold(large_heap_kb,1048576)
H = 2270427086848 ;
false.
```

```
?- null_pointer_data(109, evidence(_, _, _Fs)),
   forall(member(F, _Fs), ( print(F), nl )).
exception_record(109,3221225477,140718380167594)
exception_info(109,access,read)
exception_info(109,target,24)
threshold(null_page,65536)
true ;
false.

?- insufficient_memory(committed_memory, I,
                       evidence(P, I, _Fs)),
   selectchk(threshold(large_heap_kb, _), _Fs,
             _Fs1),
   check_finding(user, evidence(P, I, _Fs1), V).
I = heap(2270427086848),
P = 'Insufficient Memory (Committed Memory)',
V = insufficient ;
false.

?- null_pointer_data(T, evidence(P, T, _Fs)),
   selectchk(threshold(null_page, _), _Fs, _Fs1),
   check_finding(user, evidence(P, T, _Fs1), V).
T = 109,
P = 'NULL Pointer (Data)',
V = insufficient ;
false.

?- spiking_thread(102, evidence(P, I, _Fs)),
   _Fs1 = [private_helper(102) | _Fs],
   check_finding(user, evidence(P, I, _Fs1),
                 raises(Leaf, _Error)),
   _Error = existence_error(What, PI).
P = 'Spiking Thread',
I = 102,
Leaf = private_helper(102),
What = procedure,
PI = private_helper/1.
```

The lists without their threshold are ground and every leaf of each still succeeds, so the first two parts of the contract pass; the re-derivation fails, since `derivable/2` reaches the threshold goal and finds no limit in the list to take, and the verdict is `insufficient`. The last query is the interface defect in miniature: a leaf whose predicate the importing module cannot see is reported as the existence error it raises, with the leaf named, which is the verdict a private helper in an evidence list receives. `selectchk/3` removes the first element that unifies and leaves no choice point.

A successful re-derivation means that the list entails the finding only if the derivation can reach nothing but the list. `derivable/2` therefore classifies every goal it meets by its predicate indicator, after removing any module qualification, since a qualification says where a predicate is defined and not where a fact may be taken from. A goal of a relation that `dump_relation/1` lists, the fifteen relations of the schema of Section 15.2 and the four extension relations of Chapters 20 and 21, and a `threshold/2` goal, follow from the evidence list and from nowhere else; so does any other fact of `user`, the module that holds the knowledge base. The control constructs and the all-solutions and apply predicates the recognizers use, `findall/3`, `bagof/3`, `setof/3`, `aggregate_all/3`, `forall/2`,

`call/N`, `maplist/2` to `maplist/4`, `include/3`, `exclude/3`, and `foldl/4`, are interpreted, so that the goals they would call are derived in turn, which is why `hooked_functions/3`, whose body is a `bagof/3` over `hooked/3`, cannot read the dump through it. A goal of a predicate with clauses, in the module or in a chapter, follows as a member of the list or through a clause whose body follows. A first-order built-in that `evaluable/1` lists, unification, comparison, arithmetic, the type tests, term construction, and the predicates over atoms, strings, and lists, none of which reads the database, calls a goal, or dispatches through a hook that a loaded file could extend, is called. `must_be/2` is not on that list, although the recognizers use it, because it dispatches through `has_type/2` of `library(error)`, a multifile predicate to which any loaded file may add a type whose clause reads the dump; the derivation decides `must_be/2` itself, by `checked/2`, for the fixed descriptors of `derivable_type/1`, `oneof/1`, `atom`, `integer`, `list`, `list/1` and a few more, with the tests and the error terms of `library(error)` reproduced clause for clause, so that it succeeds, binds and raises exactly as the library predicate does on those descriptors; the whole descriptor is validated first, element types included, and one outside the list is a domain error whatever the term, which is the one way the two differ. For the same reason a goal or a list that carries an attributed variable of Chapter 14 is refused with a permission error before anything is derived, since unifying such a variable runs the hook attached to it. Each evaluable predicate is called in the module that defines it, `system` or the library's, which `evaluable/2` records: a meta-call from the module would otherwise resolve a name through `user`, the module's default import module of Chapter 13, before the library, so that a predicate of the same name defined in a loaded file would be run in the library's place. Any other goal is outside the vocabulary and raises a domain error, rather than being called with whatever it might reach. Four routes a derivation must not take, each of which would obtain a fact of the loaded dump from an empty evidence list, are checked, with the result the recognizer gives once its facts are supplied, and the error for a goal outside the vocabulary:

```
?- derivable([],
             user:thread_time(102, 8450, 1230)).
false.

?- derivable([], thread_time(102, 8450, 1230)).
false.

?- derivable([],
             call(thread_time(102, 8450, 1230))).
false.

?- derivable([], hooked_functions(_, _, _)).
false.

?- hooked_functions(M, H, Fs),
   findall(hooked(F, M, H), member(F, Fs), _Hs),
   derivable(_Hs, hooked_functions(M, H, Fs1)).
M = kernelbase,
H = thirdparty_hook,
Fs = Fs1, Fs1 = ['CreateFileW', 'ReadFile'].

?- catch(derivable([], assertz(seen(102))),
         error(E, _), true).
E = domain_error(derivable_goal, assertz/1).
```

```
?- aggregate_all(count, dump_relation(_), R),
   aggregate_all(count, evaluable(_), E).
R = 19,
E = 78.
```

The fact `thread_time(102, 8450, 1230)` is in the loaded dump, and the first three queries fail because the list is empty: the module qualification, its absence, and the indirection through `call/1` make no difference to where the fact is looked for. The fourth fails for the same reason one level down, since the `bagof/3` is interpreted and its goal, `user:hooked/3`, is a dump relation; the fifth supplies the two `hooked/3` facts and obtains the recognizer's own answer. `assertz/1` is neither interpreted nor evaluable, and the derivation says so rather than running it. The last query gives the size of the vocabulary: nineteen relations taken from the list only, and seventy-eight built-ins called.

The three routes that pass through a loaded file's definitions are checked in the same way. The program below adds a type to `library(error)`, as any loaded file may, whose clause reads a `thread_time/3` fact of the dump; `must_be/2` itself then succeeds on the live fact, and the derivation refuses the descriptor, alone and inside `list/1`, while deciding the descriptors it knows with the library's own errors. The program also defines `numlist/3` in `user`, which the toplevel then reaches in place of the library's, and the derivation does not. The last query attaches a goal over the dump to a variable with `freeze/2` and asks the derivation to bind it:

```
% has_type(captured_thread, T): a type that a
% loaded file could add to library(error), whose
% clause reads the dump. must_be/2 dispatches
% through this multifile hook.
:- multifile error:has_type/2.

error:has_type(captured_thread, T) :-
    user:thread_time(T, 8450, 1230).

% numlist(Low, High, List) in user: a meta-call
% from the module would reach it before the
% library's.
numlist(_, _, from_user).

?- must_be(captured_thread, 102).
true.

?- catch(derivable([],
                   must_be(captured_thread, 102)),
         error(E, _), true).
E = domain_error(derivable_type, captured_thread).

?- _T = list(captured_thread),
   catch(derivable([], must_be(_T, [102])),
         error(E, _), true).
E = domain_error(derivable_type, captured_thread).

?- derivable([], must_be(oneof([a, b]), a)).
true.
```

```
?- catch(derivable([], must_be(oneof([a, b]), c)),
         error(E, _), true).
E = type_error(oneof([a, b]), c).

?- numlist(1, 3, L).
L = from_user.

?- derivable([], numlist(1, 3, L)).
L = [1, 2, 3].

?- freeze(X, user:thread_time(X, 8450, 1230)),
   catch(derivable([], X = 102), error(_E, _),
         true),
   _E = permission_error(Action, Kind, _).
Action = derive,
Kind = attributed_variable,
freeze(X, thread_time(X, 8450, 1230)).
```

The hook is live, as the first query shows, and the derivation never reaches it. The type is refused by name rather than evaluated, because a list of admissible predicates cannot constrain a descriptor whose meaning a later file defines; the nested descriptor shows why the check must go through the structure of the type and not stop at its outermost functor. The `numlist/3` of `user` is what the toplevel runs and not what the derivation runs, which called `lists:numlist/3` by its module.

That `checked/2` is `must_be/2` on its vocabulary is a claim about two predicates, and it is checked differentially: `agrees/2` runs the library predicate and the derivation on the same descriptor and term, records whether each succeeds or raises, and which error, under a double negation so that neither binds the term, and compares the two outcomes as variants with `=@=`, since an error term may carry a fresh variable; `report/2` prints the two outcomes where they differ. The cases include the ones that are easy to get wrong: a compound with an unbound argument against `oneof/1`, which the library refuses with an instantiation error rather than binding it; a float within integer bounds of `between/2`, which is a type error, not a number in range; a partial list; and `nonneg` on an atom, whose error names `nonneg` and not `integer`. The last three cases are the one intended difference, a descriptor outside the vocabulary, which the library accepts, trivially on an empty list and through the hook otherwise, and the derivation refuses by name:

```
% outcome(Goal, Outcome): true or false for a goal
% that succeeds or fails without binding anything,
% or raises(E) for one that raises E.
outcome(Goal, Outcome) :-
    catch(( \+ \+ Goal
          ->  Outcome = true
          ;   Outcome = false
          ),
          error(E, _),
          Outcome = raises(E)).

% agrees(Type, X): must_be(Type, X) and its
% derivation from an empty list have the same
% outcome, up to variable renaming; otherwise the
% two outcomes are printed.
agrees(Type, X) :-
```

```
    outcome(must_be(Type, X), Library),
    outcome(derivable([], must_be(Type, X)),
            Derived),
    (   Library =@= Derived
    ->  true
    ;   format("~q on ~q:~n", [Type, X]),
        report(library, Library),
        report(derived, Derived)
    ).

report(Side, raises(E)) :-
    !,
    format("  ~w raises~n    ~q~n", [Side, E]).
report(Side, Outcome) :-
    format("  ~w ~q~n", [Side, Outcome]).

?- _Cases = [ oneof([a, b])-a, oneof([a, b])-c,
              oneof([f(a)])-f(_), oneof([a])-_,
              between(1, 3)-2, between(1, 3)-2.5,
              between(1, 3)-4,
              between(1.0, 3.0)-2.5,
              nonneg-a, nonneg-(-1), nonneg-0,
              positive_integer-0, integer-_,
              list-[a | _], list-foo,
              list(integer)-[1, a],
              list(integer)-[1 | _],
              list(integer)-[], list(_)-[1],
              atom-1, boolean-maybe, callable-3,
              atomic-f(x), number-x, _-a,
              list(captured_thread)-[],
              list(captured_thread)-[102],
              captured_thread-102 ],
   forall(member(T-X, _Cases), agrees(T, X)).
list(captured_thread) on []:
  library true
  derived raises
    domain_error(derivable_type,captured_thread)
list(captured_thread) on [102]:
  library true
  derived raises
    domain_error(derivable_type,captured_thread)
captured_thread on 102:
  library true
  derived raises
    domain_error(derivable_type,captured_thread)
true.
```

The frozen goal is refused before the unification that would have run it; the error names a plain copy of the variable, and the toplevel prints the goal still attached to `X`, which the derivation left unbound.

Part IV. Trace and Log Analysis Patterns

Part IV treats the trace and log analysis patterns over the `msg/6` knowledge base of Chapter 16, in five chapters: the structure of a trace, time, activities and threads of activity, messages, and sequences, motifs, and correlation across traces and between a trace and a dump. The shared trace `trace1.pl` is small enough to be read in full and rich enough to exhibit most of the patterns; where it lacks an instance, a section extends it and says so. As in Part III, each section states the recognizer's operational definition, which is often narrower than the catalog's description, and what the analyst must still judge.

Chapter 24. Trace Structure

Part IV treats the trace and log analysis patterns with the vocabulary of Chapter 17: each pattern becomes a predicate over the `msg/6` facts of Chapter 16, named by the snake_case of the catalog name, in the Boolean, enumerator, or explainer form, with thresholds as facts. This chapter covers the patterns that describe the structure of a software trace before any question about time or activity is asked: what the trace contains, along which dimensions its messages are distinguished, how messages are grouped into sets and the trace into parts, and what its edges and gaps tell. They are the patterns an analyst applies in the first minutes with an unfamiliar log, and Chapters 25 to 28 build on them.

The chapter consults `trace1.pl` and defines the helpers it needs. A **message set** is represented throughout Part IV as a sorted list of message numbers without duplicates, the ordered set of `library(ordsets)`; the message number is the primary key of the schema, so a set of numbers identifies a set of messages and the `msg/6` facts supply the rest on demand. The toplevel prints a list of more than ten elements with `|...` in place of the rest, and a `msg/6` term does not fit the page, so message sets are printed by `print_messages/1`: one line per message with its number, time, thread, component, and the first 23 characters of its text, which is what fits in fifty columns. The column stops are the `~t~N|` directives of Chapter 6.

```
:- [trace1].
:- use_module(library(dcg/basics)).

% truncated(Text, Width, Short): Short is Text cut
% to at most Width characters.
truncated(Text, Width, Short) :-
    string_length(Text, Len),
    (   Len =< Width
    ->  Short = Text
    ;   sub_string(Text, 0, Width, _, Short)
    ).

% print_row(N): print message N on one line.
print_row(N) :-
    msg(N, Time, _, TID, Component, Text),
    truncated(Text, 23, Short),
    format("~t~d~2| ~t~3f~9| ~d ~a~t~26| ~s~n",
           [N, Time, TID, Component, Short]).

% print_messages(Ns): print the messages whose
% numbers are in Ns, one per line.
print_messages(Ns) :-
    forall(member(N, Ns), print_row(N)).

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

```
?- print_messages([1, 26, 71]).
 1  0.000 100 appserver    Service starting, versi
26  2.003 103 dbclient     Send start: session=71,
71 14.010 100 appserver    Service stopping
true.
```

The flag `toplevel_print_anon` is set to `false` for the whole chapter, as in Chapter 9, so that a variable whose name begins with an underscore can hold a long message set in a query without the toplevel printing it.

24.1 Basic Facts

Basic Facts are the measurements an analyst takes of a trace before reading any message: how many messages it has, when it starts and ends and how long it covers, and which processes, threads, and components wrote to it. Their use is comparison with what is known from outside the trace, the problem description above all: a trace that ends before the reported time of the problem, or that has no messages from the component the problem is attributed to, will not answer the question it was collected for, and the basic facts say so at once.

Each fact is one aggregation of Chapter 9, and the predicate collects them in a list of `Key=Value` terms so that a report can print them and a program can look one up with `memberchk/2`:

```
% basic_facts(Facts): Facts is a list of Key=Value
% terms describing the trace as a whole.
basic_facts(Facts) :-
    aggregate_all(count, msg(_, _, _, _, _, _),
                  Count),
    aggregate_all(min(T), msg(_, T, _, _, _, _),
                  First),
    aggregate_all(max(T), msg(_, T, _, _, _, _),
                  Last),
    aggregate_all(set(P), msg(_, _, P, _, _, _),
                  PIDs),
    aggregate_all(set(T), msg(_, _, _, T, _, _),
                  TIDs),
    aggregate_all(set(C), msg(_, _, _, _, C, _),
                  Cs),
    Duration is Last - First,
    Facts = [messages=Count, first=First,
             last=Last, duration=Duration,
             processes=PIDs, threads=TIDs,
             components=Cs].

% print_basic_facts: one line per basic fact;
% lists are printed as their lengths.
print_basic_facts :-
    basic_facts(Facts),
    forall(member(Key=Value, Facts),
           print_basic_fact(Key, Value)).
```

```
print_basic_fact(Key, Value) :-
    (   is_list(Value)
    ->  length(Value, Len),
        format("~a~t~12|~d~n", [Key, Len])
    ;   float(Value)
    ->  format("~a~t~12|~3f~n", [Key, Value])
    ;   format("~a~t~12|~w~n", [Key, Value])
    ).

?- print_basic_facts.
messages    71
first       0.000
last        14.010
duration    14.010
processes   1
threads     7
components  4
true.

?- basic_facts(_F), memberchk(threads=Ts, _F).
Ts = [100, 101, 102, 103, 104, 108, 109].
```

Seventy-one messages over fourteen seconds, from one process, seven threads, and four components. The thread list is already informative beside `dump1.pl`, whose process has ten threads: three never wrote a message, which Section 24.15 returns to. The predicate is deterministic and succeeds on any knowledge base with at least one `msg/6` fact; on an empty one `min` and `max` fail, as they should, since an empty trace has no first message.

The limit of the formalization is the comparison the pattern exists for. The predicate measures the trace; it does not know the problem description, the time of the incident, or the component under suspicion. A program can hold those as facts and check the basic facts against them, which is the form of Section 24.13.

24.2 Trace Dimension

A message has several independent attributes along which messages are told apart, and each is a **Trace Dimension**: time, process, thread, and component in the `msg/6` schema, with the message text as a dimension of its own. The extent of the trace along a dimension, the number of distinct values that occur, classifies the trace: one process and seven threads make `trace1.pl` a single-process, multi-threaded trace, and which patterns apply depends on that, since a thread of activity means nothing in a trace with one thread.

The dimensions are named by atoms, and one predicate relates a message to its value along a dimension; everything else is derived from it:

```
% dimension(Dim): Dim is a dimension of the trace
% along which messages are grouped.
dimension(process).
dimension(thread).
dimension(component).

% dimension_value(Dim, N, Value): message N has
% Value along dimension Dim.
dimension_value(process, N, PID) :-
    msg(N, _, PID, _, _, _).
dimension_value(thread, N, TID) :-
    msg(N, _, _, TID, _, _).
dimension_value(component, N, C) :-
    msg(N, _, _, _, C, _).

% trace_dimension(Dim, Values): Values is the set
% of values occurring along Dim.
trace_dimension(Dim, Values) :-
    dimension(Dim),
    aggregate_all(set(V),
                  dimension_value(Dim, _, V),
                  Values).

% dimension_size(Dim, Size): Size is the number of
% distinct values along Dim.
dimension_size(Dim, Size) :-
    trace_dimension(Dim, Values),
    length(Values, Size).

?- trace_dimension(component, C).
C = [appserver, cachelib, dbclient, reportengine].

?- dimension_size(D, S).
D = process,
S = 1 ;
D = thread,
S = 7 ;
D = component,
S = 4.
```

With the dimension bound, `dimension/1` succeeds once by first-argument indexing and the predicate is deterministic; unbound, the three dimensions are enumerated. Time is not in the list, because its values are ordered rather than grouped; Chapter 25 is about that dimension. Nor is the text, whose values are all distinct until something is abstracted from them, which Section 24.17 does.

24.3 Message Set

A **Message Set** is a collection of messages selected by a shared property, treated as one object of analysis: the messages of a component, of a thread, of a kind, or those matching a word. The pattern's point is that sets combine: the messages of thread 101 that belong to the database client are an intersection, the messages that are not warnings are a complement, and a question that is hard to ask of the trace is often easy to ask of two sets.

With sets as sorted lists of numbers, the selection is a `findall/3` followed by `sort/2`, and the simplest way to say which messages are wanted is a partially instantiated `msg/6` term, the pattern matching of Chapter 3:

```
% message_set(Template, Set): Set is the set of
% numbers of the messages unifying with Template,
% a msg/6 term with variables for the free fields.
message_set(Template, Set) :-
    Template = msg(N, _, _, _, _, _),
    findall(N, Template, Ns),
    sort(Ns, Set).

% text_set(Sub, Set): Set is the set of messages
% whose text contains the string Sub.
text_set(Sub, Set) :-
    findall(N,
            ( msg(N, _, _, _, _, Text),
              sub_string(Text, _, _, _, Sub) ),
            Ns),
    sort(Ns, Set).

?- message_set(msg(_, _, _, 101, dbclient, _), S).
S = [13, 14, 54, 55].

?- text_set("Error", S).
S = [43, 45, 47, 69].

?- message_set(msg(_, _, _, 101, _, _), _A),
   text_set("Request", _B),
   ord_intersection(_A, _B, C), length(C, L).
C = [12, 17, 53, 58, 59, 60, 61, 62, 63|...],
L = 12.

?- text_set("Request", _A), text_set("Error", _B),
   ord_union(_A, _B, _U), length(_U, L).
L = 29.
```

The first unification in `message_set/2` names the message-number argument of the template so that `findall/3` can collect it and leaves the caller's other arguments as they were; a caller that binds nothing selects the whole trace, which Section 24.6 uses. The `sort/2` makes the list an ordered set, so the predicates of `library(ordsets)` apply: `ord_intersection/3`, `ord_union/3`, `ord_subtract/3` for the complement relative to another set, `ord_subset/2`, and `ord_disjoint/2`. They run in linear time on sorted input and, unlike `intersection/3` and `union/3` of Chapter 5, produce sorted output, so the result is again a message set.

A set here is a set of message numbers and nothing more. The property that selected it is not recorded with it, so two sets selected by different templates that contain the same messages are the same set; a program that must know why a message is in a set keeps the template beside it, as the evidence terms of Chapter 17 do.

24.4 Vocabulary Index

The **Vocabulary Index** of a trace lists the distinct words of its messages with the messages each word occurs in. Read from the top it gives the routine vocabulary of the system, which is how an analyst learns the names of the operations a trace records without documentation; read from the bottom it gives the words that occur once, among which are the names of the things that happened once. Chapter 9 built the frequency half of the index over `trace1.pl` with `split_string/4` and noted that its word definition split a file path at the colon and kept the drive letter as a word.

The refinement is in the separators. A word is a maximal run of characters between any of space, comma, colon, equals sign, slash, backslash, period, parenthesis, the two characters of the arrow `->`, and the percent sign; the run is lowercased with `string_lower/2`. The separator set is a fact so that it can be changed in one place:

```
% word_separators(Seps): the characters at which
% message text is split into words.
word_separators(" ,:=/\\.()->%").

% word_of(N, Word): Word is a lowercased word of
% the text of message N.
word_of(N, Word) :-
    msg(N, _, _, _, _, Text),
    word_separators(Seps),
    split_string(Text, Seps, Seps, Parts),
    member(Part, Parts),
    Part \== "",
    string_lower(Part, Word).

% lexical(Word): Word begins with a letter.
lexical(Word) :-
    string_code(1, Word, C),
    code_type(C, alpha).

% word_messages(Word, Set): Set is the non-empty
% set of messages containing Word.
word_messages(Word, Set) :-
    aggregate_all(set(N), word_of(N, Word), Set),
    Set \== [].

% vocabulary_index(Index): Index is the list of
% Word-Set pairs, one per distinct word, in the
% standard order of the words.
vocabulary_index(Index) :-
    findall(Word-N, word_of(N, Word), Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Groups),
    maplist(dedupe, Groups, Index).

dedupe(Word-Ns, Word-Set) :-
    sort(Ns, Set).
```

```
% print_vocabulary(Max): the Max most frequent
% lexical words with their message counts.
print_vocabulary(Max) :-
    vocabulary_index(Index),
    forall(limit(Max,
                 order_by([desc(Count)],
                          frequent(Index, Word,
                                   Count))),
           format("~t~d~3| ~s~n", [Count, Word])).

frequent(Index, Word, Count) :-
    member(Word-Set, Index),
    lexical(Word),
    length(Set, Count).


?- vocabulary_index(_I), length(_I, N).
N = 118.

?- aggregate_all(count,
   ( vocabulary_index(_I), member(W-_, _I),
     lexical(W) ), N).
N = 69.

?- print_vocabulary(8).
 28 request
 21 worker
 13 ms
 11 received
 11 report
 11 to
 10 start
  9 dispatched
true.

?- word_messages("error", S).
S = [43, 45, 47, 69].

?- word_messages("session", S).
S = [5, 23, 25, 26, 27, 29, 30, 67, 68].

?- word_messages("nonesuch", S).
false.
```

`group_pairs_by_key/2` from `library(pairs)` turns a key-sorted list of pairs into one pair per key with the list of that key's values, and is the idiom this chapter uses wherever messages are grouped by an attribute; `keysort/2` before it makes equal keys adjacent. The `dedupe/2` step matters because a word may occur twice in one message. `code_type/2` with `alpha` tests a character code for being a letter, digit, or underscore in SWI-Prolog, so `"0x3e6"` counts as lexical, a looseness the index tolerates. The 118 distinct words are fewer than Chapter 9's 123 because the request paths now split into words that already occur, `report`, `session`, `17`, and the arrow is no longer a word.

The index relates a word to messages, not only to a count, and that is what makes it an index: `word_messages/2` answers where a word occurs with a message set the other predicates of this chapter accept. Its limit is the word definition. Splitting at separators gives `db01` and `5432` from the

host and port, and nothing at all for a message whose significance lies in its structure; Section 24.17 abstracts the structure instead, and Chapter 27 takes the text as its subject.

24.5 Trace Partition

A **Trace Partition** divides the trace into parts that do not overlap and together make up the whole, so that each part can be measured and read on its own and the parts compared. The division may follow a dimension, one part per thread or per component, or follow boundaries in time, such as startup, main activity, and shutdown; the pattern is the division, and the sizes of the parts are its first result.

Along a dimension, the partition is `dimension_value/3` grouped by its value:

```
% trace_partition(Dim, Partition): Partition is
% the list of Value-Set pairs dividing the trace
% by its values along Dim.
trace_partition(Dim, Partition) :-
    findall(V-N, dimension_value(Dim, N, V),
            Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Partition).

% print_partition(Partition): one line per part
% with its size and first and last message.
print_partition(Partition) :-
    forall(member(Part, Partition),
           print_part(Part)).

print_part(V-Set) :-
    length(Set, Len),
    Set = [First | _],
    last(Set, Last),
    format("~w~t~14|~t~d~4+~t~d~5+~t~d~5+~n",
           [V, Len, First, Last]).


?- trace_partition(component, _P),
   print_partition(_P).
appserver       47    1   71
cachelib         1    3    3
dbclient        12    4   69
reportengine    11   15   57
true.

?- trace_partition(thread, _P),
   print_partition(_P).
100             22    1   71
101             21    7   66
102             11    8   36
103              4    9   26
104              4   10   30
108              5   48   52
109              4   67   70
true.
```

The parts by component are the counts Chapter 9 obtained with `aggregate/3`, now with the message sets attached. The parts by thread already sketch the story: the main thread 100 writes from the first message to the last, worker 101 from message 7 to 66, workers 103 and 104 stop at messages 26 and 30 and are not heard from again, and threads 108 and 109 appear only late.

That the parts are a partition is a property the program can check rather than assume. Two helpers say that a list of sets is pairwise disjoint and that its union is the whole trace; `foldl/4` of Chapter 10 threads the running union through the list:

```
% exhaustive(Sets): the union of Sets is the whole
% trace.
exhaustive(Sets) :-
    foldl(ord_union, Sets, [], Union),
    message_set(msg(_, _, _, _, _, _), Union).

% disjoint(Sets): no two sets in Sets share a
% message.
disjoint([]).
disjoint([Set | Sets]) :-
    foldl(ord_union, Sets, [], Rest),
    ord_disjoint(Set, Rest),
    disjoint(Sets).

?- trace_partition(thread, _P),
   pairs_values(_P, _Sets),
   exhaustive(_Sets), disjoint(_Sets).
true.

?- vocabulary_index(_I), pairs_values(_I, _Sets),
   disjoint(_Sets).
false.
```

The second query makes the contrast: the sets of the vocabulary index are not a partition, since a message contains many words, and `disjoint/1` says so. A partition by time, into phases, is a partition in the same sense but with parts defined by boundary messages rather than by a dimension; Chapter 25 finds the boundaries under **Milestones** (page 531), and the message sets between them are its parts.

24.6 Message Space

The catalog's **Message Space** is the view of messages as points of a space in which message sets are subsets and the trace itself is one subset: the space of all messages the traced components could have written, of which a particular trace realizes some. Within the realized part, a message is located by its coordinates along the dimensions of Section 24.2, and the relations between message sets, inclusion, disjointness, and complement, are the relations of subsets of that space.

The realized space is the message set selected by the template with nothing bound, and the coordinates of a point are the fields of its fact other than the text:

```
% message_space(Space): Space is the set of all
% messages of the trace.
message_space(Space) :-
    message_set(msg(_, _, _, _, _, _), Space).

% coordinates(N, Point): Point locates message N
% along the ordered and grouped dimensions.
coordinates(N, point(N, Time, PID, TID, C)) :-
    msg(N, Time, PID, TID, C, _).

?- message_space(_S), length(_S, L).
L = 71.

?- coordinates(26, P).
P = point(26, 2.003, 4120, 103, dbclient).

?- message_space(_S),
   message_set(msg(_, _, _, 108, _, _), _T),
   ord_subset(_T, _S),
   ord_subtract(_S, _T, _Rest), length(_Rest, L).
L = 66.
```

What the predicate does not capture is the unrealized part of the space. The messages the components could have written are fixed by the trace statements in their source code, and no fact in `trace1.pl` records those; a knowledge base that had them as facts could compare them with the realized kinds of Section 24.18 and find the statements that never fired. Without that, the space and the trace coincide, and the predicate renames the whole trace as a set. The renaming is still worth having, because the predicates of this chapter are written over sets, and the whole trace is the set they most often start from.

24.7 Head of a Message Set

The **head** of a trace, or of any message set, is its first message; it is not a catalog pattern but a helper that several patterns use. For the whole trace it tells whether the recording caught the beginning of what it was meant to record, since a trace that opens in the middle of an activity has lost its beginning; for a thread of activity it tells how the thread announced itself, and for an adjoint thread, in the sense of Chapter 26, where the activity it follows started.

On a sorted set the head is the first element, and the predicate is a single clause:

```
% head_message(Set, N): N is the first message of
% the non-empty message set Set.
head_message([N | _], N).

?- message_space(_S), head_message(_S, N),
   print_messages([N]).
 1   0.000 100 appserver    Service starting, versi
N = 1.
```

```
?- trace_partition(thread, _P),
   forall(member(_-_Set, _P),
          ( head_message(_Set, N),
            print_messages([N]) )).
 1  0.000 100 appserver    Service starting, versi
 7  0.215 101 appserver    Worker 101 ready
 8  0.215 102 appserver    Worker 102 ready
 9  0.216 103 appserver    Worker 103 ready
10  0.216 104 appserver    Worker 104 ready
48  9.002 108 appserver    Housekeeping: watchdog,
67 14.000 109 appserver    Request 10 received: GE
true.
```

The head of the trace is the service announcing its start, and the heads of the four workers are their readiness messages; the trace caught the beginning. The heads of threads 108 and 109 are a watchdog report and the arrival of a request, neither of which announces a thread, so whether those threads existed before their first message the trace cannot say. `dump1.pl` shows thread 109 as a worker like the others, which makes its missing readiness message a question for Section 24.15.

24.8 Tail of a Message Set

The **tail** is the last message of a set, the second of the two helpers. For the whole trace it is the last thing recorded, and whether it is the kind of message a trace is expected to end with, a shutdown, a completion, decides whether the recording ran to the end; for a thread, it is the last thing the thread said, which is the beginning of the **Last Activity** (page 532) pattern of Chapter 25.

```
% tail_message(Set, N): N is the last message of
% the non-empty message set Set.
tail_message(Set, N) :-
    last(Set, N).

?- message_space(_S), tail_message(_S, N),
   print_messages([N]).
71 14.010 100 appserver    Service stopping
N = 71.

?- trace_partition(thread, _P),
   forall(member(_-_Set, _P),
          ( tail_message(_Set, N),
            print_messages([N]) )).
71 14.010 100 appserver    Service stopping
66  9.309 101 appserver    Request 9 completed: 20
36  5.000 102 reportengine Warning: table resize,
26  2.003 103 dbclient     Send start: session=71,
30  2.103 104 appserver    Session 9 lookup start
52  9.005 108 appserver    Housekeeping: worker 10
70 14.003 109 appserver    Unhandled exception 0xc
true.
```

The trace ends with the service stopping, so it ran to the end; the threads end very differently. Worker 101 ends on a completion, worker 102 on a warning in the middle of a build, workers 103 and 104 on the start of operations never reported as finished, and thread 109 on an unhandled

exception. Four of the seven tails are the last word of something unfinished, and that is the diagnostic content of the pattern: a set whose tail is a start message is an activity cut off, by the end of the trace or by something else. `last/2` is linear in the length of the set, which matters on a set of a million messages; a representation that kept the bounds beside the set would avoid it.

24.9 Trace Frames

Trace Frames divide the trace into consecutive windows of fixed size, by message count or by time, as a film is divided into frames, so that the same measurements can be taken frame by frame and their change along the trace observed. A measure that is uninformative over the whole trace, the set of active components for instance, becomes informative as a sequence of per-frame values.

Frames by message count are a recursion over the message space that takes a prefix of the given length while one exists. The frame size is a count of messages and must be a positive integer, which the predicate checks with `must_be/2` of Chapter 7 before recursing:

```
% trace_frames(+Size, -Frames): Frames is the list
% of I-Set pairs dividing the trace into
% consecutive sets of Size messages; the last may
% be shorter. Size is a number of messages, a
% positive integer; any other value is an error.
trace_frames(Size, Frames) :-
    must_be(positive_integer, Size),
    message_space(Space),
    frames(Space, Size, 1, Frames).

frames([], _, _, []).
frames([N | Ns], Size, I, [I-Frame | Frames]) :-
    (   length(Prefix, Size),
        append(Prefix, Suffix, [N | Ns])
    ->  Frame = Prefix,
        Rest = Suffix
    ;   Frame = [N | Ns],
        Rest = []
    ),
    I1 is I + 1,
    frames(Rest, Size, I1, Frames).

% trace_frame(Size, I, Frame): Frame is frame I.
trace_frame(Size, I, Frame) :-
    trace_frames(Size, Frames),
    memberchk(I-Frame, Frames).


?- trace_frames(10, _F), length(_F, L).
L = 8.

?- trace_frame(10, 8, F).
F = [71].
```

```
?- trace_frame(10, 2, _F), print_messages(_F).
11  1.002 100 appserver    Request 1 received: GET
12  1.003 101 appserver    Request 1 dispatched to
13  1.004 101 dbclient     Query start: SELECT rep
14  1.048 101 dbclient     Query end: 1 row, 44 ms
15  1.049 101 reportengine Build start: report 17
16  1.310 101 reportengine Build end: report 17, 2
17  1.311 101 appserver    Request 1 completed: 20
18  1.500 100 appserver    Request 2 received: GET
19  1.501 102 appserver    Request 2 dispatched to
20  1.502 102 dbclient     Query start: SELECT rep
true.
```

The if-then-else in `frames/4` tries to take exactly `Size` elements, `length/2` with the length bound making a list of fresh variables and `append/3` matching it against the front of the list, and takes the remainder as the last frame when fewer are left. Each frame is a message set, so every predicate of this chapter applies to a frame as to the trace, and the per-frame results are the sequence the pattern is about; Section 24.21 prints one. Frames by time, windows of a fixed number of seconds, belong to Chapter 25, where the measurement taken per frame is the number of messages itself.

24.9.1 A boundary check: the frame size

The check at the head of `trace_frames/2` exists because `frames/4` does not terminate on a frame size of zero: `length(Prefix, 0)` makes the empty prefix, `append/3` leaves the whole list as the suffix, and the recursion is called again on the same list, forever. A size of zero, a negative size, or a fractional one is not a value the relation is defined on, and by the convention of Chapter 7 that is an error about the caller, not a failure about the trace. The two boundary cases, and a size larger than the trace:

```
?- catch(trace_frames(0, _), error(E, _), true).
E = type_error(positive_integer, 0).

?- catch(trace_frames(2.5, _), error(E, _), true).
E = type_error(positive_integer, 2.5).

?- trace_frames(100, _F), length(_F, L).
L = 1.
```

SWI-Prolog 9.0.4 reports zero as a type error rather than a domain error, because `must_be/2` treats `positive_integer` as a type, as the transcript of Chapter 7 showed; the error term is the same one `trace_frame/3` raises, since it calls `trace_frames/2`. A size larger than the trace gives one frame holding every message, which is the whole trace as a single frame and is correct.

24.10 Focus of Tracing

A trace is recorded with a **Focus of Tracing**: the processes, components, or verbosity levels the tracing was configured to observe, which fixes what the trace can show. Within it, the analyst narrows a focus of analysis, a thread, a component, an interval, relative to which the rest of the trace

is context. Absence of messages outside the recorded focus is not evidence of inactivity, and the chosen focus of analysis decides which absences within it are worth noticing.

The analyst's focus is a fact, and a message is in focus when it satisfies it:

```
% focus(Focus): the current focus of analysis:
% thread(TID), component(C) or interval(From, To).
focus(thread(102)).

% in_focus(N): message N is within the focus.
in_focus(N) :-
    focus(Focus),
    focus_member(Focus, N).

focus_member(thread(TID), N) :-
    msg(N, _, _, TID, _, _).
focus_member(component(C), N) :-
    msg(N, _, _, _, C, _).
focus_member(interval(From, To), N) :-
    msg(N, T, _, _, _, _),
    From =< T, T =< To.

% focus_of_tracing(Set): Set is the message set
% within the focus.
focus_of_tracing(Set) :-
    findall(N, in_focus(N), Ns),
    sort(Ns, Set).

?- focus_of_tracing(_S), print_messages(_S).
 8  0.215 102 appserver    Worker 102 ready
19  1.501 102 appserver    Request 2 dispatched to
20  1.502 102 dbclient     Query start: SELECT rep
21  1.550 102 dbclient     Query end: 1 row, 48 ms
22  1.551 102 reportengine Build start: report 18
31  2.500 102 reportengine Warning: table resize,
32  3.000 102 reportengine Warning: table resize,
33  3.500 102 reportengine Warning: table resize,
34  4.000 102 reportengine Warning: table resize,
35  4.500 102 reportengine Warning: table resize,
36  5.000 102 reportengine Warning: table resize,
true.
```

With the focus on worker 102, the trace is eleven messages: a readiness, a dispatch, a query, the start of a build, and six warnings half a second apart about a table that doubles each time, after which the thread is silent. That is all the trace shows about the thread whose CPU time Part III found at 68 percent of the process total; the focus makes the eleven messages legible and makes clear how little there is.

The recorded focus is not in the knowledge base. Nothing in `trace1.pl` says which components had tracing enabled or at what level; facts such as `traced_component/1` would record it, and the analyst adds them from the tracing configuration, the paratext of Section 24.22. The predicate formalizes the analyst's focus and leaves the recorded one as the frame within which every pattern of Part IV is read.

24.11 Trace Mask

A **Trace Mask** hides messages to make the rest visible: those of a component whose output is routine, of a thread that is not under study, or of a kind that repeats, as a mask in image processing suppresses regions so that others stand out. The masked trace is read as if the hidden messages were not there, and the mask is lifted when a hidden message is needed.

A mask is the complement of a message set, and the simplest mask hides components:

```
% trace_mask(Hidden, Visible): Visible is the set
% of messages not written by a component in the
% list Hidden.
trace_mask(Hidden, Visible) :-
    findall(N,
            ( msg(N, _, _, _, C, _),
              \+ memberchk(C, Hidden) ),
            Ns),
    sort(Ns, Visible).

?- trace_mask([reportengine], _V), length(_V, L).
L = 60.

?- trace_mask([reportengine], _V),
   trace_frame(10, 4, _F),
   ord_intersection(_F, _V, _R),
   print_messages(_R).
37  5.001 100 appserver    Request 5 received: GET
38  5.002 100 appserver    No idle worker, request
39  6.000 100 appserver    Request 6 received: GET
40  6.001 100 appserver    No idle worker, request
true.

?- text_set("Warning", _W), message_space(_S),
   ord_subtract(_S, _W, _V), length(_V, L).
L = 65.
```

With the report engine masked, the fourth frame shrinks from ten messages to four, and those four are the event the warnings were burying: requests arriving and finding no idle worker. A mask by text is the same operation with `text_set/2` and `ord_subtract/3`, as the last query shows. The mask is a set operation and composes with everything else; its limit is that a masked message is gone from every measurement taken on the visible set, so a count or density computed under a mask describes the mask as much as the trace, and the mask must be stated beside the figure.

24.12 Trace Extension

A **Trace Extension** adds messages to the trace from outside it: a later recording that continues where this one stopped, a recording of another process or machine over the same interval, or the same interval traced again at a higher verbosity. The pattern is the recognition that the trace in hand is

not the only possible one and that the question it cannot answer may be answered by a longer or wider recording; the extended trace is then analyzed as one.

In the knowledge base an extension is more facts of the `msg/6` shape. The messages below continue the trace past its tail, numbered after it; they are not part of `trace1.pl` and stand under their own predicate name so that the shared file's predicate is not redefined, which Chapter 13 forbade:

```
% extension_msg(N, Time, PID, TID, Component,
% Text): messages recorded after trace1 ends.
extension_msg(72, 14.011, 4120, 100, appserver,
    "Worker pool stopping").
extension_msg(73, 14.512, 4120, 100, dbclient,
    "Disconnected from db01:5432, session=71").
extension_msg(74, 14.513, 4120, 100, appserver,
    "Service stopped").

% trace_extension(Added): Added is the set of
% extension messages numbered after the trace.
trace_extension(Added) :-
    message_space(Space),
    tail_message(Space, Last),
    findall(N,
            ( extension_msg(N, _, _, _, _, _),
              N > Last ),
            Ns),
    sort(Ns, Added).

% extended_span(First, Last): the time span of the
% trace together with its extension.
extended_span(First, Last) :-
    aggregate_all(min(T), msg(_, T, _, _, _, _),
                  First),
    aggregate_all(max(T), any_msg(T), Last).

any_msg(T) :- msg(_, T, _, _, _, _).
any_msg(T) :- extension_msg(_, T, _, _, _, _).

?- trace_extension(A).
A = [72, 73, 74].

?- extended_span(First, Last).
First = 0.0,
Last = 14.513.
```

The extension shows what the original tail did not: the shutdown completed, and the database session the trace followed from message 5 was closed half a second later. In practice an extension is loaded as further `msg/6` facts from a second file with the numbering continued, and every predicate of Part IV sees the extended trace at once because every one refers to `msg/6`; the separate predicate here exists only to keep the shared file intact. What no predicate checks is whether the extension belongs to the same recording, the same process, clock, and numbering; the analyst establishes that before the two are joined, and Chapter 28 treats traces from different sources under Split Trace and Master Trace.

24.13 Truncated Trace

A **Truncated Trace** has lost its beginning or its end: the recording started after the events of interest began, or stopped before they ended, because of a size limit, a circular buffer, or a tracing session started late or stopped early. Its recognition is by the edges: a head message that is not a beginning, or a tail message that is not an end, measured against what a complete trace of the system is known to start and finish with.

The knowledge of how a complete trace begins and ends is two facts, and the recognizer compares the edges of a message set with them:

```
% start_marker(Prefix), end_marker(Prefix): the
% text a complete trace begins and ends with.
start_marker("Service starting").
end_marker("Service stopping").

% marked(N, Marker): the text of message N begins
% with the prefix given by the fact Marker/1.
marked(N, Marker) :-
    msg(N, _, _, _, _, Text),
    call(Marker, Prefix),
    sub_string(Text, 0, _, _, Prefix).

% truncated_trace(Set, Where): the message set
% Set, read as a trace, is cut at Where (start or
% end).
truncated_trace(Set, start) :-
    head_message(Set, N),
    \+ marked(N, start_marker).
truncated_trace(Set, end) :-
    tail_message(Set, N),
    \+ marked(N, end_marker).

?- message_space(_S), truncated_trace(_S, W).
false.

?- trace_frame(20, 2, _F), truncated_trace(_F, W).
W = start ;
W = end.

?- trace_frame(20, 1, _F), truncated_trace(_F, W).
W = end.
```

The whole trace is not truncated by this test. The second frame of twenty, taken as a trace of its own, is truncated at both ends, and the first frame only at its end, which is what a log cut at a size limit after the service started looks like. `call/2` applies the marker predicate's name to get the prefix, so that further markers can be added as facts without touching the recognizer; `sub_string/5` with the first position bound to 0 is the prefix test of Chapter 8.

The test is as good as the markers. A system that writes no start or end message cannot be judged by them, and a trace cut in the middle of an activity with its markers in place is truncated in a sense

the markers do not see. Chapter 25 adds a second test under **Last Activity** (page 532), the threads whose tail is a start message, which Section 24.8 already pointed at.

24.14 Sparse Trace

A **Sparse Trace** has few messages for the time it covers, typically because tracing was enabled at a low verbosity or for few components. The patterns that depend on the fine structure of a thread's activity have little to work with in it, and the analyst's first conclusion from a sparse trace is often that another one must be recorded.

Sparseness is a rate, messages per second over the span of a set, below a threshold. The threshold is a fact, following Chapter 17:

```
% threshold(Name, Value): the thresholds of this
% chapter.
threshold(sparse_trace, 1.0).

% message_rate(Set, Rate): Rate is the number of
% messages of Set per second of its span.
message_rate(Set, Rate) :-
    head_message(Set, First),
    tail_message(Set, Last),
    msg(First, T1, _, _, _, _),
    msg(Last, T2, _, _, _, _),
    T2 > T1,
    length(Set, Count),
    Rate is Count / (T2 - T1).

% sparse_trace(Set): the rate of Set is below the
% sparse_trace threshold.
sparse_trace(Set) :-
    message_rate(Set, Rate),
    threshold(sparse_trace, Min),
    Rate < Min.

?- message_space(_S), message_rate(_S, _R),
   format("~2f~n", [_R]).
5.07
true.

?- message_space(_S), sparse_trace(_S).
false.

?- text_set("Error", _E), message_rate(_E, _R),
   format("~2f~n", [_R]).
0.57
true.

?- text_set("Error", _E), sparse_trace(_E).
true.
```

Five messages per second is not sparse at a threshold of one per second. The set of error messages, read on its own, has four messages over seven seconds and is: it is what the same system would

have left in a log recorded at error level only, four lines saying that a queue overflowed and a copy failed, without the request lifecycle that explains why. `message_rate/2` fails for a set of one message or of messages at the same instant, where a rate is undefined, so `sparse_trace/1` fails rather than divides by zero.

The threshold carries the judgment. One message per second is sparse for an application server and dense for a daily batch job, and the fact exists so that the analyst sets it per system; the predicate knows only that a rate is below a number.

24.15 Visibility Limit

The **Visibility Limit** of a trace is the boundary of what tracing could observe at all: the moment it started, the components that had trace statements, the verbosity that was enabled, the processes that were included. Beyond the limit the trace is silent not because nothing happened but because nothing could be recorded, and an inference from silence is only as good as the analyst's knowledge of where the limit lies.

`trace1.pl` and `dump1.pl` describe the same process, and the dump sees threads the trace does not:

```
:- [dump1].

?- thread(T, _), \+ msg(_, _, _, T, _, _).
T = 105 ;
T = 106 ;
T = 107 ;
false.

?- msg(_, _, _, T, _, _), \+ thread(T, _).
false.
```

Threads 105, 106, and 107 exist in the dump and wrote nothing to the trace. For the two idle pool threads that is what idleness looks like; for thread 107, which has the deepest stack trace and the second largest CPU time in Part III, it means the code it was running has no trace statements, and the trace's silence about it is a limit of visibility, not a fact about the thread. The second query is the converse check, and it holds. The pattern has no predicate of its own because the limit is not in the trace; what a program can do is compare the trace with another artifact and mark the differences as the places where the trace does not see. Chapter 28 returns to the pair under **Small DA+TA** (page 622).

24.16 Empty Trace

An **Empty Trace** has no messages, or none from the component or process the analysis is about. It is a finding, not a failure to find: tracing was not enabled, the component never ran the code that writes messages, or the trace was taken from the wrong process. The question an empty trace answers is which of these it was.

For a message set, emptiness is a test against `[]`; for the whole knowledge base, it is negation as failure over `msg/6`:

```
% empty_trace(Set): the message set Set is empty.
empty_trace([]).

?- message_set(msg(_, _, _, _, netlib, _), S),
   empty_trace(S).
S = [].

?- message_set(msg(_, _, _, 107, _, _), S),
   empty_trace(S).
S = [].

?- \+ msg(_, _, _, _, _, _).
false.
```

There is no component `netlib` and thread 107 wrote nothing, so both sets are empty traces in the restricted sense; the trace as a whole is not. The caution is the closed-world one of Chapter 17: `\+ msg(_, _, _, _, _, _)` succeeds when no `msg/6` fact is loaded, and cannot tell a trace with no messages from a trace file that failed to load. A program that reports an empty trace should first establish, with `source_file/1`, that it consulted the file it meant to.

24.17 Singleton Event

A **Singleton Event** is a message of a kind that occurs exactly once in the trace. Routine activity repeats; an exception, a configuration load, a shutdown happen once, and the messages that happen once are therefore where an analyst looks for the events that matter, as the rare words of the vocabulary index pointed at them in Chapter 9.

Whether two messages are of the same kind is a decision about their text. Two completions with different request numbers and durations are the same event here, so a **message kind** is defined as the text with every number replaced by `#`; Chapter 27 treats the constant part of a message as the **Message Invariant** (page 567) pattern, and the kind is a helper in its service. The replacement is a grammar over character codes with `library(dcg/basics)`: a number is a hexadecimal literal after `0x`, read by `xinteger//1`, or a decimal one, read by `integer//1`, and anything else is copied. The if-then-else in the grammar body commits to the first alternative that applies, so the parse is deterministic:

```
% message_kind(N, Kind): Kind is the text of
% message N with every number replaced by #.
:- table message_kind/2.

message_kind(N, Kind) :-
    msg(N, _, _, _, _, Text),
    string_codes(Text, Codes),
    phrase(kind(KindCodes), Codes),
    string_codes(Kind, KindCodes).
```

```
kind(Ks) -->
    (   number_token
    ->  { Ks = [0'# | Ks1] },
        kind(Ks1)
    ;   [C]
    ->  { Ks = [C | Ks1] },
        kind(Ks1)
    ;   { Ks = [] }
    ).

number_token -->
    (   "0x"
    ->  xinteger(_)
    ;   integer(_)
    ).

% kind_messages(Kind, Set): Set is the non-empty
% set of messages of kind Kind.
kind_messages(Kind, Set) :-
    aggregate_all(set(N), message_kind(N, Kind),
                  Set),
    Set \== [].

% singleton_event(N): message N is the only one of
% its kind.
singleton_event(N) :-
    message_kind(N, Kind),
    kind_messages(Kind, [N]).


?- message_kind(58, K).
K = "Request # completed: #, # ms".

?- message_kind(69, K).
K = "Error: buffer copy failed, code=#".

?- kind_messages("Query end: # row, # ms", S).
S = [14, 21, 55].

?- aggregate_all(count, singleton_event(_), N).
N = 20.

?- aggregate_all(set(N),
   ( singleton_event(N), N > 40 ), _S),
   print_messages(_S).
48  9.002 108 appserver    Housekeeping: watchdog,
51  9.004 108 appserver    Housekeeping: worker 10
52  9.005 108 appserver    Housekeeping: worker 10
67 14.000 109 appserver    Request 10 received: GE
68 14.001 109 dbclient     Receive start: session=
69 14.002 109 dbclient     Error: buffer copy fail
70 14.003 109 appserver    Unhandled exception 0xc
71 14.010 100 appserver    Service stopping
true.
```

The `table/1` directive memoizes `message_kind/2`, as Chapter 11 described: the grammar runs once per message however often a kind is asked for, and `kind_messages/2`, which scans every message on each call, costs a table lookup per message instead of a parse. Twenty of the seventy-one

messages are singletons; the eight after message 40 are the watchdog's report, the whole of thread 109's failure, and the shutdown. The error of message 69 and the exception of message 70 are singletons in the trace as in the dump, where Part III found the same exception on the same thread.

The kind abstraction decides what counts as the same event, and it decides crudely. `Request # received: GET /session/#` and `Request # received: GET /session/#/data` are two kinds, though an analyst would call them one, and a message whose variable part is a word rather than a number, a user name or a file name, yields one kind per value and therefore one singleton per message. Refining the grammar, by replacing path segments or quoted strings as well, changes which messages are singletons without touching the recognizer.

24.18 Message Directory

A **Message Directory** is the listing of the distinct kinds of message a system can write, with where and how often each occurs: the equivalent, for a trace, of the table of contents of a book. Compiled from source code it is complete and names the messages that never appear in a given trace; compiled from the trace it is the part of the directory that was exercised, and it is the first thing an analyst builds to learn what a system reports about itself.

From the trace, the directory is the kinds grouped with their messages, which is the vocabulary index of Section 24.4 with kinds in place of words, with one difference that matters. `message_kind/2` is tabled, and the answers of a tabled predicate are read from its table, not produced by its clauses: SWI-Prolog yields them in the order of the table's internal structure, which Chapter 11 said is implementation-defined, differs between runs, and must not be relied on. `findall/3` therefore collects the `Kind-N` pairs in an order that is not the message order within a kind, and `keysort/2`, being stable, would keep that order inside each group, so that the head of a group would be whichever message of the kind the table happened to yield first. The directory sorts the complete pairs with `sort/2` instead: pairs with equal kinds are ordered by their second argument, the message number, so the head of every group is the earliest message of its kind. `sort/2` also removes duplicates, which here removes nothing, because the message number is the primary key and no two pairs are equal.

```
% message_directory(Directory): Directory is the
% list of Kind-Set pairs, one per message kind, in
% the standard order of the kinds; each Set is in
% message order.
message_directory(Directory) :-
    findall(Kind-N, message_kind(N, Kind), Pairs),
    sort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Directory).
```

```
% print_directory(Directory): one line per kind
% with its count and first occurrence.
print_directory(Directory) :-
    forall(member(Kind-Ns, Directory),
           ( length(Ns, Count),
             Ns = [First | _],
             truncated(Kind, 40, Short),
             format("~t~d~3| ~t~d~7| ~s~n",
                    [Count, First, Short]) )).

?- message_directory(_D), length(_D, L).
L = 33.

?- message_directory(_D),
   include([_-[_, _ | _]]>>true, _D, _Repeated),
   print_directory(_Repeated).
  2  16 Build end: report #, # nodes, # ms
  3  15 Build start: report #
  3  43 Error: queue depth # exceeds threshold #
  2  49 Housekeeping: worker # wait # ms
  3  38 No idle worker, request # queued
  3  14 Query end: # row, # ms
  3  13 Query start: SELECT report #
  6  17 Request # completed: #, # ms
  9  12 Request # dispatched to worker #
  4  39 Request # received: GET /health
  3  11 Request # received: GET /report/#
  6  31 Warning: table resize, # -> #
  4   7 Worker # ready
true.
```

Thirty-three kinds; the thirteen that repeat are printed, and the twenty that do not are the singletons of the previous section. `include/3` with a yall lambda of Chapter 10 keeps the pairs whose set matches `[_, _ | _]`. Read as a directory, the thirteen lines are the system's routine: the request lifecycle in six kinds, the worker readiness, the resize warning, the queue error, the watchdog's wait report. The second column locates where each routine begins, the workers ready at message 7, the first request at 11 and its lifecycle in the messages that follow it, the queueing at 38 and the error it leads to at 43. Three query starts against two build ends, nine dispatches against six completions: the counts already say that something started and did not finish, and kinds whose counts should pair but do not are where Chapter 27 looks for **Missing Message** (page 575) instances.

24.18.1 A boundary check: first occurrences

The difference between the two sorts is visible on a list of four pairs, without tabling: `keysort/2` orders by key and leaves the pairs under each key as they were, and `sort/2` orders the pairs as terms, by key and then by value, in the standard order of terms:

```
?- keysort([b-2, a-3, b-1, a-1], S).
S = [a-3, a-1, b-2, b-1].

?- sort([b-2, a-3, b-1, a-1], S).
S = [a-1, a-3, b-1, b-2].
```

A directory built with the first would report 3 as the first occurrence of `a`. The check that the directory does not is that the head of every group equals the smallest message number of its kind, computed independently by `aggregate_all/3`:

```
?- message_directory(_D),
   forall(member(K-[F | _], _D),
          aggregate_all(min(N),
                        message_kind(N, K), F)).
true.
```

The check holds for all thirty-three kinds. It is the form every grouping in this chapter should be read against: `trace_partition/2` and `trace_viewpoint/3` use `keysort/2` and are correct because their pairs come from `msg/6` in clause order, which is message order in `trace1.pl`, and from a sorted set; a grouping whose pairs come from a tabled predicate, or from any source whose order is not guaranteed, must sort the pairs.

The directory from the trace is a lower bound on the directory from the source. A kind absent from it was either never exercised or does not exist, and the trace cannot tell which; the complete directory is paratext in the sense of Section 24.22, and a knowledge base that had it as facts, one per trace statement, could subtract the trace's directory from it and name the statements that never fired.

24.19 Trace Summary

A **Trace Summary** is the condensed account of a trace that a tool prints before the analyst reads a line: the basic facts, the partitions by process, thread, and component with their sizes, the number of distinct message kinds, and the counts of what is exceptional, errors and singletons. It is the report that decides whether the trace is worth reading at all and where to start.

The summary composes the predicates of this chapter into one report:

```
% trace_summary: print a summary of the trace.
trace_summary :-
    print_basic_facts,
    forall(dimension(Dim),
           ( trace_partition(Dim, Partition),
             format("by ~a~n", [Dim]),
             print_partition(Partition) )),
    message_directory(Directory),
    length(Directory, Kinds),
    aggregate_all(count, singleton_event(_),
                  Singletons),
    text_set("Error", Errors),
    length(Errors, ErrorCount),
    format("kinds~t~12|~d~n", [Kinds]),
    format("singletons~t~12|~d~n", [Singletons]),
    format("errors~t~12|~d ~w~n",
           [ErrorCount, Errors]).
```

```
?- trace_summary.
messages    71
first       0.000
last        14.010
duration    14.010
processes   1
threads     7
components  4
by process
4120            71    1   71
by thread
100             22    1   71
101             21    7   66
102             11    8   36
103              4    9   26
104              4   10   30
108              5   48   52
109              4   67   70
by component
appserver       47    1   71
cachelib         1    3    3
dbclient        12    4   69
reportengine    11   15   57
kinds       33
singletons  20
errors      4 [43,45,47,69]
true.
```

Twenty-five lines summarize seventy-one messages, and an analyst who knows the system reads the diagnosis in outline from them: two workers that stopped early, a thread that appeared only at the end, errors clustered in two places. The summary has the shape of the report Chapter 23 wrote for the dump, and the two are the halves of the **Small DA+TA** (page 622) account of Chapter 28. What it omits is deliberate: it names no pattern instances but errors and singletons, because a summary that reported every pattern would be an analysis, and the analyst's reading of the summary decides which analysis to run.

24.20 Trace Viewpoints

The same messages can be arranged by thread, by component, by request, or by time, and each arrangement is one of the **Trace Viewpoints** on the trace: the messages do not change, their adjacency does, and what is adjacent is what is read together. A region that is confusing in time order, with four threads interleaved, is often clear in thread order, and the patterns of Chapter 26 are largely patterns of one viewpoint or another.

A viewpoint on a message set is the partition of Section 24.5 restricted to the set:

```
% trace_viewpoint(Dim, Set, View): View is the
% list of Value-Subset pairs dividing the message
% set Set by its values along Dim.
trace_viewpoint(Dim, Set, View) :-
    findall(V-N,
            ( member(N, Set),
              dimension_value(Dim, N, V) ),
            Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, View).

% print_view(View): each part under its value.
print_view(View) :-
    forall(member(V-Ns, View),
           ( format("~w~n", [V]),
             print_messages(Ns) )).

?- trace_frame(10, 3, _F),
   trace_viewpoint(thread, _F, _V),
   print_view(_V).
100
23  2.000 100 appserver    Request 3 received: POS
27  2.100 100 appserver    Request 4 received: GET
102
21  1.550 102 dbclient     Query end: 1 row, 48 ms
22  1.551 102 reportengine Build start: report 18
103
24  2.001 103 appserver    Request 3 dispatched to
25  2.002 103 appserver    Session 9 commit start
26  2.003 103 dbclient     Send start: session=71,
104
28  2.101 104 appserver    Request 4 dispatched to
29  2.102 104 dbclient     Reply received: session
30  2.103 104 appserver    Session 9 lookup start
true.

?- trace_frame(10, 3, _F),
   trace_viewpoint(component, _F, _V),
   print_view(_V).
appserver
23  2.000 100 appserver    Request 3 received: POS
24  2.001 103 appserver    Request 3 dispatched to
25  2.002 103 appserver    Session 9 commit start
27  2.100 100 appserver    Request 4 received: GET
28  2.101 104 appserver    Request 4 dispatched to
30  2.103 104 appserver    Session 9 lookup start
dbclient
21  1.550 102 dbclient     Query end: 1 row, 48 ms
26  2.003 103 dbclient     Send start: session=71,
29  2.102 104 dbclient     Reply received: session
reportengine
22  1.551 102 reportengine Build start: report 18
true.
```

The third frame holds the moment the deadlock of Part III was set up, and the two viewpoints show it differently. By thread, worker 103 begins a commit of session 9 and sends on the database session, and worker 104 begins a lookup of the same session 9 after receiving a reply on the same database

session: the two threads of the dump's **Deadlock (Critical Sections)** (page 388) instance are seen taking the operations that led there. By component, the database client's three messages are a send and a reply on one session from two threads, the **Data Flow** (page 559) pattern of Chapter 26 in outline. Neither viewpoint adds information; each makes a different adjacency legible. A viewpoint by request, which no dimension of the schema carries, needs the request number extracted from the text, which Chapter 26 does for the **Adjoint Thread of Activity** (page 538).

24.21 Working Set

The **Working Set** of a trace region is the set of components, or threads, that have messages in it, by analogy with the working set of a process in memory: the parts of the system that are active during the window. Followed frame by frame, the working set shows which components come into play and which drop out, and a change of working set is a change of phase.

```
% working_set(Set, Components): Components is the
% set of components with a message in Set.
working_set(Set, Components) :-
    aggregate_all(set(C),
                  ( member(N, Set),
                    msg(N, _, _, _, C, _) ),
                  Components).

% print_working_sets(Frames): one line per frame
% with its message range and working set.
print_working_sets(Frames) :-
    forall(member(Frame, Frames),
           print_working_set(Frame)).

print_working_set(I-Set) :-
    working_set(Set, Cs),
    Set = [First | _],
    last(Set, Last),
    atomic_list_concat(Cs, ' ', Line),
    format("~t~d~2| ~t~d~6|-~d~t~13|~a~n",
           [I, First, Last, Line]).

?- trace_frames(10, _F), print_working_sets(_F).
 1   1-10    appserver cachelib dbclient
 2  11-20    appserver dbclient reportengine
 3  21-30    appserver dbclient reportengine
 4  31-40    appserver reportengine
 5  41-50    appserver
 6  51-60    appserver dbclient reportengine
 7  61-70    appserver dbclient
 8  71-71    appserver
true.

?- trace_frame(10, 5, _F), working_set(_F, W).
W = [appserver].
```

The cache library is in the working set of the first frame only, at startup; the database client and the report engine enter with the first request and leave in the fourth and fifth frames, when the server

is doing nothing but queueing requests and reporting on itself, and return when the queue drains in the sixth. The fifth frame's working set of one component is the quietest phase of the trace in this measure, and the phase in which the dump of Part III would have been taken. A working set is a set of atoms, so the ordered-set operations compare working sets of different frames directly: `ord_subtract/3` between consecutive frames gives the components that dropped out.

24.22 Paratext

Everything that accompanies a trace without being part of it is its **Paratext**, a name the memory analysis catalog took from the literary term for the material around a text and which applies to traces unchanged: the file name and its date, the header a tracing tool writes, the comments in a saved log, the configuration that fixed the focus of tracing, and the problem description that came with the trace. The patterns of Part IV are defined over the messages, but what the messages mean is decided with the paratext in hand, and much of what this chapter left to the analyst is paratext: the problem description the basic facts are compared with, the configuration that fixes the visibility limit, the source code from which a complete message directory would come.

`trace1.pl` has a paratext of its own: its first lines are comments stating that the trace is synthetic and giving the schema, which a reader of the facts alone would not know and the program cannot consult. A knowledge base that records paratext does so as facts, `trace_info(origin, synthetic)`, `problem_description("Service became unresponsive")`, which the predicates of this and the following chapters can compare with what they find. Chapter 15 introduced the same discipline for dumps under provenance; for traces it is the step from the summary of Section 24.19 to a report an analyst can hand to someone else.

Chapter 25. Time

Chapter 24 treated the trace as a set of messages with coordinates. This chapter treats the one coordinate that is ordered rather than grouped: time. The patterns here are about the gaps between messages and what they mean, about how many messages arrive per second and how that rate changes, about regularity and its absence, and about the intervals in which nothing was recorded. Chapter 16 made the message number the primary order of the schema and the timestamp a secondary one, because the number is reliable where clocks are not; this chapter is where the timestamp earns its place, since a gap of five seconds between two consecutive messages is a fact about the system that no ordinal can express.

Times in `trace1.pl` are floats in seconds. Arithmetic on floats produces results such as `4.691000000000001`, which are correct and unreadable, so this chapter converts every time to an integer number of milliseconds with `round/1` before computing with it; differences and periods are then exact integers, and the unit is the one in which the messages themselves report durations. The helpers below are those of Chapter 24 together with the conversion, a thread's message set by `bagof/3`, and the two ends of the trace:

```
:- [trace1].
:- use_module(library(dcg/basics)).

truncated(Text, Width, Short) :-
    string_length(Text, Len),
    (   Len =< Width
    ->  Short = Text
    ;   sub_string(Text, 0, Width, _, Short)
    ).

print_row(N) :-
    msg(N, Time, _, TID, Component, Text),
    truncated(Text, 23, Short),
    format("~t~d~2| ~t~3f~9| ~d ~a~t~26| ~s~n",
           [N, Time, TID, Component, Short]).

print_messages(Ns) :-
    forall(member(N, Ns), print_row(N)).

% time_ms(N, Ms): message N was written Ms whole
% milliseconds after the trace start.
time_ms(N, Ms) :-
    msg(N, Time, _, _, _, _),
    Ms is round(Time * 1000).

% thread_messages(TID, Set): Set is the message
% set of thread TID.
thread_messages(TID, Set) :-
    bagof(N, T^P^C^X^msg(N, T, P, TID, C, X), Ns),
    sort(Ns, Set).
```

```
% trace_start(Ms), trace_end(Ms): the times of the
% first and last message.
trace_start(Start) :-
    aggregate_all(min(Ms), time_ms(_, Ms), Start).
trace_end(End) :-
    aggregate_all(max(Ms), time_ms(_, Ms), End).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- time_ms(67, Ms).
Ms = 14000.

?- thread_messages(103, S).
S = [9, 24, 25, 26].

?- trace_start(S), trace_end(E).
S = 0,
E = 14010.
```

`thread_messages/2` uses `bagof/3` rather than `findall/3` so that it has the two modes a recognizer needs: with the thread bound it collects that thread's messages once, and with it unbound it enumerates the threads in order, one message set each, because the thread is the goal's only free variable. The thresholds of this chapter follow the convention of Chapter 17, `threshold/2` facts, and are collected in one place, in milliseconds, so that the analyst adjusts them without touching a recognizer:

```
% threshold(Name, Value): the thresholds of this
% chapter, in milliseconds unless stated.
threshold(discontinuity, 1000).
threshold(correlation, 100).
threshold(period_tolerance, 10).
threshold(min_period, 100).
threshold(timeout_tolerance, 5).
threshold(burst_gap, 500).
threshold(no_activity, 2000).
threshold(blackout, 2000).
```

25.1 Time Delta

The **Time Delta** between two consecutive messages is the time that passed between them, and the sequence of deltas along a trace, or along one thread's messages, is where the time went: a delta of one millisecond is the system doing one thing after another, and a delta of seven seconds is the system waiting for something, or the thread being descheduled, or nothing at all being recorded. The pattern is the measurement; the patterns that follow are readings of it.

Two forms are needed, one over the trace in message order and one over a message set, since the messages adjacent in a thread's set are not adjacent in the trace. `nextto/3` of `library(lists)` relates two adjacent elements of a list and enumerates every adjacent pair when both are unbound:

```
% time_delta(N, Next, Delta): Next follows N in
% the trace, Delta milliseconds later.
time_delta(N, Next, Delta) :-
    time_ms(N, T1),
    Next is N + 1,
    time_ms(Next, T2),
    Delta is T2 - T1.

% time_delta(Set, N, Next, Delta): Next follows N
% in the message set Set, Delta ms later.
time_delta(Set, N, Next, Delta) :-
    nextto(N, Next, Set),
    time_ms(N, T1),
    time_ms(Next, T2),
    Delta is T2 - T1.

% print_deltas(Set): one line per adjacent pair of
% Set with the delta and the later message's text.
print_deltas(Set) :-
    forall(time_delta(Set, N, Next, Delta),
           print_delta(N, Next, Delta)).

print_delta(N, Next, Delta) :-
    msg(Next, _, _, _, _, Text),
    truncated(Text, 28, Short),
    format("~t~d~2|-~d~t~6| ~t~d~12| ~s~n",
           [N, Next, Delta, Short]).

?- time_delta(66, Next, D).
Next = 67,
D = 4691.

?- aggregate_all(max(D, N), time_delta(N, _, D),
   M).
M = max(4691, 66).

?- thread_messages(101, _S), print_deltas(_S).
 7-12    788 Request 1 dispatched to work
12-13      1 Query start: SELECT report 1
13-14     44 Query end: 1 row, 44 ms
14-15      1 Build start: report 17
15-16    261 Build end: report 17, 2611 n
16-17      1 Request 1 completed: 200, 30
17-53   7695 Request 5 dispatched to work
53-54      1 Query start: SELECT report 1
54-55     53 Query end: 1 row, 53 ms
55-56      1 Build start: report 19
56-57    239 Build end: report 19, 2402 n
57-58      1 Request 5 completed: 200, 43
58-59      1 Request 6 dispatched to work
59-60      1 Request 6 completed: 200, 33
60-61      1 Request 7 dispatched to work
61-62      1 Request 7 completed: 200, 23
62-63      1 Request 8 dispatched to work
63-64      1 Request 8 completed: 200, 13
64-65      1 Request 9 dispatched to work
65-66      1 Request 9 completed: 200, 30
true.
```

The deltas of worker 101 are a profile of its work. The query took 44 and 53 milliseconds and the report build 261 and 239, which agree with the durations the messages themselves report; the one-millisecond deltas are the bookkeeping between steps; and the two large deltas are the worker idle, 788 milliseconds from readiness to the first request and 7695 from the completion of request 1 to the dispatch of request 5. The largest delta in the whole trace, 4691 milliseconds after message 66, is a different matter, treated in Section 25.17.

The measurement is exact and its meaning is not. A delta between two messages of one thread is the time the thread spent between writing them, which may be work, waiting, or preemption; a delta between two consecutive messages of the trace from different threads says nothing about either thread. The two forms of the predicate keep the two readings apart, and every pattern below says which it uses.

25.2 Discontinuity

A **Discontinuity** is a time delta large enough to break the continuity of the activity it interrupts: a gap between two adjacent messages, of the trace or of a thread, that exceeds what the activity's normal rhythm allows. It is the place in a thread's story where the thread stopped, and the question it raises is what the thread was waiting for.

The recognizer is the time delta compared with the `discontinuity` threshold, in the same two forms:

```
% discontinuity(N, Delta): the gap after message N
% in the trace exceeds the threshold.
discontinuity(N, Delta) :-
    time_delta(N, _, Delta),
    threshold(discontinuity, Min),
    Delta > Min.

% discontinuity(Set, N, Delta): the gap after N
% within the message set Set exceeds the
% threshold.
discontinuity(Set, N, Delta) :-
    time_delta(Set, N, _, Delta),
    threshold(discontinuity, Min),
    Delta > Min.

% thread_discontinuity(TID, N, Delta): a
% discontinuity within the messages of TID.
thread_discontinuity(TID, N, Delta) :-
    thread_messages(TID, Set),
    discontinuity(Set, N, Delta).

?- discontinuity(N, D).
N = 66,
D = 4691 ;
false.
```

```
?- thread_discontinuity(T, N, D).
T = 100,
N = 27,
D = 2901 ;
T = 100,
N = 47,
D = 5009 ;
T = 101,
N = 17,
D = 7695 ;
T = 102,
N = 8,
D = 1286 ;
T = 103,
N = 9,
D = 1785 ;
T = 104,
N = 10,
D = 1885 ;
false.
```

At a threshold of one second the trace as a whole has one discontinuity, after message 66, and the threads have six between them. Three are the workers 102, 103, and 104 idle after startup until their first request, which is a discontinuity only in the formal sense; two are the main thread, silent while the workers were busy and again before the shutdown; and one is worker 101's idle stretch from Section 25.1. The trailing `false.` in each answer is the search for a further instance exhausting the trace.

The threshold decides everything. At 500 milliseconds the main thread's one-second health checks become a discontinuity each, which is noise, and at 5000 only its last silence remains, which loses worker 101; there is no threshold right for every thread, because a thread's normal rhythm is its own, and a refinement the predicate does not make is to take the threshold per thread from the thread's own median delta. What the predicate also cannot say is why. The gap after message 17 is a worker with nothing to do, and the gap after message 47 is the whole system quiet before an exception; the recognizer finds both and distinguishes neither.

25.3 Correlated Discontinuity

When discontinuities in different threads, or in different traces of the same system, begin at the same moment, they are a **Correlated Discontinuity**, and the correlation points to a cause outside any one thread: the process suspended, the machine paused, the tracing stopped, or every thread waiting for the same thing. A single thread's gap is that thread's problem; a correlated gap is the system's.

Two thread discontinuities are correlated when their starting times differ by at most the `correlation` threshold; the standard order on the thread identifiers reports each pair once:

```
% correlated_discontinuity(T1-N1, T2-N2): the
% discontinuities after N1 in thread T1 and after
% N2 in thread T2 begin within the correlation
% threshold of each other.
correlated_discontinuity(T1-N1, T2-N2) :-
    thread_discontinuity(T1, N1, _),
    thread_discontinuity(T2, N2, _),
    T1 @< T2,
    time_ms(N1, S1),
    time_ms(N2, S2),
    threshold(correlation, Max),
    abs(S1 - S2) =< Max.

?- correlated_discontinuity(A, B).
A = 102-8,
B = 103-9 ;
A = 102-8,
B = 104-10 ;
A = 103-9,
B = 104-10 ;
false.
```

The three correlated pairs are one event seen three times: workers 102, 103, and 104 all fell silent within a millisecond of each other at 0.216 seconds, and the cause is in the trace, since the worker pool had just started and no request had arrived. The correlation is real and the reading is benign, which is the first thing to say about the pattern: correlation locates a common cause and does not grade it. The predicate compares starts only; a stronger form would require the gaps to end together as well, and a form over two traces, as the catalog intends for recordings from several machines, needs the `msg/7` convention of Chapter 16 with a trace identifier and a check that the two clocks agree, which Chapter 28 treats under Inter-Correlation.

25.4 Statement Density and Current

Statement Density and Current is one catalog pattern with two measures, and this section gives each an operational definition. The **statement current** is, in this book's definition, the number of messages per unit of time, measured over a window; the name is the catalog's, by analogy with electric current as a rate of flow, and the definition is the one this book uses to compute it. A high current is a busy system or a verbose one, a current of zero is silence, and the profile of the current along the trace is the first picture of where the activity was.

The window is a half-open interval of seconds, so that consecutive windows do not share a message. The bounds of a window are in seconds, the unit of the timestamps, and a window width is a whole number of seconds, which `window/3` checks with `must_be/2` before it computes anything:

```
% window_messages(+From, +To, -Set): Set is the
% set of messages with From =< Time < To; From and
% To are times in seconds, From < To.
window_messages(From, To, Set) :-
    findall(N,
            ( msg(N, T, _, _, _, _),
              T >= From, T < To ),
            Ns),
    sort(Ns, Set).

% statement_current(+From, +To, -Current): Current
% is the number of messages per second in the
% window From..To, in seconds, From < To.
statement_current(From, To, Current) :-
    window_messages(From, To, Set),
    length(Set, Count),
    Current is Count / (To - From).

% window(+Width, -From, -To): the consecutive
% windows of Width seconds covering the trace.
% Width is a positive integer number of seconds;
% From and To are then integers in seconds.
window(Width, From, To) :-
    must_be(positive_integer, Width),
    trace_end(End),
    Last is End // (Width * 1000),
    between(0, Last, I),
    From is I * Width,
    To is From + Width.

% print_current(Width): the current per window, as
% a number and as a bar.
print_current(Width) :-
    forall(window(Width, From, To),
           print_window_current(From, To)).

print_window_current(From, To) :-
    statement_current(From, To, Current),
    Stars is round(Current),
    format("~t~d~3|-~d~t~7| ~t~1f~12| ~*c~n",
           [From, To, Current, Stars, 0'*]).


?- statement_current(9, 10, C).
C = 21.

?- statement_current(10, 14, C).
C = 0.

?- print_current(1).
  0-1   10.0 **********
  1-2   12.0 ************
  2-3    9.0 *********
  3-4    2.0 **
  4-5    2.0 **
  5-6    3.0 ***
  6-7    2.0 **
  7-8    3.0 ***
  8-9    2.0 **
  9-10  21.0 *********************
```

```
 10-11   0.0
 11-12   0.0
 12-13   0.0
 13-14   0.0
 14-15   5.0 *****
true.
```

The profile has four parts: a busy start, with the service coming up and the first requests served; six seconds of low current while the workers were stalled and the main thread queued requests; a burst of twenty-one messages in the tenth second as worker 101 drained the queue; and then four seconds of nothing before the final five messages. `~*c` takes the repeat count from the argument list, as Chapter 6 showed, and draws the bar. The integer division `//` in `window/3` gives the index of the last window that contains the trace end, so the windows cover the trace exactly.

25.4.1 A boundary check: the window width

Integer division is defined on integers only, and `14010 // 500.0` is a type error; without the check, that is where a width of half a second would have failed, inside the arithmetic, with an error about `//` rather than about the width. The check states the contract instead, and a width of zero, which would otherwise divide by zero, is refused with the same error. The two boundary cases, and a count of the windows at a width that does not divide the trace length evenly:

```
?- catch(window(0.5, _, _), error(E, _), true).
E = type_error(positive_integer, 0.5).

?- catch(window(0, _, _), error(E, _), true).
E = type_error(positive_integer, 0).

?- aggregate_all(count, window(5, _, _), C).
C = 3.
```

Three windows of five seconds cover a trace of 14.010 seconds, the last one, from ten to fifteen, only partly filled, and `print_current/1` prints what is in it. The contract is whole seconds because the window bounds are then integers, which `print_window_current/2` prints with `~d` and which tile the trace without rounding; a sub-second window needs the predicate restated in milliseconds, with `time_ms/2` in place of the comparison of floats in `window_messages/3` and a printer that does not assume integer bounds, and this chapter does not need one.

The current depends on the window. A one-second window shows the burst; a five-second window averages it into the stall around it, and a window of ten milliseconds, in a version of the predicate that allowed it, would show every burst of bookkeeping as a spike. The definition does not say what the right window is, and the analyst chooses it for the question, as a physicist chooses the interval over which a current is averaged.

The **statement density** is, in this book's operational definition, the ratio of the number of messages of interest in a region to the total number of messages in that region, where the messages of interest are those of a component, a thread, or a kind. Where the current measures how much was recorded,

the density measures what share of it was the thing being looked for: a density of one is a region in which nothing else was happening, and a density near zero is a region in which the component of interest was a bystander.

The messages of interest are given by a `msg/6` template, as in Chapter 24, and the region by a window:

```
% statement_density(Template, From, To, Density):
% Density is the share of the messages in the
% window that unify with Template.
statement_density(Template, From, To, Density) :-
    window_messages(From, To, Set),
    length(Set, Total),
    Total > 0,
    Template = msg(N, _, _, _, _, _),
    aggregate_all(count,
                  ( member(N, Set), Template ),
                  Count),
    Density is Count / Total.

?- statement_density(
   msg(_, _, _, _, appserver, _), 5, 10, _D),
   format("~2f~n", [_D]).
0.84
true.

?- statement_density(msg(_, _, _, 101, _, _),
   9, 10, _D), format("~2f~n", [_D]).
0.67
true.

?- statement_density(msg(_, _, _, _, _, _),
   10, 11, D).
false.
```

Between five and ten seconds, 84 percent of the messages are the application server's: the stall is a period in which the server talks mostly to itself. In the tenth second, two thirds of the messages are worker 101's, which is the queue draining. The third query fails because the window is empty, and the predicate fails rather than divides by zero; a density over no messages is undefined, not zero, and the failure says so.

The definition's limit is that a share is relative to whatever else was recorded. A component's density rises when other components fall silent as much as when it becomes busier, and the two windows above are examples of both; the density alone does not distinguish them, and the current must be read beside it.

25.5 Density Distribution

The **Density Distribution** of a message set is its density window by window along the trace: the picture of where in time the component or kind of interest concentrated and where it was absent.

A component present at a steady density throughout is background; one whose density is zero everywhere but in one window is an event.

```
% density_distribution(Template, Width, Dist):
% Dist is the list of From-Density pairs for the
% non-empty windows of Width seconds.
density_distribution(Template, Width, Dist) :-
    findall(From-Density,
            window_density(Template, Width, From,
                           Density),
            Dist).

window_density(Template, Width, From, Density) :-
    window(Width, From, To),
    statement_density(Template, From, To,
                      Density).

% print_distribution(Dist): one line per window.
print_distribution(Dist) :-
    forall(member(From-Density, Dist),
           ( Stars is round(20 * Density),
             format("~t~d~3| ~t~2f~9| ~*c~n",
                    [From, Density, Stars, 0'*])
           )).

?- density_distribution(
   msg(_, _, _, _, reportengine, _), 1, _D),
   print_distribution(_D).
  0   0.00
  1   0.25 *****
  2   0.11 **
  3   1.00 ********************
  4   1.00 ********************
  5   0.33 *******
  6   0.00
  7   0.00
  8   0.00
  9   0.10 **
 14   0.00
true.

?- density_distribution(
   msg(_, _, _, _, dbclient, _), 1, _D),
   print_distribution(_D).
  0   0.20 ****
  1   0.33 *******
  2   0.22 ****
  3   0.00
  4   0.00
  5   0.00
  6   0.00
  7   0.00
  8   0.00
  9   0.10 **
 14   0.40 ********
true.
```

The report engine's density is one in the fourth and fifth seconds: every message recorded then was a resize warning from worker 102's build, and nothing else was being written. The database client is present at the start, absent through the stall, and back for the drain and the failure. The windows from the eleventh to the fourteenth second are missing from both distributions because they are empty, and `statement_density/4` fails on them; the distribution is a list of the windows for which a density exists, and the gap in the list is the **Blackout** (page 535) of Section 25.17 seen from this side.

25.6 Relative Density

Relative Density compares two message sets within a region instead of one set with the whole: the number of messages of one kind per message of another, which is the right measure when the two are expected to occur in some proportion, errors per request, replies per send, warnings per build.

```
% relative_density(TemplateA, TemplateB, From, To,
% Ratio): Ratio is the number of messages matching
% TemplateA per message matching TemplateB in the
% window.
relative_density(TemplateA, TemplateB, From, To,
                 Ratio) :-
    window_messages(From, To, Set),
    TemplateA = msg(NA, _, _, _, _, _),
    TemplateB = msg(NB, _, _, _, _, _),
    aggregate_all(count,
                  ( member(NA, Set), TemplateA ),
                  CountA),
    aggregate_all(count,
                  ( member(NB, Set), TemplateB ),
                  CountB),
    CountB > 0,
    Ratio is CountA / CountB.

?- relative_density(msg(_, _, _, _, dbclient, _),
   msg(_, _, _, _, appserver, _), 0, 15, _R),
   format("~2f~n", [_R]).
0.26
true.

?- relative_density(
   msg(_, _, _, _, reportengine, _),
   msg(_, _, _, _, appserver, _), 2, 6, R).
R = 0.75.
```

Over the whole trace the database client writes one message per four of the application server's; between two and six seconds the report engine writes three per four, against one per four over the trace, which is the resize warnings again, measured now against the server's own output rather than against everything. The ratio is undefined when the second set is empty in the window, and the predicate fails then, as the density did. A ratio compares counts and knows nothing of what the counts should be; that two sets are expected in a proportion of one to one, as sends and replies are, is knowledge the analyst brings, and Chapter 27 encodes under **Opposition Messages** (page 572).

25.7 Trace Acceleration

Trace Acceleration is the change of statement current from one window to the next. A current that climbs over several windows is a system getting busier, or a problem announcing itself more often as it worsens; a sudden drop is a stall or a stop. Where the current says how much, the acceleration says whether it is getting more or less, which is the question when a trace is read for the onset of a problem.

```
% trace_acceleration(Width, From, Acceleration):
% the current of the window after the one starting
% at From minus the current of that window.
trace_acceleration(Width, From, Acceleration) :-
    window(Width, From, To),
    statement_current(From, To, C1),
    Next is To + Width,
    statement_current(To, Next, C2),
    Acceleration is C2 - C1.

% print_acceleration(Width): one signed value per
% window boundary.
print_acceleration(Width) :-
    forall(trace_acceleration(Width, From, A),
           ( (   A >= 0
             ->  Sign = '+'
             ;   Sign = ''
             ),
             format("~t~d~3| ~t~a~w~10|~n",
                    [From, Sign, A]) )).

?- print_acceleration(1).
  0     +2
  1     -3
  2     -7
  3     +0
  4     +1
  5     -1
  6     +1
  7     -1
  8    +19
  9    -21
 10     +0
 11     +0
 12     +0
 13     +5
 14     -5
true.

?- aggregate_all(max(A, From),
   trace_acceleration(1, From, A), M).
M = max(19, 8).
```

The acceleration is the difference of two integers here because the windows are one second wide, and the toplevel prints it as such. Two values stand out: the drop of seven after the third second, where the first requests had been served and the stall began, and the rise of nineteen into the tenth

second, where the queue drained. The last value compares the final window with an empty one past the end of the trace and is an artifact of the trace ending, not of the system; a predicate that stopped one window earlier would avoid it, at the price of saying nothing about the last window. The acceleration inherits the window dependence of the current and adds a sensitivity of its own, since a difference of two noisy values is noisier than either; on a trace with many messages per window the acceleration is readable, and on this one the single large values are all that is.

25.8 Delay Dynamics

Delay Dynamics is the change over the trace of the delays the system itself reports: response times, query times, wait times that appear as numbers in the message text. Delays that grow from request to request are a queue building or a resource running out; delays that shrink are a backlog clearing; and a single large delay in a series of small ones is an event.

The durations are in the text and must be extracted. For the completion messages, whose layout is fixed, a grammar with `library(dcg/basics)` reads the request number, the status, and the duration in one pass; `integer//1` consumes a run of digits and binds the integer, and the literal strings consume the fixed parts:

```
% completion(N, Request, Ms): message N reports
% the completion of Request in Ms milliseconds.
completion(N, Request, Ms) :-
    msg(N, _, _, _, _, Text),
    string_codes(Text, Codes),
    phrase(completion(Request, Ms), Codes).

completion(Request, Ms) -->
    "Request ", integer(Request),
    " completed: ", integer(_Status),
    ", ", integer(Ms), " ms".

% reported_duration(N, Ms): the text of message N
% ends with a duration of Ms milliseconds.
reported_duration(N, Ms) :-
    msg(N, _, _, _, _, Text),
    string_concat(Front, " ms", Text),
    split_string(Front, " ", "", Parts),
    last(Parts, Last),
    number_string(Ms, Last).

% delay_dynamics(Durations, Changes): Durations is
% the list of completion times in trace order and
% Changes their successive differences.
delay_dynamics(Durations, Changes) :-
    findall(Ms, completion(_, _, Ms), Durations),
    changes(Durations, Changes).

changes([], []).
changes([X | Xs], Changes) :-
    changes_from(Xs, X, Changes).
```

```
changes_from([], _, []).
changes_from([Y | Ys], X, [D | Ds]) :-
    D is Y - X,
    changes_from(Ys, Y, Ds).

?- completion(58, R, Ms).
R = 5,
Ms = 4300.

?- forall(completion(N, R, Ms),
   format("~t~d~3| request ~d ~t~d~22| ms~n",
          [N, R, Ms])).
 17 request 1      309 ms
 58 request 5     4300 ms
 60 request 6     3302 ms
 62 request 7     2305 ms
 64 request 8     1307 ms
 66 request 9      309 ms
true.

?- delay_dynamics(Ms, Changes).
Ms = [309, 4300, 3302, 2305, 1307, 309],
Changes = [3991, -998, -997, -998, -998].

?- reported_duration(49, Ms).
Ms = 7001.
```

The dynamics are in the second query's third column and in the list of changes. Request 1 took 309 milliseconds; request 5 took 4300, having waited in the queue from its arrival at 5.001 until worker 101 took it at 9.006; and each request after it took almost exactly a second less, because the health checks arrived a second apart and were all served in the same instant when the queue drained. The last, request 9, took the same 309 milliseconds as the first, which is what a request costs when nothing is in its way. The durations are the server's own account of its delays, and they agree with the time deltas of Section 25.1 to the millisecond, which is worth checking in any trace, since a reported duration measured by a different clock, or from a different starting point, need not.

`reported_duration/2` is the loose form for any message ending in a duration: `string_concat/3` with the suffix bound splits off the `" ms"` deterministically, as Chapter 8 showed, and the last word before it is converted with `number_string/2`. It finds the watchdog's report of a 7001 millisecond wait, which the completion grammar does not match; Section 25.11 uses it. Neither predicate knows what a duration measures. The 4300 of request 5 is queueing time plus service time, and the 309 of request 1 is service time alone, and the dynamics mix the two; separating them needs the request's lifecycle, which Chapter 26 reconstructs under **Use Case Trail** (page 561).

25.9 Periodic Error

A **Periodic Error** is an error message that recurs at a regular interval. Its regularity is the point: an error that fires every second is driven by a timer or a retry loop rather than by the events it reports, and it is often background that a **Trace Mask** (page 496) should hide, though a periodic error that begins at a certain moment marks the onset of the condition it keeps reporting.

An **error message** is one whose text contains an error word, and a set of at least three messages is periodic when the deltas between its consecutive members agree within the `period_tolerance` threshold and the period is at least the `min_period`; the message kind of Chapter 24 says which messages are the same error:

```
% error_word(Word): a word marking an error.
error_word("Error").
error_word("exception").
error_word("failed").

% error_message(N): message N contains an error
% word.
error_message(N) :-
    msg(N, _, _, _, _, Text),
    once(( error_word(Word),
           sub_string(Text, _, _, _, Word) )).

:- table message_kind/2.

% message_kind(N, Kind): the text of message N
% with every number replaced by # (Chapter 24).
message_kind(N, Kind) :-
    msg(N, _, _, _, _, Text),
    string_codes(Text, Codes),
    phrase(kind(KindCodes), Codes),
    string_codes(Kind, KindCodes).

kind(Ks) -->
    (   number_token
    ->  { Ks = [0'# | Ks1] },
        kind(Ks1)
    ;   [C]
    ->  { Ks = [C | Ks1] },
        kind(Ks1)
    ;   { Ks = [] }
    ).

number_token -->
    (   "0x"
    ->  xinteger(_)
    ;   integer(_)
    ).

% kind_messages(Kind, Set): the non-empty set of
% messages of kind Kind.
kind_messages(Kind, Set) :-
    aggregate_all(set(N), message_kind(N, Kind),
                  Set),
    Set \== [].
```

```
% periodic(Set, Period): the messages of Set, at
% least three, are Period ms apart within the
% tolerance, and Period is at least min_period.
periodic(Set, Period) :-
    Set = [_, _, _ | _],
    maplist(time_ms, Set, Times),
    changes(Times, [Period | Deltas]),
    threshold(min_period, MinPeriod),
    Period >= MinPeriod,
    threshold(period_tolerance, Tolerance),
    forall(member(D, Deltas),
           abs(D - Period) =< Tolerance).

% periodic_error(Kind, Set, Period): the error
% messages of kind Kind recur with Period.
periodic_error(Kind, Set, Period) :-
    distinct(Kind, ( error_message(N),
                     message_kind(N, Kind) )),
    kind_messages(Kind, Set),
    periodic(Set, Period).

?- findall(N, error_message(N), Ns).
Ns = [43, 45, 47, 69, 70].

?- periodic_error(K, S, P).
K = "Error: queue depth # exceeds threshold #",
S = [43, 45, 47],
P = 999 ;
false.
```

Of the five error messages, three are one kind recurring at 999 and 1000 milliseconds, and the recognizer reports the first delta as the period. The queue-depth error is periodic because the requests that overflow the queue arrive periodically, so the period belongs to the client, not to the error, and the error is not background: it began at 7.002 when the queue depth first exceeded two and would have gone on as long as requests kept arriving and no worker was free. `once/1` in `error_message/1` stops the search at the first error word found, so that a message containing two of them, as message 69 does, is one error message and not two; `distinct/2` enumerates each error kind once.

The error words are a convention of the system, and the three here are the ones `trace1.pl` uses; a system that writes `E:` or `[ERR]` needs its own facts. The tolerance is a judgment about jitter, ten milliseconds here, and a period of 999 against 1000 is within it; a system with coarse timestamps needs a wider one. Three occurrences are the minimum for regularity to mean anything, and the predicate says nothing about an error that recurred twice.

25.10 Periodic Message Block

A **Periodic Message Block** is a group of consecutive messages that recurs as a group at a regular interval: a polling loop’s report, a watchdog’s round, a timer’s work. Recognizing the block as one

thing lets the analyst mask it as one thing, and a change in the block's content between occurrences, or a missed occurrence, is then visible against its regularity.

A block of a given length is the sequence of kinds of that many consecutive messages, and it recurs where the same sequence starts again:

```
% block_kinds(N, Length, Kinds): Kinds are the
% kinds of the Length messages from N on.
block_kinds(N, Length, Kinds) :-
    msg(N, _, _, _, _, _),
    Last is N + Length - 1,
    numlist(N, Last, Ns),
    maplist(message_kind, Ns, Kinds).

% periodic_message_block(Length, Kinds, Starts,
% Period): the block Kinds of Length messages
% starts at each message in Starts, periodically.
periodic_message_block(Length, Kinds, Starts,
                       Period) :-
    distinct(Kinds,
             block_kinds(_, Length, Kinds)),
    aggregate_all(set(N),
                  block_kinds(N, Length, Kinds),
                  Starts),
    periodic(Starts, Period).

% print_kinds(Kinds): one kind per line, indented.
print_kinds(Kinds) :-
    forall(member(Kind, Kinds),
           ( truncated(Kind, 46, Short),
             format("  ~s~n", [Short]) )).

?- periodic_message_block(1, _K, S, P),
   print_kinds(_K).
  Warning: table resize, # -> #
S = [31, 32, 33, 34, 35, 36],
P = 500 ;
  No idle worker, request # queued
S = [38, 40, 42],
P = 999 ;
  Request # received: GET /health
S = [39, 41, 44, 46],
P = 1000 ;
  Error: queue depth # exceeds threshold #
S = [43, 45, 47],
P = 999 ;
false.

?- periodic_message_block(2, _K, S, P),
   print_kinds(_K).
  Warning: table resize, # -> #
  Warning: table resize, # -> #
S = [31, 32, 33, 34, 35],
P = 500 ;
false.
```

Blocks of one message are periodic messages, and four kinds recur regularly: the resize warning every half second, six times, and three kinds a second apart that belong to the requests arriving while no worker was free. Blocks of two find only the resize warning again, paired with itself, because no two-message group recurs three times in this trace; the health check and its queued message recur twice as a pair and then the queued message is missing, for requests 8 and 9, which Chapter 27 treats under **Missing Message** (page 575). The `min_period` threshold is what excludes the one group that does recur as a pair, the dispatch and completion of requests 6 to 9 two milliseconds apart, and it is excluded on purpose: regularity at that scale is a loop iterating, not a timer firing, and the predicate's notion of period is the timer's.

`numlist/3` of Chapter 5 generates the message numbers of a block and `maplist/3` looks up their kinds, failing when the block runs past the end of the trace. The recognizer finds blocks whose every occurrence is identical in kind; a block in which one message varies between occurrences, as a polling report with an optional line would, is not found, and a formulation that matched blocks up to a wildcard would be the next step.

25.11 Timeout

A **Timeout** shows in a trace as a duration or a time delta that matches a configured limit: a wait reported as 30000 milliseconds, a retry arriving a second after the request it repeats, an error written exactly five seconds after the operation it reports on. The match identifies the mechanism, a timer expiring, where the message text may not name it.

The limits are facts, since they come from configuration and not from the trace, and a value matches one within the `timeout_tolerance`:

```
% timeout_value(Name, Ms): a configured timeout.
timeout_value(health_check, 1000).
timeout_value(worker_wait, 7000).

% near_timeout(Ms, Name): Ms is within tolerance
% of the timeout Name.
near_timeout(Ms, Name) :-
    timeout_value(Name, Value),
    threshold(timeout_tolerance, Tolerance),
    abs(Ms - Value) =< Tolerance.

% timeout(N, Name): message N reports a duration
% matching the timeout Name.
timeout(N, Name) :-
    reported_duration(N, Ms),
    near_timeout(Ms, Name).
```

```
% timeout(N, Next, Name): Next is the next message
% of the kind of N, and the delta between them
% matches the timeout Name.
timeout(N, Next, Name) :-
    msg(N, _, _, _, _, _),
    message_kind(N, Kind),
    kind_messages(Kind, Set),
    time_delta(Set, N, Next, Delta),
    near_timeout(Delta, Name).

?- timeout(N, Name).
N = 49,
Name = worker_wait ;
false.

?- timeout(N, Next, health_check).
N = 38,
Next = 40 ;
N = 39,
Next = 41 ;
N = 40,
Next = 42 ;
N = 41,
Next = 44 ;
N = 43,
Next = 45 ;
N = 44,
Next = 46 ;
N = 45,
Next = 47 ;
false.
```

The watchdog's report that worker 103 had waited 7001 milliseconds matches a seven-second limit, and the health checks and the messages they cause recur at the one-second limit. Both readings are hypotheses the facts make precise rather than findings the trace makes certain: if the client that sends health checks gives up after a second and retries, the trace is consistent with that, and if it polls every second regardless, the trace is consistent with that too. A value that matches a timeout may be a timer that is not a timeout at all, as the resize warning's half second is, and the predicate cannot tell a limit from a schedule. Its use is the other direction: an analyst who knows the configured limits supplies them as facts, and the recognizer finds the places in the trace where they show.

25.12 Significant Interval

A **Significant Interval** is the stretch of the trace between two messages that matter, read as a unit: what happened between the request and its failure, between the last normal message and the first error, between a start and the end that never came. The two boundary messages are chosen by the analyst, and the interval's content and duration are the finding.

```
% significant_interval(N1, N2, Duration, Inside):
% Duration ms passed between messages N1 and N2,
% and Inside is the set of messages between them.
significant_interval(N1, N2, Duration, Inside) :-
```

```
    N1 < N2,
    time_ms(N1, T1),
    time_ms(N2, T2),
    Duration is T2 - T1,
    Low is N1 + 1,
    High is N2 - 1,
    (   Low =< High
    ->  numlist(Low, High, Inside)
    ;   Inside = []
    ).

?- significant_interval(22, 31, D, _I),
   print_messages(_I).
23  2.000 100 appserver    Request 3 received: POS
24  2.001 103 appserver    Request 3 dispatched to
25  2.002 103 appserver    Session 9 commit start
26  2.003 103 dbclient     Send start: session=71,
27  2.100 100 appserver    Request 4 received: GET
28  2.101 104 appserver    Request 4 dispatched to
29  2.102 104 dbclient     Reply received: session
30  2.103 104 appserver    Session 9 lookup start
D = 949.

?- significant_interval(66, 67, D, I).
D = 4691,
I = [].
```

Between the start of the build of report 18 on worker 102 and the first warning that its table was being resized lie 949 milliseconds and eight messages, and the eight are the arrival of requests 3 and 4 and the two workers taking the operations on session 9 that the dump of Part III found deadlocked. Whether the build's trouble and the deadlock are connected the interval does not say; it says they are contemporaneous, which is the question an analyst then takes to the dump. The second interval is the gap before thread 109's request, with nothing inside, and Section 25.17 names it.

The predicate is a cut of the trace by message number, and its significance is entirely in the choice of `N1` and `N2`; given two message numbers it reports the interval between any two messages whatever. The choice is the analyst's, informed by the **Significant Event** (page 566) and **Anchor Messages** (page 577) patterns of Chapter 27, which are the usual sources of the boundaries.

25.13 Punctuated Activity

Punctuated Activity is activity that comes in bursts separated by gaps: a thread that does several things in quick succession, falls silent, and does several more. The bursts are the units of the thread's work, and the gaps between them are where the thread waited; reading the bursts rather than the messages gives the thread's story at the right grain.

A **burst** is a maximal run of a set's messages in which no two consecutive messages are more than the `burst_gap` apart; the recursion takes one burst at a time:

```
% bursts(Set, Bursts): Bursts divides the message
% set Set into maximal runs with no gap above the
% burst_gap threshold.
bursts([], []).
bursts([N | Ns], [Burst | Bursts]) :-
    burst(Ns, N, Burst, Rest),
    bursts(Rest, Bursts).

burst(Ns, N, [N | Burst], Rest) :-
    threshold(burst_gap, Gap),
    (   Ns = [Next | Ns1],
        time_ms(N, T1),
        time_ms(Next, T2),
        T2 - T1 =< Gap
    ->  burst(Ns1, Next, Burst, Rest)
    ;   Burst = [],
        Rest = Ns
    ).

% punctuated_activity(TID, Bursts): the messages
% of thread TID form at least two bursts.
punctuated_activity(TID, Bursts) :-
    thread_messages(TID, Set),
    bursts(Set, Bursts),
    Bursts = [_, _ | _].

% print_bursts(Bursts): one line per burst with
% its message range, size, and span in ms.
print_bursts(Bursts) :-
    forall(member(Burst, Bursts),
           print_burst(Burst)).

print_burst(Burst) :-
    Burst = [First | _],
    last(Burst, Last),
    length(Burst, Count),
    time_ms(First, T1),
    time_ms(Last, T2),
    Span is T2 - T1,
    format("~t~d~3|-~d~t~7|~t~d~10|~t~d~16| ms~n",
           [First, Last, Count, Span]).

?- punctuated_activity(101, _B), print_bursts(_B).
  7-7    1     0 ms
 12-17   6   308 ms
 53-66  14   303 ms
true.

?- punctuated_activity(100, _B), print_bursts(_B).
  1-6    6   214 ms
 11-27   4  1098 ms
 37-38   2     1 ms
 39-40   2     1 ms
 41-43   3     2 ms
 44-45   2     1 ms
 46-47   2     1 ms
 71-71   1     0 ms
true.
```

```
?- punctuated_activity(T, _B), length(_B, L).
T = 100,
L = 8 ;
T = 101,
L = 3 ;
T = 102,
L = 3 ;
T = 103,
L = 2 ;
T = 104,
L = 2 ;
false.
```

Worker 101's activity is three bursts: its readiness message alone, request 1 in 308 milliseconds, and the whole drain of the queue in 303, five requests in the time one took before, because four of them had nothing to do. The main thread's eight bursts are startup, the arrival of the first four requests, five one-second visits to queue a request and report the depth, and the shutdown. Threads 108 and 109 are not punctuated: each wrote one burst, and the predicate does not report an activity with a single burst as punctuated, since the pattern is in the alternation.

The `burst_gap` of half a second is the same kind of judgment as the discontinuity threshold of Section 25.2 and interacts with it: at this setting the main thread's one-second rhythm separates its bursts, and at a gap of two seconds its five visits merge into one burst of eleven messages, which is the health-check phase read as one activity. Both readings are correct at their grain, and the predicate reports the one its threshold selects.

25.14 Milestones

Milestones are the messages that mark the stages of the system's life in the trace: it started, it finished initializing, it began serving, it failed, it stopped. They partition the trace into phases in time, as Chapter 24 partitioned it by attribute, and the intervals between them are the durations of the phases, the first figures an analyst compares against a healthy run.

Which messages are milestones is knowledge of the system, held as facts giving the prefixes of their texts:

```
% milestone_prefix(Prefix): the text a milestone
% message begins with.
milestone_prefix("Service starting").
milestone_prefix("Worker pool started").
milestone_prefix("Unhandled exception").
milestone_prefix("Service stopping").

% milestone(N): message N is a milestone.
milestone(N) :-
    msg(N, _, _, _, _, Text),
    milestone_prefix(Prefix),
    sub_string(Text, 0, _, _, Prefix).
```

```
% milestones(Ns): the message set of milestones.
milestones(Ns) :-
    findall(N, milestone(N), Ns0),
    sort(Ns0, Ns).

?- milestones(_Ns), print_messages(_Ns).
 1  0.000 100 appserver    Service starting, versi
 6  0.214 100 appserver    Worker pool started, th
70 14.003 109 appserver    Unhandled exception 0xc
71 14.010 100 appserver    Service stopping
true.

?- milestones(_Ns), print_deltas(_Ns).
 1-6     214 Worker pool started, threads
 6-70  13789 Unhandled exception 0xc00000
70-71      7 Service stopping
true.
```

Four milestones, three phases: 214 milliseconds of startup, 13789 milliseconds of service, and seven milliseconds from the exception to the stop. The last figure is the one an analyst notes: the service stopped seven milliseconds after an unhandled exception on a worker thread, which is the behavior of a process being torn down by its exception, and it agrees with the dump of Part III, taken of a process whose thread 109 had just faulted. The milestone set is a message set, so `print_deltas/1` of Section 25.1 prints the phase durations without a new predicate, and the message sets between consecutive milestones are the phases as trace partitions in the sense of Chapter 24.

The facts carry the knowledge, and a trace from another system needs its own; a milestone that is absent, a stopping message that never came, is as informative as one that is present, and the predicate reports absence only by the milestone set lacking it, which the analyst must notice. Chapter 27 treats the related **Anchor Messages** (page 577), which are chosen for the analysis rather than fixed by the system.

25.15 Last Activity

The **Last Activity** of a thread is its final message and how long before the end of the trace it was written. Threads whose last message is close to the end were alive at the end; threads whose last message is far before it stopped, and what they were doing when they stopped is the last thing the trace knows about them.

```
% last_activity(TID, N, Age): N is the last
% message of thread TID, written Age ms before the
% trace ended.
last_activity(TID, N, Age) :-
    thread_messages(TID, Set),
    last(Set, N),
    time_ms(N, T),
    trace_end(End),
    Age is End - T.
```

```
% print_last_activity: one line per thread, the
% longest silent first.
print_last_activity :-
    forall(order_by([desc(Age)],
                    last_activity(TID, N, Age)),
           print_last(TID, N, Age)).

print_last(TID, N, Age) :-
    msg(N, _, _, _, _, Text),
    truncated(Text, 26, Short),
    format("~d~t~5|~t~d~8| ~t~d~15| ~s~n",
           [TID, N, Age, Short]).

?- print_last_activity.
103   26  12007 Send start: session=71, by
104   30  11907 Session 9 lookup start
102   36   9010 Warning: table resize, 262
108   52   5005 Housekeeping: worker 101 i
101   66   4701 Request 9 completed: 200,
109   70      7 Unhandled exception 0xc000
100   71      0 Service stopping
true.
```

The table is the trace's account of what the dump of Part III found. Workers 103 and 104 last spoke twelve seconds before the end, each at the start of an operation on session 9, and the dump shows them deadlocked on two critical sections; worker 102 last spoke nine seconds before the end, mid-build, and the dump shows it consuming the processor; worker 101 completed its last request and went quiet; thread 109 faulted seven milliseconds before the end. `order_by/2` of Chapter 9 sorts the threads by the age of their last message, so that the ones that stopped first come first.

The age is measured from the end of the trace, and the end of the trace is not the end of the process; a thread silent for twelve seconds before a trace stopped by a size limit may have resumed a second later. The reading of the table assumes that the trace ran to the end, which Chapter 24's **Truncated Trace** (page 498) test established for this one, and it must be established before the ages mean anything.

25.16 No Activity

No Activity is the absence of messages from a thread or component in an interval during which activity was expected, including the interval from its last message to the end of the trace and from the start of the trace to its first. Where a **Discontinuity** (page 513) is a gap between two messages that exist, No Activity is an interval of silence of a set, bounded by the trace as much as by messages, and the question it raises is whether the silent party was idle, blocked, dead, or outside the focus of tracing.

The **silences** of a set are the leading interval, the gaps between consecutive messages, and the trailing interval, each a pair of times; those longer than the `no_activity` threshold are instances:

```
% silence(Set, From, To): no message of Set was
% written in the interval From..To, which is
% bounded by messages of Set or the trace ends.
silence(Set, Start, T) :-
    trace_start(Start),
    Set = [First | _],
    time_ms(First, T).
silence(Set, T1, T2) :-
    nextto(N1, N2, Set),
    time_ms(N1, T1),
    time_ms(N2, T2).
silence(Set, T, End) :-
    last(Set, Last),
    time_ms(Last, T),
    trace_end(End).

% no_activity(Set, From, To): a silence of Set
% longer than the no_activity threshold.
no_activity(Set, From, To) :-
    silence(Set, From, To),
    threshold(no_activity, Min),
    To - From > Min.


?- thread_messages(103, _S),
   no_activity(_S, F, T).
F = 2003,
T = 14010.

?- thread_messages(108, _S),
   no_activity(_S, F, T).
F = 0,
T = 9002 ;
F = 9005,
T = 14010.

?- thread_messages(109, _S),
   no_activity(_S, F, T).
F = 0,
T = 14000 ;
false.

?- thread_messages(101, _S),
   no_activity(_S, F, T).
F = 1311,
T = 9006 ;
F = 9309,
T = 14010.
```

Worker 103 was silent for the last twelve seconds, which Section 25.15 said from the other end. The housekeeping thread 108 was silent before its one round and after it, and thread 109 was silent for fourteen seconds before its first message, which the dump explains: it was a worker thread throughout, and it wrote nothing because it did nothing that the trace statements record, until a request reached it. The three clauses of `silence/3` produce the leading, inner, and trailing intervals in that order, and the predicate is nondeterministic by design, since a set may have several.

The pattern's condition, that activity was expected, is not in the predicate. Thread 109's leading silence and worker 103's trailing one are the same shape, and one is a thread with nothing to do while the other is a thread that cannot do anything; the dump tells them apart, and the trace does not. The predicate reports the silences, and the expectation is the analyst's.

25.17 Blackout

A **Blackout** is an interval in which nothing at all was recorded: no thread, no component, no message. It is the **Discontinuity** (page 513) of the whole trace, and it is read differently from a thread's gap, because the whole system falling silent has fewer causes than one thread doing so: the process was suspended or idle, the machine was paused, or tracing was interrupted and the trace is missing a piece. Which of these it was is the first question a blackout raises.

The recognizer is the trace-wide time delta against the `blackout` threshold, in the enumerator form and in the explainer form of Chapter 17, whose evidence term lists the facts the instance rests on:

```
% blackout(From-To): no message was written
% between From and To, an interval longer than the
% blackout threshold.
blackout(From-To) :-
    time_delta(N, Next, Delta),
    threshold(blackout, Min),
    Delta > Min,
    time_ms(N, From),
    time_ms(Next, To).

% blackout(Interval, Evidence): the explainer
% form, with the facts the instance rests on.
blackout(From-To, evidence('Blackout', From-To,
                           Facts)) :-
    time_delta(N, Next, Delta),
    threshold(blackout, Min),
    Delta > Min,
    time_ms(N, From),
    time_ms(Next, To),
    Facts = [time_delta(N, Next, Delta),
             threshold(blackout, Min)].

% print_evidence(Evidence): the instance and each
% supporting fact as a clause.
print_evidence(evidence(Pattern, Instance, Fs)) :-
    format("~w: ~w~n", [Pattern, Instance]),
    forall(member(Fact, Fs),
           portray_clause(Fact)).

?- blackout(I).
I = 9309-14000 ;
false.
```

```
?- blackout(_, _E), print_evidence(_E).
Blackout: 9309-14000
time_delta(66, 67, 4691).
threshold(blackout, 2000).
true ;
false.

?- blackout(_, evidence(_, _, _Facts)),
   maplist(call, _Facts).
true ;
false.
```

The trace has one blackout, from the completion of request 9 at 9.309 seconds to the arrival of request 10 at 14.000, and the evidence term says why: a time delta of 4691 milliseconds against a threshold of 2000. `portray_clause/1` of Chapter 8 prints each fact as a clause, and the last query checks what an evidence term promises, that each of its facts is a goal that holds; `maplist(call, Facts)` runs them all, and this is the test Chapter 29 generalizes when it derives explanations automatically.

The blackout's cause is not in the trace. The dump of Part III is the other artifact, and it shows a process with two workers deadlocked, one spinning, and one idle: the process was running, not suspended, and the trace resumes with the same numbering and the same clock, which argues against an interruption of tracing. What remains is a system with nothing to say: no request arrived in the interval, the stalled workers wrote nothing, and the spinning worker had stopped writing at 5.000. A blackout in a trace from a system that should have been busy is evidence that it was not, and that is as far as the trace alone takes it.

Chapter 26. Activities and Threads of Activity

A software trace records what several threads did, interleaved in the order they wrote. The patterns of this chapter take the interleaving apart: they follow one thread's messages as a sequence, follow a request or a component through the messages of many threads, find where activities overlap, hand off, and resume, and tell the components in the foreground of an activity from those in its background. They are the patterns of the viewpoint by thread and by activity that Chapter 24 introduced under **Trace Viewpoints** (page 506), and they are where the trace begins to be read as a narrative rather than measured as a set.

The helpers are those of the previous two chapters, and the thresholds are collected first, in milliseconds where they are times:

```
:- [trace1].
:- use_module(library(dcg/basics)).

truncated(Text, Width, Short) :-
    string_length(Text, Len),
    (   Len =< Width
    ->  Short = Text
    ;   sub_string(Text, 0, Width, _, Short)
    ).

print_row(N) :-
    msg(N, Time, _, TID, Component, Text),
    truncated(Text, 23, Short),
    format("~t~d~2| ~t~3f~9| ~d ~a~t~26| ~s~n",
           [N, Time, TID, Component, Short]).

print_messages(Ns) :-
    forall(member(N, Ns), print_row(N)).

time_ms(N, Ms) :-
    msg(N, Time, _, _, _, _),
    Ms is round(Time * 1000).

% threshold(Name, Value): the thresholds of this
% chapter.
threshold(coupling, 10).
threshold(resume, 1000).
threshold(background, 0.75).
threshold(host_size, 3).

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

26.1 Thread of Activity

A **Thread of Activity** is the sequence of messages written by one thread, read in order as that thread's account of what it did. It is the basic unit of narrative in a trace: a thread does one thing at

a time, so its messages form a story with a beginning and an end, where the trace as a whole is several stories cut into pieces and shuffled together.

The predicate is Chapter 25's `thread_messages/2` under the pattern's name, with `bagof/3` providing the two modes:

```
% thread_of_activity(TID, Set): Set is the message
% set of thread TID; with TID unbound, one answer
% per thread.
thread_of_activity(TID, Set) :-
    bagof(N, T^P^C^X^msg(N, T, P, TID, C, X), Ns),
    sort(Ns, Set).

?- thread_of_activity(103, _S),
   print_messages(_S).
 9  0.216 103 appserver    Worker 103 ready
24  2.001 103 appserver    Request 3 dispatched to
25  2.002 103 appserver    Session 9 commit start
26  2.003 103 dbclient     Send start: session=71,
true.

?- thread_of_activity(T, _S), length(_S, L).
T = 100,
L = 22 ;
T = 101,
L = 21 ;
T = 102,
L = 11 ;
T = 103,
L = 4 ;
T = 104,
L = 4 ;
T = 108,
L = 5 ;
T = 109,
L = 4.
```

Worker 103's thread of activity is four messages and reads as a story: ready, given request 3, began committing session 9, began sending on the database connection, and then nothing. The second query enumerates the seven threads, with no choice point after the last, because `bagof/3` produces one answer per binding of its free variable and knows when they are exhausted. The sequence is the thread's own order of writing, which is also time order within a thread; what the predicate does not supply is the meaning of the sequence, which is in the text, and the relations between threads, which the rest of this chapter supplies.

26.2 Adjoint Thread of Activity

An **Adjoint Thread of Activity** is a sequence of messages selected by some attribute other than the thread and read as if a thread had written it: all messages of one component, or all messages that mention one request, in trace order. The construction lets the analyst follow a thing through the

trace, a request as it passes from the dispatcher to a worker, where no single thread's story contains it. The short form Adjoint Thread is used below.

The attribute is a term, `component(C)` or `request(R)`, and the request number must be read from the text. A grammar scans for the word `request`, lowercased with `string_lower/2` so that both `Request 5` and `request 5` match, and reads the number that follows; the if-then-else in the grammar body makes the scan deterministic, and the number is read into a fresh variable and unified afterwards, so that a bound `Request` of 1 does not match the prefix of `10`:

```
% request_id(N, Request): the text of message N
% mentions request number Request.
request_id(N, Request) :-
    msg(N, _, _, _, _, Text),
    string_lower(Text, Lower),
    string_codes(Lower, Codes),
    phrase(request_mention(Request), Codes).

request_mention(Request) -->
    (   "request "
    ->  integer(Found),
        remainder(_),
        { Request = Found }
    ;   [_],
        request_mention(Request)
    ).

% adjoint(Key, N): message N belongs to the
% adjoint thread selected by Key.
adjoint(component(C), N) :-
    msg(N, _, _, _, C, _).
adjoint(request(R), N) :-
    request_id(N, R).

% adjoint_thread_of_activity(Key, Set): Set is the
% message set of the adjoint thread Key.
adjoint_thread_of_activity(Key, Set) :-
    bagof(N, adjoint(Key, N), Ns),
    sort(Ns, Set).

?- request_id(38, R).
R = 5.

?- request_id(67, 1).
false.

?- adjoint_thread_of_activity(request(5), _S),
   print_messages(_S).
37  5.001 100 appserver    Request 5 received: GET
38  5.002 100 appserver    No idle worker, request
53  9.006 101 appserver    Request 5 dispatched to
58  9.301 101 appserver    Request 5 completed: 20
true.
```

```
?- adjoint_thread_of_activity(component(C), _S),
   length(_S, L).
C = appserver,
L = 47 ;
C = cachelib,
L = 1 ;
C = dbclient,
L = 12 ;
C = reportengine,
L = 11.
```

The adjoint thread of request 5 crosses two threads: received and queued by the main thread, dispatched and completed by worker 101 four seconds later. `remainder//1` of `library(dcg/basics)` consumes the rest of the text after the number, so that `phrase/2` succeeds on the whole string. The adjoint by component is the partition of Chapter 24 one part at a time, which shows the pattern's relation to it: a **Trace Partition** (page 489) along a dimension is the family of all adjoint threads of that dimension.

The adjoint thread knows only what the text says. The query and build messages that worker 101 wrote while serving request 5 do not mention the request and are not in its adjoint thread; they belong to it by being on the same thread between its dispatch and its completion, which is a different kind of membership, and Section 26.20 joins the two under **Use Case Trail** (page 561).

26.3 Activity Region

An **Activity Region** is the stretch of the trace over which an activity extends: from the first message of a thread, adjoint thread, or other message set to its last, in time and in message numbers. The region locates the activity in the trace so that it can be compared with other regions, and its length is the activity's duration as the trace saw it.

```
% activity_region(Set, From, To): the messages of
% Set were written between From and To seconds.
activity_region(Set, From, To) :-
    Set = [First | _],
    last(Set, Last),
    msg(First, From, _, _, _, _),
    msg(Last, To, _, _, _, _).

?- thread_of_activity(103, _S),
   activity_region(_S, From, To).
From = 0.216,
To = 2.003.

?- adjoint_thread_of_activity(request(5), _S),
   activity_region(_S, From, To).
From = 5.001,
To = 9.301.
```

Worker 103 was active from its readiness to its last message at 2.003 seconds, and request 5 extends from 5.001 to 9.301, which is the 4300 milliseconds its completion message reports. The region is

the span between head and tail, in the sense of Chapter 24, and includes the gaps inside it; an activity that was silent for most of its region, as worker 101 was between requests 1 and 5, has a region that overstates how much it did, and the bursts of Chapter 25's **Punctuated Activity** (page 529) are the finer description.

26.4 Activity Theatre

The **Activity Theatre** of an interval is the set of threads performing in it, the actors on the stage during that stretch of the trace, with how much each said. Where the region locates one activity, the theatre asks who else was active at the same time, and a theatre with one actor is a different situation from one with four.

```
% window_messages(From, To, Set): the messages
% with From =< Time < To, in seconds.
window_messages(From, To, Set) :-
    findall(N,
            ( msg(N, T, _, _, _, _),
              T >= From, T < To ),
            Ns),
    sort(Ns, Set).

% activity_theatre(From, To, Actors): Actors is
% the list of TID-Count pairs of the threads with
% messages in the interval.
activity_theatre(From, To, Actors) :-
    window_messages(From, To, Set),
    findall(TID-N,
            ( member(N, Set),
              msg(N, _, _, TID, _, _) ),
            Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Groups),
    maplist(count_group, Groups, Actors).

count_group(Key-Ns, Key-Count) :-
    length(Ns, Count).

?- activity_theatre(2.0, 2.2, Actors).
Actors = [100-2, 103-3, 104-3].

?- activity_theatre(9.0, 9.4, Actors).
Actors = [100-2, 101-14, 108-5].

?- activity_theatre(3.0, 5.0, Actors).
Actors = [102-4].
```

Three intervals, three theatres. Just after two seconds the main thread and workers 103 and 104 are on stage, setting up the deadlock; just after nine seconds the main thread, the housekeeping thread, and worker 101 draining the queue; and from three to five seconds worker 102 alone, resizing its table while every other thread is silent. The predicate counts messages per thread and says nothing about what the actors did or whether they interacted, which the **Coupled Activities** (page 545) of

Section 26.9 ask; an interval is given in seconds, and the theatre of a region found by Section 26.3 is the theatre of its two bounds.

26.5 Activity Overlap

Two activities exhibit **Activity Overlap** when their regions share an interval: both were in progress at once. For activities that should be sequential, overlap is the finding; for activities that may run concurrently, the overlap is where they could have interfered, and a problem in one is read against what the other was doing then.

```
% activity_overlap(SetA, SetB, From, To): the
% regions of the message sets SetA and SetB share
% the interval From..To.
activity_overlap(SetA, SetB, From, To) :-
    activity_region(SetA, FromA, ToA),
    activity_region(SetB, FromB, ToB),
    From is max(FromA, FromB),
    To is min(ToA, ToB),
    From < To.


?- adjoint_thread_of_activity(request(5), _A),
   adjoint_thread_of_activity(request(6), _B),
   activity_overlap(_A, _B, From, To).
From = 6.0,
To = 9.301.

?- adjoint_thread_of_activity(request(1), _A),
   adjoint_thread_of_activity(request(2), _B),
   activity_overlap(_A, _B, From, To).
false.

?- adjoint_thread_of_activity(request(RA), _A),
   adjoint_thread_of_activity(request(RB), _B),
   RA < RB, activity_overlap(_A, _B, _, _),
   RA =< 6.
RA = 5,
RB = 6 ;
RA = 5,
RB = 7 ;
RA = 5,
RB = 8 ;
RA = 5,
RB = 9 ;
RA = 6,
RB = 7 ;
RA = 6,
RB = 8 ;
RA = 6,
RB = 9 ;
false.
```

Requests 1 and 2 did not overlap: the first completed before the second arrived. Requests 5 to 9 all overlap one another, because each arrived while the earlier ones were still queued, and the overlap

of requests 5 and 6 runs from the arrival of 6 to the completion of 5. The predicate compares regions as intervals and `max/2` and `min/2` of Chapter 6 compute their intersection; the strict inequality excludes regions that merely touch. Overlap by region is coarse: two threads whose regions overlap for ten seconds may never have been active in the same millisecond, and the **Message Interleave** (page 556) of Section 26.17 is the finer test.

26.6 Fiber of Activity

A **Fiber of Activity** is, in the operational sense this book gives the term, the part of one thread's activity that belongs to one adjoint thread: the messages worker 101 wrote on behalf of the database client, or on behalf of request 5. The thread is cut lengthwise by the attribute, and each strand is a fiber; the catalog's term is borrowed from topology, where a fiber is what lies over one point of a base space, and here the base is the attribute and the point is its value.

```
% fiber_of_activity(TID, Key, Fiber): Fiber is the
% non-empty intersection of thread TID's activity
% with the adjoint thread Key.
fiber_of_activity(TID, Key, Fiber) :-
    thread_of_activity(TID, Thread),
    adjoint_thread_of_activity(Key, Adjoint),
    ord_intersection(Thread, Adjoint, Fiber),
    Fiber \== [].

?- fiber_of_activity(101, component(dbclient), F).
F = [13, 14, 54, 55].

?- fiber_of_activity(101, request(1), F).
F = [12, 17].

?- fiber_of_activity(T, request(5), F).
T = 100,
F = [37, 38] ;
T = 101,
F = [53, 58] ;
false.
```

Worker 101's database fiber is two queries, one per report it built, and its fiber for request 1 is the dispatch and the completion, the two messages of that thread that name the request. Request 5 has fibers in two threads, and the two fibers together are its adjoint thread, which is the relation the next section states for all threads at once. The fiber is an `ord_intersection/3` of sets, so it inherits the limits of both: the thread's set is exact, and the adjoint's is as good as the text.

26.7 Fiber Bundle

A **Fiber Bundle** is, operationally, the family of all fibers over one adjoint thread, indexed by thread: for a component, which threads used it and with what messages; for a request, which threads took part in it. The bundle is the adjoint thread seen as a collection of per-thread strands, and the base over which the strands lie is the attribute that selected them.

```
% fiber_bundle(Key, Bundle): Bundle is the list of
% TID-Fiber pairs of the adjoint thread Key.
fiber_bundle(Key, Bundle) :-
    findall(TID-Fiber,
            fiber_of_activity(TID, Key, Fiber),
            Bundle).

% print_bundle(Bundle): one line per fiber.
print_bundle(Bundle) :-
    forall(member(TID-Fiber, Bundle),
           format("~d~t~6|~w~n", [TID, Fiber])).

?- fiber_bundle(component(dbclient), _B),
   print_bundle(_B).
100   [4,5]
101   [13,14,54,55]
102   [20,21]
103   [26]
104   [29]
109   [68,69]
true.

?- fiber_bundle(request(5), _B), print_bundle(_B).
100   [37,38]
101   [53,58]
true.
```

Six threads used the database client, and the bundle shows how: the main thread to connect, workers 101 and 102 for paired query starts and ends, workers 103 and 104 for one message each, a send and a reply, and thread 109 for a receive that failed. The unpaired fibers are where the reading stops: 103 sent and never reported a reply, 104 received a reply it had not been seen to request, and 109 began a receive and failed. Those are three threads of the dump of Part III, and the bundle over one component is where the trace points at them. The bundle is a list of pairs and no more; the catalog’s use of the term carries a picture of fibers lying side by side over a base that the list does not draw, and `print_bundle/1` is the nearest the formalization comes to drawing it.

26.8 Sheaf of Activities

A **Sheaf of Activities** is, operationally, the assignment to every interval of the trace of the activities present in it, the threads with their messages restricted to that interval, such that the assignment for a smaller interval is the restriction of the assignment for a larger one. The term is again topological; what the book takes from it is the picture of local views of the activity that agree where they overlap and can be read at any scale, and the predicate computes the local view for one interval.

```
% sheaf_of_activities(From, To, Sections):
% Sections is the list of TID-Set pairs of the
% threads with messages in From..To.
sheaf_of_activities(From, To, Sections) :-
    window_messages(From, To, Set),
    findall(TID-N,
            ( member(N, Set),
              msg(N, _, _, TID, _, _) ),
            Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Sections).

% restriction(Sections, From, To, Restricted): the
% sections cut down to the interval From..To.
restriction(Sections, From, To, Restricted) :-
    window_messages(From, To, Window),
    findall(TID-Part,
            ( member(TID-Ns, Sections),
              ord_intersection(Ns, Window, Part),
              Part \== [] ),
            Restricted).

?- sheaf_of_activities(2.0, 2.1, S).
S = [100-[23], 103-[24, 25, 26]].

?- sheaf_of_activities(1.0, 2.2, _Large),
   restriction(_Large, 2.0, 2.1, R).
R = [100-[23], 103-[24, 25, 26]].

?- sheaf_of_activities(1.0, 2.2, _Large),
   sheaf_of_activities(2.0, 2.1, S),
   restriction(_Large, 2.0, 2.1, S).
S = [100-[23], 103-[24, 25, 26]].
```

The sections over the tenth of a second after 2.0 are the main thread's one message and worker 103's three; the sections over the larger interval, restricted to the smaller, are the same, which is the agreement the picture requires and the third query checks. The sections of a sheaf over an interval are the actors of Section 26.4 with their messages rather than their counts, and the sheaf over the whole trace is the partition by thread of Chapter 24. The formalization is deliberately modest. It gives local views and their restriction and does not attempt the gluing of local views into global ones that the mathematical term stands for, since in a trace the global view is given and the local ones are cut from it; the value of the picture is in reading the same activities at several scales, and the predicate supports that.

26.9 Coupled Activities

Two threads show **Coupled Activities** when a message of one is regularly followed at once by a message of the other: a dispatcher and its workers, a sender and its receiver, a watchdog and the thread it wakes. The coupling is a hand-off visible in time, and the pairs of messages at which it happens are the joints of the trace, where one thread's story passes to another's.

The recognizer looks for consecutive messages of different threads within the `coupling` threshold, a hand-off in time:

```
% coupled_activities(T1, T2, N1-N2): message N2 of
% thread T2 follows message N1 of thread T1 within
% the coupling threshold.
coupled_activities(T1, T2, N1-N2) :-
    msg(N1, Time1, _, T1, _, _),
    N2 is N1 + 1,
    msg(N2, Time2, _, T2, _, _),
    T1 \== T2,
    threshold(coupling, Max),
    round((Time2 - Time1) * 1000) =< Max.

% print_couplings: one line per hand-off.
print_couplings :-
    forall(coupled_activities(T1, T2, N1-N2),
           print_coupling(T1, T2, N1, N2)).

print_coupling(T1, T2, N1, N2) :-
    format("~t~d~3| -> ~d~t~12|~t~d~16|-~d~n",
           [T1, T2, N1, N2]).

?- print_couplings.
100 -> 101     6-7
101 -> 102     7-8
102 -> 103     8-9
103 -> 104     9-10
100 -> 101    11-12
100 -> 102    18-19
100 -> 103    23-24
100 -> 104    27-28
102 -> 100    36-37
100 -> 108    47-48
108 -> 101    52-53
109 -> 100    70-71
true.

?- aggregate_all(count,
   coupled_activities(100, _, _), N).
N = 6.
```

Twelve hand-offs. The main thread hands to each worker once, at the dispatch of each of the first four requests, which is the coupling of a dispatcher with its pool; the housekeeping thread's last message is followed within a millisecond by worker 101 taking request 5, which is consistent with the watchdog having woken it; and thread 109's exception is followed by the main thread stopping the service. Not all twelve are hand-offs in a causal sense: the readiness messages at startup follow one another because the workers started together, and the report engine's last warning happens to precede the arrival of request 5 by a millisecond. Temporal adjacency is the whole of what the predicate tests, and it is evidence of coupling, not proof; the pairs it finds are where an analyst looks for the message content, a request number or a session identifier, that would confirm the hand-off, which the **Data Flow** (page 559) of Section 26.19 does for one value.

26.10 Glued Activity

A **Glued Activity** is a thread of activity made of several unrelated activities joined end to end, because the thread was reused: a pool worker that serves one request, waits, and serves another has one thread of activity and several stories in it, glued at the points where a new piece of work was taken up. Reading the thread as one story misreads it; the pattern is the recognition of the joints.

The joints in this system are the dispatch messages, and the thread's set is split at them:

```
% boundary(N): message N begins a new piece of
% work on its thread.
boundary(N) :-
    msg(N, _, _, _, _, Text),
    sub_string(Text, 0, _, _, "Request "),
    sub_string(Text, _, _, _, " dispatched").

% activity_parts(Set, Parts): Parts divides the
% message set Set at its boundary messages, each
% part beginning with one except possibly the
% first.
activity_parts([], []).
activity_parts([N | Ns], [[N | Part] | Parts]) :-
    take_part(Ns, Part, Rest),
    activity_parts(Rest, Parts).

take_part(Ns, Part, Rest) :-
    (   Ns = [N | Ns1],
        \+ boundary(N)
    ->  Part = [N | Part1],
        take_part(Ns1, Part1, Rest)
    ;   Part = [],
        Rest = Ns
    ).

% glued_activity(TID, Parts): the activity of
% thread TID consists of at least two parts.
glued_activity(TID, Parts) :-
    thread_of_activity(TID, Set),
    activity_parts(Set, Parts),
    Parts = [_, _ | _].

% print_parts(Parts): one line per part with its
% message range and size.
print_parts(Parts) :-
    forall(member(Part, Parts),
           ( Part = [First | _],
             last(Part, Last),
             length(Part, Count),
             format("~t~d~3|-~d~t~7| ~t~d~10|~n",
                    [First, Last, Count]) )).
```

```
?- glued_activity(101, _P), print_parts(_P).
  7-7    1
 12-17   6
 53-58   6
 59-60   2
 61-62   2
 63-64   2
 65-66   2
true.

?- glued_activity(T, _P), length(_P, L).
T = 101,
L = 7 ;
T = 102,
L = 2 ;
T = 103,
L = 2 ;
T = 104,
L = 2 ;
false.
```

Worker 101's twenty-one messages are seven pieces of work: its readiness, request 1 in six messages, request 5 in six, and requests 6 to 9 in two each. Read as one story the thread is confusing, with two queries and two builds for what looks like one worker; read as seven it is a worker doing the same thing six times with one long wait. The other three workers have two parts each, readiness and a single request, and the threads that were never dispatched to have one part and are not glued.

The boundaries are knowledge of the system, as the milestones of Chapter 25 were, and a system that marks the start of a unit of work differently needs its own `boundary/1`. Without any marker the joints can still be guessed from time, splitting at the discontinuities of Chapter 25, and the two methods agree on worker 101's first joint, at the 7695-millisecond gap, and disagree on the rest, where requests 6 to 9 follow one another within a millisecond; the marker is the reliable method, and the gap is the fallback.

26.11 Resume Activity

A **Resume Activity** is the message with which a thread takes up its activity again after a long silence: the point at which a worker that waited receives work, or a thread that was blocked is released. It is the **Discontinuity** (page 513) of Chapter 25 seen from its far end, and what the thread does first on resuming often says what it was waiting for.

```
% resume_activity(TID, N, Gap): message N of
% thread TID follows the thread's previous message
% after a gap of Gap ms above the threshold.
resume_activity(TID, N, Gap) :-
    thread_of_activity(TID, Set),
    nextto(Previous, N, Set),
    time_ms(Previous, T1),
    time_ms(N, T2),
    Gap is T2 - T1,
    threshold(resume, Min),
    Gap > Min.

?- resume_activity(101, N, Gap),
   print_messages([N]).
53  9.006 101 appserver    Request 5 dispatched to
N = 53,
Gap = 7695 ;
false.

?- forall(resume_activity(_, N, _),
          print_row(N)).
37  5.001 100 appserver    Request 5 received: GET
71 14.010 100 appserver    Service stopping
53  9.006 101 appserver    Request 5 dispatched to
19  1.501 102 appserver    Request 2 dispatched to
24  2.001 103 appserver    Request 3 dispatched to
28  2.101 104 appserver    Request 4 dispatched to
true.
```

Every resumption in the trace is a request: the workers resume when dispatched to, and the main thread resumes twice, once to receive request 5 after three seconds in which nothing arrived and once, after five seconds of silence, to stop the service. The main thread's second resumption is the one that matters, since what it did on resuming was shut down, seven milliseconds after thread 109's exception, which is a thread resuming in reaction to another's failure. The predicate finds the resumption and its gap; whether the thread resumed because it was woken, because it was released, or because a timer fired is in what it wrote next, and sometimes nowhere in the trace.

26.12 Background and Foreground Components

Background and Foreground Components is one catalog pattern that distinguishes two roles of the components of a trace. The **background components** of a trace are those whose messages are present throughout, as the setting against which the activity of interest takes place: the framework that logs every request, the monitor that reports every second. They are what a **Trace Mask** (page 496) of Chapter 24 is usually applied to, and recognizing them is the first step to seeing what is not background.

Presence throughout is measured over one-second windows: a component's presence is the share of the windows containing any message in which it has a message, and a component is background when its presence reaches the `background` threshold:

```
% window(From): From is the start of a one-second
% window of the trace.
window(From) :-
    aggregate_all(max(T), msg(_, T, _, _, _, _),
                  End),
    Last is floor(End),
    between(0, Last, From).

% active_window(From): the window has a message.
active_window(From) :-
    window(From),
    To is From + 1,
    msg(_, T, _, _, _, _),
    T >= From, T < To.

% in_window(C, From): the window has a message of
% component C.
in_window(C, From) :-
    window(From),
    To is From + 1,
    msg(_, T, _, _, C, _),
    T >= From, T < To.

% presence(C, Share): C has messages in the share
% Share of the active windows.
presence(C, Share) :-
    aggregate_all(set(C0), msg(_, _, _, _, C0, _),
                  Cs),
    member(C, Cs),
    aggregate_all(count,
                  distinct(F, in_window(C, F)),
                  Own),
    aggregate_all(count,
                  distinct(F, active_window(F)),
                  All),
    Share is Own / All.

print_presence :-
    forall(presence(C, Share),
           format("~a~t~14|~2f~n", [C, Share])).

% background_components(Cs): the components whose
% presence reaches the background threshold.
background_components(Cs) :-
    threshold(background, Min),
    findall(C,
            ( presence(C, Share), Share >= Min ),
            Cs).


?- print_presence.
appserver     0.82
cachelib      0.09
dbclient      0.45
reportengine  0.55
true.

?- background_components(Cs).
Cs = [appserver].
```

The application server is present in nine of the eleven windows that have any message, and it is the one background component at a threshold of three quarters; the report engine, present in six, is not, though its warnings dominate two windows. `distinct/2` inside the count removes the repetitions that `in_window/2` produces when a window has several messages of the component, so that windows are counted and not messages.

The measure is one operational reading of "present throughout," and a component can be background in the catalog's sense without passing it: a monitor that reports every ten seconds is background and is present in a tenth of one-second windows. The window width and the threshold are the two settings, and both are judgments about the system's rhythm; the figure the predicate computes is the share, and the word background is the analyst's.

The **foreground components** of an interval are the components active in it that are not background: those that came into play for the activity in that interval and are therefore the ones the activity is about.

```
% foreground_components(From, To, Cs): the
% components with messages in From..To that are
% not background components.
foreground_components(From, To, Cs) :-
    window_messages(From, To, Set),
    aggregate_all(set(C),
                  ( member(N, Set),
                    msg(N, _, _, _, C, _) ),
                  Present),
    background_components(Background),
    ord_subtract(Present, Background, Cs).

?- foreground_components(1, 2, Cs).
Cs = [dbclient, reportengine].

?- foreground_components(3, 5, Cs).
Cs = [reportengine].

?- foreground_components(6, 9, Cs).
Cs = [].

?- foreground_components(14, 15, Cs).
Cs = [dbclient].
```

In the window from one to two seconds the database client and the report engine are in the foreground, serving requests 1 and 2; from three to five seconds the report engine alone, resizing; from six to nine seconds nothing is in the foreground, the server is queueing requests against its own background; and in the last window the database client is in the foreground, failing. The foreground is defined by `ord_subtract/3` from the background and is as good as that definition; a component that is both the setting and the subject, as the application server is in the stall, is background by presence and foreground by relevance, and the subtraction cannot see the second.

26.13 Guest Component

A **Guest Component** is one that appears once, or nearly once, in an activity otherwise written by other components: a library called a single time in a thread's story, a component with a single message in the whole trace. Its single appearance is why it is noticed, and the question is what it was doing there.

A guest in a message set is a component with exactly one message in a set of at least `host_size` messages:

```
% guest_component(Set, C): component C has exactly
% one message in Set, which has at least host_size
% messages.
guest_component(Set, C) :-
    length(Set, Size),
    threshold(host_size, Min),
    Size >= Min,
    aggregate_all(set(C0),
                  ( member(N, Set),
                    msg(N, _, _, _, C0, _) ),
                  Cs),
    member(C, Cs),
    aggregate_all(count,
                  ( member(N, Set),
                    msg(N, _, _, _, C, _) ),
                  1).

?- thread_of_activity(T, _S),
   guest_component(_S, C).
T = 100,
C = cachelib ;
T = 103,
C = dbclient ;
T = 104,
C = dbclient ;
false.

?- fiber_of_activity(100, component(cachelib), F),
   print_messages(F).
 3  0.011 100 cachelib     Cache initialized, capa
F = [3].
```

The cache library is a guest in the main thread, where it appears once at initialization and never again, and the database client is a guest in workers 103 and 104, each of which used it once. The first is a guest in the whole trace as well, since its one message is its only message, and Chapter 24 found it as a singleton kind; the second is more interesting, because the database client is a regular in the other workers' activities, with paired starts and ends, and a guest in these two, where the one message is an unpaired send and an unpaired reply. The predicate counts appearances, and the reading, that an unpaired appearance of a component that elsewhere appears in pairs is an operation that did not finish, is the analyst's; Chapter 27 treats the pairing under **Opposition Messages** (page 572).

26.14 Watch Thread

A **Watch Thread** is a thread whose messages report on other threads: a watchdog, a monitor, a housekeeping thread that records what the workers are doing. Its own activity is uninteresting and its messages are valuable, because they are observations of threads that may themselves be silent, which makes the watch thread a second witness in the trace.

A thread watches another when its text names the other's identifier after the words `worker` or `thread`; a grammar collects the identifiers so named:

```
% thread_mentions(N, TIDs): the identifiers named
% as worker or thread in the text of message N.
thread_mentions(N, TIDs) :-
    msg(N, _, _, _, _, Text),
    string_lower(Text, Lower),
    string_codes(Lower, Codes),
    phrase(mentions(TIDs), Codes).

mentions(TIDs) -->
    (   ( "worker " ; "thread " ),
        integer(TID)
    ->  { TIDs = [TID | TIDs1] },
        mentions(TIDs1)
    ;   [_]
    ->  mentions(TIDs)
    ;   { TIDs = [] }
    ).

% thread_id(TID): TID is a thread of the trace.
thread_id(TID) :-
    aggregate_all(set(T), msg(_, _, _, T, _, _),
                  TIDs),
    memberchk(TID, TIDs).

% mentioned_thread(N, TID): message N names TID, a
% thread of the trace other than its own.
mentioned_thread(N, TID) :-
    thread_mentions(N, TIDs),
    msg(N, _, _, Own, _, _),
    member(TID, TIDs),
    TID \== Own,
    thread_id(TID).

% watch_thread(TID, Watched): thread TID names at
% least two other threads in its messages.
watch_thread(TID, Watched) :-
    thread_of_activity(TID, Set),
    aggregate_all(set(T),
                  ( member(N, Set),
                    mentioned_thread(N, T) ),
                  Watched),
    Watched = [_, _ | _].
```

```
?- thread_mentions(49, Ts).
Ts = [103].

?- thread_mentions(6, Ts).
Ts = [].

?- mentioned_thread(N, T).
N = 49,
T = 103 ;
N = 50,
T = 104 ;
N = 51,
T = 102 ;
N = 52,
T = 101 ;
false.

?- watch_thread(T, W).
T = 108,
W = [101, 102, 103, 104] ;
false.
```

Thread 108 is the watch thread, naming all four workers in one round of housekeeping, and its four observations are the trace's only direct statements about the state of workers 102, 103, and 104 after they fell silent: two waiting for seven seconds, one at full processor load, one idle. Each agrees with the dump of Part III, which found the first two blocked on critical sections and the third spinning, and the watch thread is the place where the two artifacts can be laid side by side without inference.

The grammar reads a mention only after the words `worker` or `thread`, which is this system's convention; `Worker pool started, threads=4` yields no mention, because the digit after `threads=` does not follow either word, and `worker 102 cpu 100%` yields one and not two, where a grammar that read every number would take the `100` of the percentage for the main thread. The dispatch messages name their own thread and are excluded by the test against `Own`. Two mentions is the threshold for watching, and a thread that names one other thread once is not a watch thread but a thread with one message about another, which may matter as much.

26.15 Error Thread

An **Error Thread** is a thread whose activity contains error messages. Errors are read in the context of the thread that reported them, since the thread's preceding messages are the nearest account of what led to the error, and a trace with several error threads has several problems or one problem seen from several places.

```
% error_word(Word): a word marking an error.
error_word("Error").
error_word("exception").
error_word("failed").
```

```
% error_message(N): message N contains an error
% word.
error_message(N) :-
    msg(N, _, _, _, _, Text),
    once(( error_word(Word),
           sub_string(Text, _, _, _, Word) )).

% error_thread(TID, Errors): Errors is the
% non-empty set of error messages of thread TID.
error_thread(TID, Errors) :-
    thread_of_activity(TID, Set),
    include(error_message, Set, Errors),
    Errors \== [].

?- error_thread(T, E).
T = 100,
E = [43, 45, 47] ;
T = 109,
E = [69, 70].

?- error_thread(109, _E),
   thread_of_activity(109, _S),
   print_messages(_S).
67 14.000 109 appserver    Request 10 received: GE
68 14.001 109 dbclient     Receive start: session=
69 14.002 109 dbclient     Error: buffer copy fail
70 14.003 109 appserver    Unhandled exception 0xc
true.
```

Two error threads: the main thread, with the three queue-depth errors, and thread 109, with the copy failure and the exception. Thread 109's whole activity is four messages, and read as the context of its errors it is a complete account: a request for session data arrived, a receive on the database session began, the buffer copy failed with code `0x3e6`, and the exception followed. The dump of Part III has the same thread stopped in `Buffer::Copy` with a read of address `0x18`, and the trace's `0x3e6` is the error code the dump does not record. `include/3` keeps the messages for which `error_message/1` succeeds and preserves the order of the set; `once/1` inside `error_message/1` counts a message with two error words once. The error words are the system's, as in Chapter 25, and an error reported without one of them is not found.

26.16 Error Distribution

The **Error Distribution** of a trace is how its errors are spread along a dimension: per thread, per component, per interval. Errors concentrated in one thread or one second are one event; errors spread evenly are a condition; and a distribution that shifts over the trace, from one component to another, is a problem propagating.

```
% error_dimension(Dim, N, Value): the value of
% message N along Dim, with second as the integer
% part of its time.
error_dimension(thread, N, TID) :-
    msg(N, _, _, TID, _, _).
error_dimension(component, N, C) :-
    msg(N, _, _, _, C, _).
error_dimension(second, N, S) :-
    msg(N, T, _, _, _, _),
    S is floor(T).

% error_distribution(Dim, Dist): Dist is the list
% of Value-Count pairs of error messages along
% Dim.
error_distribution(Dim, Dist) :-
    findall(V-N,
            ( error_message(N),
              error_dimension(Dim, N, V) ),
            Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Groups),
    maplist(count_group, Groups, Dist).

?- error_distribution(thread, D).
D = [100-3, 109-2].

?- error_distribution(component, D).
D = [appserver-4, dbclient-1].

?- error_distribution(second, D).
D = [7-1, 8-1, 9-1, 14-2].
```

Three errors a second apart in the main thread, in the windows beginning at seven, eight, and nine seconds, and two in thread 109 in the window beginning at fourteen: two events, not one, and the distribution by second separates them where the count by component, four against one, does not. `count_group/2` from Section 26.4 turns each group into its size, `floor/1` gives the second of a timestamp, and the three dimensions share the one predicate, as the trace dimensions of Chapter 24 did. The distribution counts error messages as `error_message/1` defines them, and an error that the system reports without an error word, or reports once and then repeats as a warning, is distributed accordingly.

26.17 Message Interleave

Message Interleave is the alternation of messages from different threads within a region: how often consecutive messages change thread. A region in which every message is from a different thread than the last is several threads running at once, and their stories must be separated before they can be read; a region with no changes is one thread working alone, and the trace there reads as it stands.

```
% message_interleave(From, To, Count, Switches):
% of the Count messages in From..To, Switches
% consecutive pairs are from different threads.
message_interleave(From, To, Count, Switches) :-
    window_messages(From, To, Set),
    length(Set, Count),
    aggregate_all(count,
                  ( nextto(N1, N2, Set),
                    msg(N1, _, _, T1, _, _),
                    msg(N2, _, _, T2, _, _),
                    T1 \== T2 ),
                  Switches).

?- message_interleave(1.5, 2.2, C, S).
C = 13,
S = 5.

?- message_interleave(9.0, 9.4, C, S).
C = 21,
S = 2.

?- message_interleave(0, 15, C, S).
C = 71,
S = 18.
```

Between 1.5 and 2.2 seconds, thirteen messages change thread five times, as the main thread dispatches requests 2, 3, and 4 and the three workers begin them; between 9.0 and 9.4 seconds, twenty-one messages change thread twice, the main thread handing to the watchdog and the watchdog to worker 101, after which the worker writes fourteen messages alone. The first region is read by thread, the second as it stands. Over the whole trace there are eighteen changes in seventy-one messages, which is a trace of moderate interleave and the reason the thread viewpoint of Chapter 24 was worth having. The measure counts changes of thread between consecutive messages and does not say which threads alternate; the theatre of Section 26.4 says who was present, and the two together describe the region.

26.18 Dominant Event Sequence

The **Dominant Event Sequence** of a trace is the sequence of message kinds that recurs most often as a unit of work: what the system normally does, against which the units that deviate from it are read. In a server it is the lifecycle of an ordinary request, and the request whose sequence is not the dominant one is the request to look at. Chapter 28 treats the recurring subsequence formally under **Motif** (page 607); here the units are the parts of Section 26.10.

The kind of a message is the number-free text of Chapter 24, memoized again with `table/1`, and the **event sequence** of a part is the list of its kinds:

```
:- table message_kind/2.
```

```
% message_kind(N, Kind): the text of message N
% with every number replaced by # (Chapter 24).
message_kind(N, Kind) :-
    msg(N, _, _, _, _, Text),
    string_codes(Text, Codes),
    phrase(kind(KindCodes), Codes),
    string_codes(Kind, KindCodes).

kind(Ks) -->
    (   number_token
    ->  { Ks = [0'# | Ks1] },
        kind(Ks1)
    ;   [C]
    ->  { Ks = [C | Ks1] },
        kind(Ks1)
    ;   { Ks = [] }
    ).

number_token -->
    (   "0x"
    ->  xinteger(_)
    ;   integer(_)
    ).

% event_sequence(TID, Part, Kinds): Part is a part
% of thread TID's activity that begins at a
% boundary, and Kinds the sequence of its kinds.
event_sequence(TID, Part, Kinds) :-
    thread_of_activity(TID, Set),
    activity_parts(Set, Parts),
    member(Part, Parts),
    Part = [First | _],
    boundary(First),
    maplist(message_kind, Part, Kinds).

% sequence_count(Kinds, Count): the event sequence
% Kinds occurs Count times.
sequence_count(Kinds, Count) :-
    distinct(Kinds, event_sequence(_, _, Kinds)),
    aggregate_all(count,
                  event_sequence(_, _, Kinds),
                  Count).

% dominant_event_sequence(Kinds, Count): the most
% frequent event sequence and its count.
dominant_event_sequence(Kinds, Count) :-
    aggregate_all(max(C, Ks),
                  sequence_count(Ks, C),
                  max(Count, Kinds)).

% print_kinds(Kinds): one kind per line.
print_kinds(Kinds) :-
    forall(member(Kind, Kinds),
           ( truncated(Kind, 46, Short),
             format("  ~s~n", [Short]) )).
```

```
% print_sequences(Max): the Max most frequent
% event sequences with their counts.
print_sequences(Max) :-
    forall(limit(Max,
                 order_by([desc(Count)],
                          sequence_count(Kinds,
                                         Count))),
           ( format("count ~d~n", [Count]),
             print_kinds(Kinds) )).

?- dominant_event_sequence(_K, Count),
   print_kinds(_K).
  Request # dispatched to worker #
  Request # completed: #, # ms
Count = 4.

?- print_sequences(2).
count 4
  Request # dispatched to worker #
  Request # completed: #, # ms
count 2
  Request # dispatched to worker #
  Query start: SELECT report #
  Query end: # row, # ms
  Build start: report #
  Build end: report #, # nodes, # ms
  Request # completed: #, # ms
true.

?- aggregate_all(count, sequence_count(_, 1), N).
N = 3.
```

The dominant sequence is dispatch and completion with nothing between, four times, which is the health checks served from the queue in an instant; the full request lifecycle of six kinds occurs twice, for the two reports built to completion; and three sequences occur once, those of the three workers whose requests never completed. The dominant sequence here is dominant by count and uninformative, since it is the degenerate case of the lifecycle, and the sequence of two occurrences is the one an analyst would call normal; dominance by count is one reading of the pattern, and the predicate gives the counts so that the analyst can take another. `aggregate_all/3` with `max(C, Ks)` takes the count and the sequence as its witness, as Chapter 9 showed, and on a tie reports one of the tied sequences without saying there was a tie; `print_sequences/1` shows the ranking and is the safer report.

26.19 Data Flow

Data Flow follows a value through the trace: a session identifier, a handle, a request number that appears in messages of several threads and components, and whose appearances are the path the data took through the system. Where the adjoint thread follows an attribute of messages, data flow follows a value inside them, and it reaches across threads and components alike.

The values in `trace1.pl` are written as `key=value` fields, and `split_string/4` of Chapter 8 takes them apart:

```
% key_value(N, Key, Value): the text of message N
% contains the field Key=Value.
key_value(N, Key, Value) :-
    msg(N, _, _, _, _, Text),
    split_string(Text, " ,", "", Parts),
    member(Part, Parts),
    split_string(Part, "=", "", [Key, Value]).

% data_flow(Key, Value, Trail): Trail is the set
% of messages carrying the field Key=Value.
data_flow(Key, Value, Trail) :-
    distinct(Key-Value, key_value(_, Key, Value)),
    aggregate_all(set(N),
                  key_value(N, Key, Value),
                  Trail).

?- data_flow(K, V, T), T = [_, _ | _].
K = "session",
V = "71",
T = [5, 26, 29, 68] ;
false.

?- data_flow("session", "71", _T),
   print_messages(_T).
 5  0.212 100 dbclient     Connected to db01:5432,
26  2.003 103 dbclient     Send start: session=71,
29  2.102 104 dbclient     Reply received: session
68 14.001 109 dbclient     Receive start: session=
true ;
false.

?- data_flow("bytes", V, T).
V = "1024",
T = [26] ;
V = "512",
T = [29] ;
false.
```

One value flows: the database session 71, opened by the main thread at startup, used by worker 103 to send a kilobyte, by worker 104 to receive half a kilobyte, and by thread 109 to begin a receive that failed. Four threads and one connection, and the trail says what the threads of activity could not: that the three workers whose stories end badly in the dump of Part III were all operating on the same database session, which is the kind of shared resource a deadlock and a corrupted buffer have in common. The trail is a message set and prints as one; `distinct/2` enumerates each field once.

The flow is as visible as the system made it. A value that appears as `session=71` in one component and as `sid 71` in another is two values to the predicate, and a value passed without being logged leaves no trail at all; the splitting at `=` is the convention of this system, and another needs its own `key_value/3`. The direction of the flow, which message produced the value and which consumed it, is in the text of the messages and not in the predicate.

26.20 Use Case Trail

A **Use Case Trail** is the set of messages that belong to one use case of the system, one request from arrival to completion, gathered across the threads and components that served it. It is the unit an analyst most often wants and the trace least often gives directly, and reconstructing it combines the adjoint thread by request with the thread of activity of the worker that served it.

The trail of a request is its adjoint thread together with the segment of the serving worker's activity from the dispatch to the completion, or, if there is no completion, to the worker's next piece of work or the end of its activity. The segment is a part in the sense of Section 26.10, cut at the completion message when there is one, with `append/3` locating the start and the end in the worker's set:

```
% received(R, N), dispatched(R, N, Worker),
% completed(R, N): the lifecycle messages of R.
received(Request, N) :-
    request_id(N, Request),
    msg(N, _, _, _, _, Text),
    sub_string(Text, _, _, _, " received").

dispatched(Request, N, Worker) :-
    request_id(N, Request),
    msg(N, _, _, Worker, _, Text),
    sub_string(Text, _, _, _, " dispatched").

completed(Request, N) :-
    request_id(N, Request),
    msg(N, _, _, _, _, Text),
    sub_string(Text, _, _, _, " completed").

% use_case_trail(Request, Trail): Trail is the set
% of messages serving Request across threads.
use_case_trail(Request, Trail) :-
    received(Request, N0),
    (   dispatched(Request, Start, Worker)
    ->  true
    ;   Start = N0,
        msg(N0, _, _, Worker, _, _)
    ),
    thread_of_activity(Worker, Set),
    append(_, [Start | After], Set),
    take_part(After, Part, _),
    (   completed(Request, End)
    ->  append(Before, [End | _], [Start | Part]),
        append(Before, [End], Segment)
    ;   Segment = [Start | Part]
    ),
    adjoint_thread_of_activity(request(Request),
                               Adjoint),
    ord_union(Adjoint, Segment, Trail).
```

```
?- use_case_trail(5, _T), print_messages(_T).
37  5.001 100 appserver    Request 5 received: GET
38  5.002 100 appserver    No idle worker, request
53  9.006 101 appserver    Request 5 dispatched to
54  9.007 101 dbclient     Query start: SELECT rep
55  9.060 101 dbclient     Query end: 1 row, 53 ms
56  9.061 101 reportengine Build start: report 19
57  9.300 101 reportengine Build end: report 19, 2
58  9.301 101 appserver    Request 5 completed: 20
true ;
false.

?- use_case_trail(2, _T), print_messages(_T).
18  1.500 100 appserver    Request 2 received: GET
19  1.501 102 appserver    Request 2 dispatched to
20  1.502 102 dbclient     Query start: SELECT rep
21  1.550 102 dbclient     Query end: 1 row, 48 ms
22  1.551 102 reportengine Build start: report 18
31  2.500 102 reportengine Warning: table resize,
32  3.000 102 reportengine Warning: table resize,
33  3.500 102 reportengine Warning: table resize,
34  4.000 102 reportengine Warning: table resize,
35  4.500 102 reportengine Warning: table resize,
36  5.000 102 reportengine Warning: table resize,
true ;
false.

?- use_case_trail(10, T).
T = [67, 68, 69, 70] ;
false.

?- forall(use_case_trail(R, _T),
          ( length(_T, L),
            format("request ~d: ~d messages~n",
                   [R, L]) )).
request 1: 7 messages
request 2: 11 messages
request 3: 4 messages
request 4: 4 messages
request 5: 8 messages
request 6: 4 messages
request 7: 4 messages
request 8: 3 messages
request 9: 3 messages
request 10: 4 messages
true.
```

Request 5's trail is eight messages across two threads, and it is the complete story of one request that was queued for four seconds and then served in three hundred milliseconds. Request 2's trail has no completion and runs to the end of worker 102's activity: a query, the start of a build, and six warnings that the build's table was doubling, which is where the thread's story and the dump's account of its processor time meet. Request 10's trail is thread 109's failure, found without a dispatch message because the receiving thread served it directly, which the if-then-else on `dispatched/3` provides for. The trails of requests 8 and 9 are three messages, having no queued message, which Chapter 27 returns to under **Missing Message** (page 575).

The predicate is the most knowledge-laden in this chapter. It knows that a request is received, possibly dispatched to a worker, and completed, and that the worker's messages between dispatch and completion belong to the request; all of that is this system's protocol, and another system's trail predicate encodes another protocol. It leaves a choice point, which the toplevel shows as `true ;` before `false.`, because `received/2` searches the trace and does not know that a request is received once; `once/1` around it would remove the choice point at the cost of the enumerating mode that the last query uses. The trail is the unit over which Chapter 25's delay dynamics should be read, request by request, and the unit Chapter 28 compares across requests under **Event Sequence Order** (page 603) and **Motif** (page 607).

Chapter 27. Messages

The patterns of Chapters 24 to 26 treat a software trace as a structure: a set with dimensions, a timeline, a bundle of threads of activity. The patterns of this chapter treat its unit, the **message**, and ask what a single message is, what it says, what it fails to say, and how it is related to other messages by its text alone. Most of them are formulated as predicates over the text of `msg/6` facts, which is why the string predicates of Chapter 8, the collectors of Chapter 9, and the grammars of Chapter 12 do most of the work here, and why several sections define a small log excerpt of their own: a trace of 71 ordinary messages has no stack trace in it, no messages in another language, and no message written over another.

Every section follows the shape of Part IV. A paragraph states the pattern's intent in this book's words, a predicate recognizes instances of it in `trace1.pl`, a transcript exercises the predicate, and a closing remark says what the formalization leaves out. The recognizers are deliberately plain: a pattern that an analyst recognizes by reading is here a test on strings, and the gap between the two is the subject of the closing remarks.

The chapter begins with the helpers every section uses. `print_messages/1` prints a list of messages by number, with the thread, because the toplevel's rendering of a `msg/6` term is too wide for the page and hides the text behind the other five arguments; `text_of/2` is the text of a message; `thread_sequence/2` is the ordered list of a thread's message numbers, which Chapter 26 called a thread of activity; and `trace_thread/1` enumerates the threads of the trace once each. The flag that hides the bindings of variables beginning with an underscore is set for the whole chapter, as in Chapter 9.

```
:- [trace1].

% print_messages(+Ns): print message N for each N
% in Ns, with its thread.
print_messages(Ns) :-
    forall(member(N, Ns), print_msg(N)).

print_msg(N) :-
    msg(N, _, _, TID, _, Text),
    format("~t~d~3| ~d ~s~n", [N, TID, Text]).

% text_of(?N, ?Text): the text of message N.
text_of(N, Text) :-
    msg(N, _, _, _, _, Text).

% trace_thread(-TID): TID is a thread of the
% trace; each thread once.
trace_thread(TID) :-
    distinct(TID, msg(_, _, _, TID, _, _)).

% thread_sequence(+TID, -Ns): the numbers of the
% messages of thread TID, in order.
thread_sequence(TID, Ns) :-
    findall(N, msg(N, _, _, TID, _, _), Ns0),
    sort(Ns0, Ns).
```

```
?- set_prolog_flag(toplevel_print_anon, false).
true.

?- print_messages([1, 70, 71]).
  1 100 Service starting, version 3.2.1.0
 70 109 Unhandled exception 0xc0000005, thread 109
 71 100 Service stopping
true.

?- thread_sequence(109, Ns).
Ns = [67, 68, 69, 70].
```

27.1 Error Message

An **Error Message** is a message that its own writer marked as reporting an error: by a severity field, a prefix, a keyword, or an error code. It is the first thing an analyst searches for and the pattern whose instances are easiest to find mechanically, since the marking is in the text. Its value is also the most often overestimated: the catalog's point about error messages is that their presence is not the diagnosis, and that the error that matters is frequently not marked at all, which is the **Hidden Error** (page 584) of Section 27.17, or is marked but harmless, which is the **False Positive Error** (page 589) of Section 27.21.

In `trace1.pl` the marking is the prefix `Error:`. A prefix test is `sub_string/5` with the position before the substring bound to 0, as Chapter 8 showed; it is semideterministic, so `error_message/1` with the number bound is a test and with it unbound an enumerator.

```
% error_message(?N): message N is marked as an
% error by the component that wrote it.
error_message(N) :-
    text_of(N, Text),
    sub_string(Text, 0, _, _, "Error:").

?- findall(N, error_message(N), Ns).
Ns = [43, 45, 47, 69].

?- error_message(70).
false.

?- print_messages([43, 69]).
 43 100 Error: queue depth 3 exceeds threshold 2
 69 109 Error: buffer copy failed, code=0x3e6
true.
```

Four messages are marked, three of them the same message with a growing number, which Section 27.4 reads as a counter. Message 70, the unhandled exception that ends the service, is not an Error Message in this sense, because its writer did not mark it; the predicate recognizes the marking, not the severity. Every other formalization of the pattern has the same limit in a different place: a severity field is the writer's judgment, a list of keywords is the analyst's, and neither is the trace's.

27.2 Significant Event

A **Significant Event** is a message the analyst singles out as marking the moment the analysis turns on: the first error, a state change, a message that should not be there, or the last message before a gap. It is where reading begins, forward from it to the consequences and backward to the causes, and it is a judgment: the same trace has different significant events for different questions. A predicate can encode the criteria an analyst would apply, and the second argument of `significant_event/2` names which criterion recognized the message, so that the output records why a message was selected and not only that it was.

```
% significant_event(?N, ?Why): message N is
% significant for the reason Why.
significant_event(N, first_error) :-
    aggregate_all(min(E), error_message(E), N).
significant_event(N, exception) :-
    text_of(N, Text),
    sub_string(Text, _, _, _, "exception").
significant_event(N, lifecycle) :-
    text_of(N, Text),
    sub_string(Text, 0, _, _, "Service").
significant_event(N, saturation) :-
    text_of(N, Text),
    sub_string(Text, 0, _, _, "No idle worker").

?- forall(significant_event(N, Why),
          format("~t~d~3| ~a~n", [N, Why])).
 43 first_error
 70 exception
  1 lifecycle
 71 lifecycle
 38 saturation
 40 saturation
 42 saturation
true.

?- significant_event(N, first_error),
   print_messages([N]).
 43 100 Error: queue depth 3 exceeds threshold 2
N = 43.
```

The first clause uses `aggregate_all/3` with `min/1` to pick the earliest marked error, which is the usual meaning of “the first error”; the others are text tests. The criteria are the analyst’s, written down, and that is both the strength and the limit of the formalization: the predicate finds every message that satisfies a stated criterion and none that would strike a reader as odd without satisfying one. Message 48, the first message of the housekeeping thread after nine seconds of silence from it, is significant in `trace1.pl` because of what follows it, and no criterion above selects it. The predicate is a checklist, not a reader.

27.3 Message Invariant

A **Message Invariant** is the part of a message's text that does not change between its occurrences: the format string behind the message, with the variable parts, numbers, identifiers, and sizes, removed. Two messages with the same invariant were written by the same trace statement, or by two statements that happen to agree, and the invariant is therefore the key by which messages are grouped into kinds, counted, searched for across traces, and compared with each other. Most of the patterns of this chapter and the next are stated in terms of invariants rather than texts.

The book's operational definition replaces every maximal run of characters that begins with a digit and continues with letters, digits, and dots by a single `#`. This blanks decimal numbers, version numbers, hexadecimal codes written with the `0x` prefix, and the digits of names such as `db01`, and keeps everything else. The replacement is a recursion over the character codes of the text with the if-then-else of Chapter 7, and the result is a string again.

```
% invariant(+Text, -Inv): Inv is Text with every
% number replaced by #.
invariant(Text, Inv) :-
    string_codes(Text, Codes),
    blank_numbers(Codes, Skeleton),
    string_codes(Inv, Skeleton).

blank_numbers([], []).
blank_numbers([C|Cs], Skeleton) :-
    (   code_type(C, digit)
    ->  skip_number(Cs, Rest),
        Skeleton = [0'#|Skeleton1],
        blank_numbers(Rest, Skeleton1)
    ;   Skeleton = [C|Skeleton1],
        blank_numbers(Cs, Skeleton1)
    ).

% skip_number(+Cs, -Rest): Rest is Cs without its
% leading letters, digits, and dots.
skip_number(Cs, Rest) :-
    (   Cs = [C|Cs1],
        (   code_type(C, alnum)
        ;   C == 0'.
        )
    ->  skip_number(Cs1, Rest)
    ;   Rest = Cs
    ).

% message_invariant(?N, ?Inv): Inv is the
% invariant of the text of message N.
message_invariant(N, Inv) :-
    text_of(N, Text),
    invariant(Text, Inv).
```

```
% prefix_of(+Prefix, ?N): the invariant of
% message N begins with Prefix.
prefix_of(Prefix, N) :-
    message_invariant(N, Inv),
    string_concat(Prefix, _, Inv).

?- message_invariant(16, Inv).
Inv = "Build end: report #, # nodes, # ms".

?- message_invariant(70, Inv).
Inv = "Unhandled exception #, thread #".

?- aggregate_all(count,
      distinct(I, message_invariant(_, I)), C).
C = 33.

?- forall(limit(5, order_by([desc(C), asc(I)],
      aggregate(count, N^message_invariant(N, I),
                C))),
      format("~t~d~3| ~s~n", [C, I])).
  9 Request # dispatched to worker #
  6 Request # completed: #, # ms
  6 Warning: table resize, # -> #
  4 Request # received: GET /health
  4 Worker # ready
true.
```

The 71 messages have 33 invariants, and the five most frequent account for 29 of them. `prefix_of/2` is the test the later sections use for “a message of such-and-such kind”: a kind is named by a prefix of its invariant, so that `"Request # received"` names all the request arrivals whatever their URL, which the full invariant would separate. The `code_type/2` test `alnum` is what makes `0xc0000005` one run, since `x` and `c` are letters; the same rule turns the `01` of `db01` into `#`, and the invariant of a message that contains a word with a digit in its middle is not the format string the programmer wrote. The definition is also blind to variable parts that are not numbers: two messages that differ only in a path or a user name have different invariants and are counted as two kinds. A better invariant needs the format strings themselves, which the trace does not contain.

27.4 Counter Value

A **Counter Value** is a number in a message that changes from one occurrence of the message to the next and whose sequence of values tells a story the individual messages do not: a queue that grows, a table that doubles, a duration that shrinks. The pattern is recognized when the same invariant occurs several times with a varying number in the same position; the predicate collects the series as `N-Value` pairs so that each value remains attached to the message it came from.

```
% numbers_in(+Text, -Numbers): the numbers in Text
% that number_string/2 accepts, in order.
numbers_in(Text, Numbers) :-
    split_string(Text, " ,:=%", " ,:=%", Parts),
    convlist([P, V]>>number_string(V, P), Parts,
             Numbers).

% counter_value(?Inv, ?I, -Series): Series is the
% list of N-Value pairs of the I-th number of the
% messages with invariant Inv; it has at least two
% elements and is not constant.
counter_value(Inv, I, Series) :-
    distinct(Inv, message_invariant(_, Inv)),
    findall(N-Vs,
            ( message_invariant(N, Inv),
              text_of(N, Text),
              numbers_in(Text, Vs)
            ),
            Rows),
    Rows = [_-Vs0|_],
    length(Vs0, K),
    between(1, K, I),
    findall(N-V,
            ( member(N-Vs, Rows),
              nth1(I, Vs, V)
            ),
            Series),
    Series = [_, _|_],
    pairs_values(Series, [V0|Values]),
    \+ maplist(==(V0), Values).

% increasing(+Values): each value exceeds the one
% before it.
increasing([_]).
increasing([A, B|Rest]) :-
    A < B,
    increasing([B|Rest]).

% print_series(+Series): one line per pair.
print_series(Series) :-
    forall(member(N-V, Series),
           format("~t~d~3| ~w~n", [N, V])).


?- message_invariant(43, _I),
   counter_value(_I, I, S).
I = 1,
S = [43-3, 45-4, 47-5] ;
false.

?- message_invariant(31, _I),
   counter_value(_I, 2, _S),
   pairs_values(_S, _Vs),
   increasing(_Vs),
   last(_Vs, Last).
Last = 524288 ;
false.
```

```
?- message_invariant(58, _I),
   counter_value(_I, 3, _S),
   print_series(_S).
 17 309
 58 4300
 60 3302
 62 2305
 64 1307
 66 309
true ;
false.
```

`convlist/3` is `maplist/3` that drops the elements for which the goal fails, here every part that is not a number; `number_string/2` reads `0x3e6` as 998 and rejects `3.2.1.0`, so the numbers of a message are not quite the `#` marks of its invariant, which is harmless as long as all messages of one invariant agree, as they do. The queue depth grows by one per health request; the table sizes double, and `increasing/1` confirms the direction; the completion times of requests 5 to 9 fall by about a second each, which Chapter 25 read as **Delay Dynamics** (page 522), and here they are the third number of the completion message. The second number of that message, the status 200, is constant and is not reported, which is the `\+ maplist(==(V0), Values)` at the end. The trailing `false.` in each answer is the choice point left by `distinct/2` and `between/3`, and the predicate is correct but not deterministic; the formalization does not know which number is a counter and which an identifier, and the request numbers in the first position of the completion message form a series too.

27.5 Linked Messages

Linked Messages are messages, possibly far apart and on different threads, that mention the same thing: a session number, a request number, a handle, an address. The shared datum links them into a chain that an analyst follows from one to the next, which is how a request is followed from the thread that received it to the thread that served it, and how a value is traced from where it was created to where it failed. The recognizer needs a definition of “the same thing”, and the book’s is a **key**: a `name=value` token, or a word from a short list of identifier names followed by a number. Which words name identifiers is knowledge about the application, stated as facts.

```
% id_word(?Word): Word, lowercased, introduces an
% identifier in this application's messages.
id_word("request").
id_word("report").
id_word("worker").
id_word("thread").
id_word("session").

% message_key(?N, ?Key): Key is a name=value token
% of message N, or a Name-Number pair of adjacent
% words where Name is an id_word/1.
message_key(N, Key) :-
    text_of(N, Text),
    split_string(Text, " ,:", " ,:", Words),
    (   member(Key, Words),
        sub_string(Key, _, _, _, "=")
```

```
    ;   nextto(Name, Number, Words),
        number_string(_, Number),
        string_lower(Name, Lower),
        id_word(Lower),
        Key = Lower-Number
    ).

% linked_messages(?N1, ?N2, ?Key): messages N1 and
% N2, N1 before N2, share Key.
linked_messages(N1, N2, Key) :-
    message_key(N1, Key),
    message_key(N2, Key),
    N1 < N2.
```

```
?- message_key(12, Key).
Key = "request"-"1" ;
Key = "worker"-"101" ;
false.

?- linked_messages(26, N, Key).
N = 29,
Key = "session=71" ;
N = 68,
Key = "session=71" ;
false.

?- linked_messages(N, 26, Key).
N = 5,
Key = "session=71" ;
false.

?- print_messages([5, 26, 29, 68]).
  5 100 Connected to db01:5432, session=71
 26 103 Send start: session=71, bytes=1024
 29 104 Reply received: session=71, bytes=512
 68 109 Receive start: session=71
true.
```

`nextto/3` enumerates adjacent pairs of a list, which is how “a word followed by a number” is found. The database session 71 links four messages on four threads, from the connection at startup through the send on worker 103 and the reply on worker 104 to the receive on thread 109 that precedes the failure; Chapter 26 followed the same value as a **Data Flow** (page 559), and Section 27.6 asks what it means that so many threads touch it. The limits are those of the key definition: `GET /report/17` is not linked to `SELECT report 17` because the path is one token, and a key that occurs once links nothing. Links are also only as good as the identifiers are unique; `worker 101` names the same thread throughout, but a request number is reused when a service restarts, and the chain would then join two requests.

27.6 Shared Point

A **Shared Point** is a datum, a resource, or a message that several threads of activity have in common: the place where otherwise separate activities meet. Where **Linked Messages** (page 570) follows one

chain, Shared Point asks which keys are touched by more than one thread, because a resource shared by threads that later stall is where the stall is likely to be. The predicate groups the threads by key with `aggregate_all(set(...))`, which collects without the free-variable grouping of `setof/3` that Chapter 9 warned about.

```
% shared_point(?Key, -TIDs): Key occurs in
% messages of at least two threads, TIDs.
shared_point(Key, TIDs) :-
    distinct(Key, message_key(_, Key)),
    aggregate_all(set(TID),
                  ( message_key(N, Key),
                    msg(N, _, _, TID, _, _)
                  ),
                  TIDs),
    TIDs = [_, _|_].

?- shared_point(Key, TIDs), length(TIDs, L),
   L > 2.
Key = "session=71",
TIDs = [100, 103, 104, 109],
L = 4 ;
false.

?- shared_point("session"-"9", TIDs).
TIDs = [103, 104] ;
false.

?- aggregate_all(count, shared_point(_, _), C).
C = 15.
```

Fifteen keys are shared, most of them a request number shared by the main thread that received the request and the worker that served it, which is the normal shape of a dispatch. Two stand out. Database session 71 is touched by four threads, among them the two workers that never complete and the thread that fails. Application session 9 is shared by exactly workers 103 and 104, the two threads that `dump1.pl` shows waiting on each other's critical sections, and nothing in the trace says so except this coincidence of keys. The predicate finds the coincidence; whether the shared datum is the shared lock is a question for the dump, which Chapter 28 asks in its last section. A shared point found from text is a hypothesis about the program's data, not a fact about its synchronization.

27.7 Opposition Messages

Opposition Messages are pairs of message kinds that bracket an activity: start and end, open and close, send and reply, enter and leave. A matched pair bounds an interval; an unmatched opening is an activity that did not finish, or whose finish was not traced, and an unmatched closing is an activity whose beginning lies before the trace. The pairs themselves are knowledge about the application, stated as facts, together with the scope in which an opening and a closing belong to the same activity: the same thread for kinds that do not carry an identifier, or a shared key for those that do.

```
% opposition(?Open, ?Close, ?Scope): messages
% beginning with Open and Close bracket an
```

```
% activity; Scope is thread or a key word.
opposition("Query start", "Query end", thread).
opposition("Build start", "Build end", thread).
opposition("Service starting", "Service stopping",
           thread).
opposition("Send start", "Reply received",
           "session").
opposition("Request # received",
           "Request # completed", "request").
opposition("Request # dispatched",
           "Request # completed", "request").

% key_word(+Key, -Word): the name part of a key.
key_word(Name-_, Name).
key_word(Key, Name) :-
    string(Key),
    sub_string(Key, B, _, _, "="),
    sub_string(Key, 0, B, _, Name).

% in_scope(+Scope, +N1, +N2): N1 and N2 belong to
% the same activity under Scope.
in_scope(thread, N1, N2) :-
    msg(N1, _, _, TID, _, _),
    msg(N2, _, _, TID, _, _).
in_scope(Word, N1, N2) :-
    string(Word),
    once(( message_key(N1, Key),
           key_word(Key, Word),
           message_key(N2, Key) )).

% opposition_messages(?N1, ?N2): N2 is the closing
% message matching the opening message N1.
opposition_messages(N1, N2) :-
    opposition(Open, Close, Scope),
    prefix_of(Open, N1),
    prefix_of(Close, N2),
    N1 < N2,
    in_scope(Scope, N1, N2),
    \+ ( prefix_of(Open, N3), N1 < N3, N3 < N2,
         in_scope(Scope, N1, N3) ),
    \+ ( prefix_of(Close, N3), N1 < N3, N3 < N2,
         in_scope(Scope, N1, N3) ).

% unmatched_open(?N): N opens an activity that no
% message closes.
unmatched_open(N) :-
    opposition(Open, _, _),
    prefix_of(Open, N),
    \+ opposition_messages(N, _).

?- opposition_messages(13, N).
N = 14 ;
false.

?- opposition_messages(N, 17).
N = 11 ;
N = 12 ;
false.
```

```
?- opposition_messages(26, N).
N = 29 ;
false.

?- findall(N, unmatched_open(N), _Ns),
   sort(_Ns, Ns).
Ns = [18, 19, 22, 23, 24, 27, 28, 67].

?- print_messages([18, 22, 23, 27, 67]).
 18 100 Request 2 received: GET /report/18
 22 102 Build start: report 18
 23 100 Request 3 received: POST /session/commit
 27 100 Request 4 received: GET /session/9
 67 109 Request 10 received: GET /session/9/data
true.
```

The two negations say that no opening or closing of the same kind and scope lies between the pair, which is what makes the match the innermost one and keeps the two `Query start` messages of one thread from pairing with each other's ends. `once/1` in `in_scope/3` makes the key test a test, since two messages can share more than one key. The unmatched openings are the diagnosis of `trace1.pl` in eight numbers: requests 2, 3, 4, and 10 were received and never completed, the report build of request 2 never ended, and the dispatches to workers 102, 103, and 104 never returned. The send on worker 103 did find its reply, on another thread, because the scope is the session and not the thread. What the predicate cannot say is whether an unmatched opening is a hang or a trace that ended first; message 67 is followed by an exception, which settles it, and messages 22 and 24 are not, and only the dump settles those.

27.8 Characteristic Message Block

A **Characteristic Message Block** is a sequence of message kinds that recurs as a unit and identifies an activity: the handful of messages that every successful request produces, in their fixed order, so that a reader who sees the first expects the rest. Once the block is known, each occurrence can be found as a unit, the blocks can be counted, and a block that breaks off can be recognized, which is Section 27.9. The block is knowledge, stated as a list of prefixes; its instances are runs of consecutive messages of one thread whose invariants begin with the prefixes in order, found with `append/3` as the sublist enumerator of Chapter 5.

```
% characteristic_block(?Name, ?Prefixes): the
% block Name is the sequence of message kinds
% Prefixes.
characteristic_block(report_request,
    [ "Request # dispatched", "Query start",
      "Query end", "Build start", "Build end",
      "Request # completed" ]).
```

```
% characteristic_message_block(?Name, ?TID, -Ns):
% the consecutive messages Ns of thread TID form
% an instance of the block Name.
characteristic_message_block(Name, TID, Ns) :-
    characteristic_block(Name, Prefixes),
    trace_thread(TID),
    thread_sequence(TID, Seq),
    append(_, Rest, Seq),
    append(Ns, _, Rest),
    maplist(prefix_of, Prefixes, Ns).

?- characteristic_message_block(report_request,
      TID, Ns).
TID = 101,
Ns = [12, 13, 14, 15, 16, 17] ;
TID = 101,
Ns = [53, 54, 55, 56, 57, 58] ;
false.

?- print_messages([53, 54, 55, 56, 57, 58]).
 53 101 Request 5 dispatched to worker 101
 54 101 Query start: SELECT report 19
 55 101 Query end: 1 row, 53 ms
 56 101 Build start: report 19
 57 101 Build end: report 19, 2402 nodes, 239 ms
 58 101 Request 5 completed: 200, 4300 ms
true.
```

Worker 101 produced the block twice, for requests 1 and 5, and no other thread produced it at all. `maplist/3` applies `prefix_of/2` pairwise to the prefixes and the candidate messages and fails as soon as one does not match, which is what makes the double `append/3` enumeration affordable. The block requires the messages to be consecutive on the thread, so a block interleaved with unrelated messages of the same thread, a warning in the middle of a request, is not recognized, and the health requests, which complete without a query, are a different block the knowledge base does not have.

27.9 Missing Message

A **Missing Message** is a message that the analyst expects at a point in the trace and does not find: the next message of a **Characteristic Message Block** (page 574) that never comes, the end of an activity whose start is present, the acknowledgment that should follow a request. The absence is the evidence, and absence is only visible against an expectation, which is why the pattern is formalized relative to a block. The predicate walks the block as far as the thread's messages follow it and reports the first prefix that the next message does not match.

```
% missing_message(?Name, ?TID, -Ns, -Expected):
% the messages Ns of thread TID match a proper
% prefix of the block Name, at least two messages
% long, and the block's next kind, Expected, does
% not follow.
missing_message(Name, TID, Ns, Expected) :-
    characteristic_block(Name, Prefixes),
```

```
    trace_thread(TID),
    thread_sequence(TID, Seq),
    append(_, Rest, Seq),
    matched_prefix(Prefixes, Rest, Ns, Missing),
    Ns = [_, _|_],
    Missing = [Expected|_].

% matched_prefix(+Prefixes, +Seq, -Ns, -Missing):
% Ns are the leading messages of Seq that match
% the leading Prefixes in order; Missing are the
% prefixes left unmatched.
matched_prefix([], _, [], []).
matched_prefix([P|Ps], Seq, Ns, Missing) :-
    (   Seq = [N|Seq1],
        prefix_of(P, N)
    ->  Ns = [N|Ns1],
        matched_prefix(Ps, Seq1, Ns1, Missing)
    ;   Ns = [],
        Missing = [P|Ps]
    ).

?- missing_message(report_request, TID, Ns, E).
TID = 102,
Ns = [19, 20, 21, 22],
E = "Build end" ;
false.

?- thread_sequence(102, Ns).
Ns = [8, 19, 20, 21, 22, 31, 32, 33, 34|...].
```

Worker 102 followed the block for four messages and then, where `Build end: report 18` should be, wrote the first of six table-resize warnings and nothing else. The toplevel abbreviates the thread’s sequence, which is the `max_depth` option of its answer printing, and the elision is itself the finding: the thread went on writing, but not the message that was due. The requirement of at least two matched messages excludes the dispatches to workers 103 and 104 and the health requests, whose next message is not `Query start` because they are different activities, not because a message is missing; without the requirement, every departure from the one block in the knowledge base would be reported as an absence. The deeper limit is the one the catalog states: a message is missing relative to an expectation, and the expectation here is a single block for a single kind of request.

27.10 Ordinals in Messages

Some messages carry an **ordinal**, a number that counts their own kind: the request number in `Request 5 received`, an iteration number, a sequence number of a packet. The ordinals of one kind, read in trace order, should be the natural numbers from some start; a gap is a message that was never written, lost, or written to another trace, and a repetition or a descent is a restart or a reordering. The predicate collects the ordinals of a kind and the second predicate compares them with the full range.

```
% ordinal_message(+Prefix, -Ordinals): Ordinals
% are the request numbers of the messages of kind
% Prefix, in trace order.
ordinal_message(Prefix, Ordinals) :-
    findall(O,
            ( prefix_of(Prefix, N),
              message_key(N, "request"-S),
              number_string(O, S)
            ),
            Ordinals).

% missing_ordinals(+Ordinals, -Missing): the
% numbers between the least and the greatest of
% Ordinals that do not occur in it.
missing_ordinals(Ordinals, Missing) :-
    min_list(Ordinals, Min),
    max_list(Ordinals, Max),
    numlist(Min, Max, All),
    subtract(All, Ordinals, Missing).

?- ordinal_message("Request # received", _Os),
   missing_ordinals(_Os, M),
   format("~w~n", [_Os]).
[1,2,3,4,5,6,7,8,9,10]
M = [].

?- ordinal_message("Request # dispatched", Os),
   missing_ordinals(Os, M).
Os = [1, 2, 3, 4, 5, 6, 7, 8, 9],
M = [].

?- ordinal_message("Request # completed", Os),
   missing_ordinals(Os, M).
Os = [1, 5, 6, 7, 8, 9],
M = [2, 3, 4].
```

Ten requests were received and nine dispatched; request 10 was received on thread 109 and never dispatched, which `missing_ordinals/2` does not see because 10 is beyond the greatest dispatched ordinal, not between two. Six completed, and the gap 2, 3, 4 names the three requests whose workers stalled. `numlist/3` and `subtract/3` do the set difference. The predicate is tied to the `request` key because an ordinal is an identifier of a particular kind, and the knowledge of which number in a message counts and which measures is again the analyst’s; the formalization also assumes ordinals that begin at one and increase, which a trace from the middle of a long run does not satisfy.

27.11 Anchor Messages

Anchor Messages are messages chosen as fixed reference points in a trace, so that other messages can be located relative to them: the arrival of each request, the beginning of each test step, each tick of a periodic task. They give the trace a coordinate system that is coarser than message numbers and more meaningful, and once chosen they divide the trace into stretches that are read one at a

time. The predicate chooses the request arrivals, and `anchored/2` is the stretch from an anchor to the message before the next.

```
% anchor_messages(-Ns): Ns are the anchors of the
% trace, the request arrivals.
anchor_messages(Ns) :-
    findall(N, prefix_of("Request # received", N),
            Ns).

% anchored(+Anchor, -Ns): the messages from Anchor
% up to, not including, the next anchor.
anchored(Anchor, Ns) :-
    anchor_messages(Anchors),
    memberchk(Anchor, Anchors),
    (   aggregate_all(min(A),
                      ( member(A, Anchors),
                        A > Anchor
                      ),
                      Next)
    ->  true
    ;   aggregate_all(max(N), text_of(N, _),
                      Last),
        Next is Last + 1
    ),
    End is Next - 1,
    numlist(Anchor, End, Ns).

?- anchor_messages(_As), format("~w~n", [_As]).
[11,18,23,27,37,39,41,44,46,67]
true.

?- anchored(37, Ns).
Ns = [37, 38].

?- anchored(67, Ns), print_messages(Ns).
 67 109 Request 10 received: GET /session/9/data
 68 109 Receive start: session=71
 69 109 Error: buffer copy failed, code=0x3e6
 70 109 Unhandled exception 0xc0000005, thread 109
 71 100 Service stopping
Ns = [67, 68, 69, 70, 71].
```

The ten anchors cut the trace into ten stretches, and the stretch of request 5 is two messages long: received, queued, and then the next request arrives. The `aggregate_all(max(N), ...)` in the second branch of the if-then-else finds the last message when no anchor follows, so that the final stretch runs to the end of the trace. Anchors are a choice, and the predicate hardwires one; the stretch between anchors is a stretch of the whole trace and not of a thread, so it contains the messages of other threads that happened to be written then, which is sometimes what the analyst wants and sometimes the noise Section 27.12 removes.

27.12 Pivot Message

A **Pivot Message** is a message whose attributes are used to select the rest of the trace: having found a message of interest, the analyst takes its thread, its component, or a datum in it and reads everything that shares it, turning the trace around that message as a table is turned around a pivot. The predicate takes the pivot's number and a dimension and yields the selected messages; the third clause pivots on any key of the message, which connects the pattern to **Linked Messages** (page 570). The dimension is an atom or a `key/1` term, so that the three clauses exclude each other by unification.

```
% pivot_message(+N, ?By, -Ns): Ns are the messages
% that share with message N its thread, its
% component, or a key; By is thread, component,
% or key(Key).
pivot_message(N, thread, Ns) :-
    msg(N, _, _, TID, _, _),
    thread_sequence(TID, Ns).
pivot_message(N, component, Ns) :-
    msg(N, _, _, _, C, _),
    findall(M, msg(M, _, _, _, C, _), Ns).
pivot_message(N, key(Key), Ns) :-
    message_key(N, Key),
    findall(M, message_key(M, Key), Ns).

?- pivot_message(69, thread, Ns).
Ns = [67, 68, 69, 70] ;
false.

?- pivot_message(69, component, _Ns),
   length(_Ns, L).
L = 12 ;
false.

?- pivot_message(26, By, Ns), length(Ns, L).
By = thread,
Ns = [9, 24, 25, 26],
L = 4 ;
By = component,
Ns = [4, 5, 13, 14, 20, 21, 26, 29, 54|...],
L = 12 ;
By = key("session=71"),
Ns = [5, 26, 29, 68],
L = 4 ;
By = key("bytes=1024"),
Ns = [26],
L = 1 ;
false.
```

Pivoting message 69 on its thread gives the four messages of the failing thread, and on its component the twelve messages of `dbclient`, which is the **Trace Partition** (page 489) of Chapter 24 arrived at from one message rather than from the component list. The trailing `false.` after a query with the dimension given is a choice point: SWI-Prolog indexes clauses on their first argument, which is the

same variable in all three, and in this edition (9.0.4) it does not build an index on the second argument of a procedure this small, so the clauses that cannot match are tried and fail. The predicate is correct, and an if-then-else over the dimension would make it deterministic at the cost of the clause-per-dimension form. Pivoting message 26 on its keys gives the session chain of Section 27.5 and a one-element selection for a key nothing else shares. The predicate pivots on one attribute at a time; an analyst pivots on several in succession, narrowing, and the composition of two pivots is the intersection of their selections, which `intersection/3` of Chapter 5 computes and the predicate does not.

27.13 Message Context

The **Message Context** of a message is the messages around it: those immediately before and after in the trace, which show what else the process was doing at that moment, and those before and after on its own thread, which show what the thread itself was doing. The two contexts differ when threads interleave, and reading both is how a message is placed. The predicates take a radius; the thread context is computed on the thread's sequence by position, with `nth0/3` used in both directions, and `once/1` around the first use because a message occurs once in its thread's sequence and the choice point `nth0/3` would leave is known to be empty.

```
% message_context(+N, +K, -Ns): the messages from
% N-K to N+K that exist.
message_context(N, K, Ns) :-
    Lo is N - K,
    Hi is N + K,
    findall(M,
            ( between(Lo, Hi, M),
              msg(M, _, _, _, _, _)
            ),
            Ns).

% thread_context(+N, +K, -Ns): the messages of the
% thread of N within K positions of N on that
% thread.
thread_context(N, K, Ns) :-
    msg(N, _, _, TID, _, _),
    thread_sequence(TID, Seq),
    once(nth0(I, Seq, N)),
    findall(M,
            ( nth0(J, Seq, M),
              abs(J - I) =< K
            ),
            Ns).

?- message_context(53, 1, Ns).
Ns = [52, 53, 54].

?- thread_context(53, 1, Ns).
Ns = [17, 53, 54].
```

```
?- message_context(53, 1, _A),
   thread_context(53, 1, _B),
   append(_A, _B, _C), sort(_C, Ns),
   print_messages(Ns).
 17 101 Request 1 completed: 200, 309 ms
 52 108 Housekeeping: worker 101 idle
 53 101 Request 5 dispatched to worker 101
 54 101 Query start: SELECT report 19
Ns = [17, 52, 53, 54].
```

Message 53 is preceded in the trace by the housekeeping thread's note that worker 101 is idle, and on its own thread by the completion of request 1 eight seconds earlier; the two contexts together say that the worker had been idle since then and was given request 5 as soon as the watchdog had run. `message_context/3` is deliberately defined on numbers rather than on a list, so that it works when numbers have gaps; the thread context is defined by position because a thread's numbers always have gaps. Neither context is defined by time, and a context by time, every message within a second of the pivot, is the **Significant Interval** (page 528) of Chapter 25.

27.14 Indirect Message

An **Indirect Message** is a message written by one thread or component about another: a watchdog reporting a worker's state, a server reporting a client's error, a parent reporting a child's exit. It is often the only evidence about a thread that writes nothing itself, and it must be read with the knowledge that it is secondhand: it says what the writer observed or believed, at the time the writer looked. The predicate finds messages whose `worker` or `thread` key names a thread other than the one that wrote them.

```
% indirect_message(?N, -About): message N was
% written by one thread about the thread About.
indirect_message(N, About) :-
    msg(N, _, _, TID, _, _),
    message_key(N, Word-Number),
    memberchk(Word, ["worker", "thread"]),
    number_string(About, Number),
    About =\= TID.

?- indirect_message(N, About).
N = 49,
About = 103 ;
N = 50,
About = 104 ;
N = 51,
About = 102 ;
N = 52,
About = 101 ;
false.

?- thread_sequence(103, Ns).
Ns = [9, 24, 25, 26].
```

The four housekeeping messages are the trace's only indirect messages, and they are also its only information about workers 103 and 104 after message 30: the workers themselves wrote nothing for seven seconds, and the watchdog's `wait 7001 ms` and `wait 6900 ms` are what the trace knows of the deadlock that `dump1.pl` shows directly. Message 12, `Request 1 dispatched to worker 101`, names a worker but was written by it, and is not indirect. The predicate depends on the convention that the key names the subject, which `thread 109` in message 70 does and a message such as `sent to worker 103` would not; the direction of aboutness is in the grammar of the text, which a key does not have.

27.15 Identification Messages

Identification Messages are the messages that say what was running: the product and its version, the configuration loaded, the environment connected to, the sizes chosen at startup. They are written once, usually first, and are the first thing to check when a trace is compared with another or with a dump, because a difference in them explains differences in everything after. The predicate is a vocabulary test, and the vocabulary is the words that introduce identity in this application's messages.

```
% identity_word(?Word): Word introduces
% identifying information in a message.
identity_word("version").
identity_word("configuration").
identity_word("Connecting to").
identity_word("capacity=").
identity_word("threads=").

% identification_messages(-Ns): the messages
% that contain an identity_word/1.
identification_messages(Ns) :-
    findall(N,
            ( text_of(N, Text),
              once(( identity_word(W),
                     sub_string(Text, _, _, _, W)
                   ))
            ),
            Ns).

?- identification_messages(Ns),
   print_messages(Ns).
  1 100 Service starting, version 3.2.1.0
  2 100 Loading configuration C:\App\app.cfg
  3 100 Cache initialized, capacity=4096
  4 100 Connecting to db01:5432
  6 100 Worker pool started, threads=4
Ns = [1, 2, 3, 4, 6].
```

Five of the first six messages identify the run: the version 3.2.1.0, which is the version `dump1.pl` records for `appserver` and `reportengine` and not for `dbclient`, the configuration file, the cache capacity, the database host and port, and the pool of four workers, which is the number the watchdog later reports as busy. `once/1` around the vocabulary search keeps a message with two

identity words from being collected twice. The vocabulary is the formalization's whole content; a message that identifies the run in words the list lacks, the operating system build, say, is not found, and `Connected to db01:5432, session=71` is correctly not found, since the session is an identifier of the run's data, not of the run.

27.16 Implementation Discourse

The messages of a trace speak at two levels. Some speak the language of the application's domain, of requests, reports, and sessions; others speak the language of its implementation, of threads, tables, buffers, pointers, and hexadecimal codes. **Implementation Discourse** is the second level, and recognizing it matters because it is where the trace gives away how the program is built, and because the two levels must be translated into each other when a domain-level symptom, a request that never completes, is to be explained by an implementation-level event, a table that keeps doubling. The predicate is a vocabulary test with hexadecimal numbers counted as implementation vocabulary.

```
% implementation_word(?Word): Word belongs to the
% implementation level of the messages.
implementation_word("0x").
implementation_word("thread").
implementation_word("table").
implementation_word("buffer").
implementation_word("cpu").
implementation_word("pool").
implementation_word("Cache").

% implementation_discourse(?N): message N uses
% implementation vocabulary.
implementation_discourse(N) :-
    text_of(N, Text),
    once(( implementation_word(W),
           sub_string(Text, _, _, _, W) )).

?- aggregate_all(count,
                 implementation_discourse(_), C).
C = 11.

?- findall(N, implementation_discourse(N), _Ns),
   exclude([N]>>prefix_of("Warning", N), _Ns, Ns),
   print_messages(Ns).
  3 100 Cache initialized, capacity=4096
  6 100 Worker pool started, threads=4
 51 108 Housekeeping: worker 102 cpu 100%
 69 109 Error: buffer copy failed, code=0x3e6
 70 109 Unhandled exception 0xc0000005, thread 109
Ns = [3, 6, 51, 69, 70].
```

Eleven messages, six of them the resize warnings that `exclude/3` removes from the listing, speak of the implementation; the other sixty speak of requests and sessions. The failure at the end is told entirely in implementation discourse, a buffer, a code, an exception, a thread, and the request it belongs to is named only by the domain-level message 67 three lines earlier, which is the translation

the analyst performs. The vocabulary is the limit: `worker`, which is both a domain word and a thread, is on neither list, and the predicate cannot tell that `Query start: SELECT report 17` quotes an implementation, the SQL, inside a domain message.

27.17 Hidden Error

A **Hidden Error** is an error that the trace reports without marking it as one: a message of ordinary severity whose text says that something failed, was refused, or was not available, or a message whose status code is nonzero in a field nobody reads. Searching for `Error` finds the marked errors and misses these, and in many traces the hidden error is the earlier and the more informative one. The predicate finds messages that are not Error Messages but contain failure vocabulary.

```
% failure_word(?Word): Word reports a failure
% without the error marking.
failure_word("failed").
failure_word("exception").
failure_word("No idle").
failure_word("Unhandled").

% hidden_error(?N): message N is not marked as an
% error but reports one.
hidden_error(N) :-
    text_of(N, Text),
    \+ error_message(N),
    once(( failure_word(W),
           sub_string(Text, _, _, _, W) )).

?- findall(N, hidden_error(N), Ns),
   print_messages(Ns).
 38 100 No idle worker, request 5 queued
 40 100 No idle worker, request 6 queued
 42 100 No idle worker, request 7 queued
 70 109 Unhandled exception 0xc0000005, thread 109
Ns = [38, 40, 42, 70].
```

The three `No idle worker` messages are hidden errors in the sense that matters most: they precede the first marked error by two seconds and name its cause, a pool with no free worker, where the marked error names only its symptom, the queue depth. The unhandled exception is the gravest message of the trace and carries no marking at all. `\+ error_message(N)` is safe here because `N` is bound by `text_of/2` before the negation is reached, as Chapter 7 requires. The vocabulary is short and the pattern is open-ended; a hidden error expressed as a status code in a completion message, `completed: 500`, is found only by a rule about codes that this list does not have.

27.18 Silent Messages

Silent Messages record that something happened without recording how it went: a start whose kind has no end, an operation whose outcome the trace statement never included, a message that speaks and says nothing about success or failure. When the activity they begin fails, the trace is silent about

the failure, and the silence is found only by noticing that a kind of message has no counterpart. The predicate looks for invariants that end in `start` and whose stem, the text before `start`, begins no other invariant.

```
% silent_messages(?Inv): Inv is the invariant of a
% start message whose kind has no other message
% kind with the same stem.
silent_messages(Inv) :-
    distinct(Inv, message_invariant(_, Inv)),
    sub_string(Inv, B, _, 0, " start"),
    sub_string(Inv, 0, B, _, Stem),
    \+ ( message_invariant(_, Other),
         Other \== Inv,
         sub_string(Other, 0, _, _, Stem)
       ).

?- silent_messages(Inv).
Inv = "Session # commit start" ;
Inv = "Session # lookup start" ;
false.

?- print_messages([25, 30]).
 25 103 Session 9 commit start
 30 104 Session 9 lookup start
true.
```

`Query start` and `Build start` have their `Query end` and `Build end`, and `Send start` has, by a different name, its reply; the two session operations on workers 103 and 104 have nothing. Whether the code has no trace statement at the end of a commit or a lookup, or has one that was never reached, the trace cannot say, and that is the pattern: these two messages are the last words of the two threads that deadlock, and they are silent about it. The predicate's reading of silence is narrow, a start without an end kind, and misses the other form, an end message that omits its status; it also relies on the stem convention, which `Send start` and `Reply received` break and which it therefore reports correctly only because the stem `Send` is not tested when the kind is not silent.

27.19 Exception Stack Trace

An **Exception Stack Trace** in a software trace is a stack trace written out as messages: a header followed by one message per frame, in the format the component's exception handler chose. It is the point where trace and log analysis meets memory dump analysis, since the frames are the same frames a debugger would show, and it is recognized as a run of consecutive frame-like messages on one thread. `trace1.pl` contains no such run, so this section adds one: the report the exception filter of `appserver` writes after message 71, as eight further `msg/6` facts.

The facts go into the same predicate as the trace, which requires a word about files. SWI-Prolog associates the clauses of a predicate with the file they were loaded from, and consulting a second file that defines clauses for `msg/6` would replace the procedure, with a warning, rather than extend it. The `multifile/1` directive declares that the clauses of a predicate may come from several files,

and with it the excerpt extends `trace1.pl` instead of overwriting it. The declaration must precede the clauses in the extending file.

```
:- multifile msg/6.

msg(72, 14.011, 4120, 109, appserver,
    "Stack trace for thread 109:").
msg(73, 14.011, 4120, 109, appserver,
    "  00 dbclient!Buffer::Copy+0x1a").
msg(74, 14.012, 4120, 109, appserver,
    "  01 dbclient!Connection::Receive+0x8c").
msg(75, 14.012, 4120, 109, appserver,
    "  02 appserver!Session::Fetch+0x44").
msg(76, 14.013, 4120, 109, appserver,
    "  03 appserver!Worker::Run+0x120").
msg(77, 14.013, 4120, 109, appserver,
    "  04 ucrtbase!thread_start<...>+0x42").
msg(78, 14.014, 4120, 109, appserver,
    "  05 kernel32!BaseThreadInitThunk+0x14").
msg(79, 14.014, 4120, 109, appserver,
    "  06 ntdll!RtlUserThreadStart+0x21").
```

A frame line is parsed by a grammar from `library(dcg/basics)`, as Chapter 12 parsed debugger output: blanks, the frame number, blanks, the module up to the `!`, the function up to the `+`, and the hexadecimal offset after `0x`, with `eos//0` requiring that nothing follows. The result has the shape of the `frame/5` facts of `dump1.pl` without the thread, which the message supplies.

```
:- use_module(library(dcg/basics)).

% frame_line(-Frame)//: parse a stack frame line
% into frame(N, Module, Function, Offset).
frame_line(frame(N, Module, Function, Offset)) -->
    blanks, integer(N), blank, blanks,
    string_without("!", MCodes), "!",
    string_without("+", FCodes), "+0x",
    xinteger(Offset), eos,
    { atom_codes(Module, MCodes),
      atom_codes(Function, FCodes) }.

% frame_message(?N, -Frame): message N is a frame
% line.
frame_message(N, Frame) :-
    text_of(N, Text),
    string_codes(Text, Codes),
    phrase(frame_line(Frame), Codes).
```

```
% exception_stack_trace(?TID, -Ns, -Frames): the
% frame messages Ns of thread TID parse to Frames.
exception_stack_trace(TID, Ns, Frames) :-
    trace_thread(TID),
    findall(N,
            ( msg(N, _, _, TID, _, _),
              frame_message(N, _)
            ),
            Ns),
    Ns = [_|_],
    maplist(frame_message, Ns, Frames).

% print_frames(+Frames): one line per frame, as a
% debugger would print it.
print_frames(Frames) :-
    forall(member(frame(N, M, F, O), Frames),
           format("~t~d~2| ~a!~a+0x~16r~n",
                  [N, M, F, O])).

?- frame_message(73, F).
F = frame(0, dbclient, 'Buffer::Copy', 26).

?- frame_message(72, F).
false.

?- exception_stack_trace(TID, Ns, _Frames),
   print_frames(_Frames).
 0 dbclient!Buffer::Copy+0x1a
 1 dbclient!Connection::Receive+0x8c
 2 appserver!Session::Fetch+0x44
 3 appserver!Worker::Run+0x120
 4 ucrtbase!thread_start<...>+0x42
 5 kernel32!BaseThreadInitThunk+0x14
 6 ntdll!RtlUserThreadStart+0x21
TID = 109,
Ns = [73, 74, 75, 76, 77, 78, 79] ;
false.
```

The seven frames are those of thread 109 in `dump1.pl`, and Chapter 28 compares the two mechanically; the offsets come back as integers, 26 for `0x1a`, as the toplevel prints every integer in decimal, and `print_frames/1` restores the hexadecimal with `~16r`. `string_without//2` and `xinteger//1` carry the grammar; the header line fails to parse, as it should. The predicate collects every frame-like message of the thread, which is right for one stack trace and wrong for two: a thread that reports two exceptions would have its frames merged, and a run-based definition with `append/3`, as in Section 27.8, is the repair. The grammar is also the format of one exception handler, and a managed-code stack trace, or one printed innermost frame last, needs its own.

27.20 Message Change

A **Message Change** is a difference in the text of a message between two traces of the same software: a word changed, a field added, a format reordered, usually because the source changed between builds. It defeats searches that were written for the old text and breaks comparisons of the two

traces kind by kind, and it is recognized by comparing the sets of invariants of the two traces and pairing each invariant that exists in one only with the most similar invariant that exists in the other only.

A second trace is needed, and it is given in the `msg/7` form of Chapter 16, with the trace identifier first. The trace `t2` is the first second and a half of the same service built as version 3.3.0.0 and run under a test harness, which also wrote messages of its own; it serves this section and Sections 27.21 to 27.24. The predicate `trace_text/5` is a view over both forms, so that one definition of "the invariants of a trace" covers `trace1.pl` and the in-chapter traces.

```
:- multifile msg/7.

msg(t2, 1, 0.000, 4120, 100, tracer,
    "Trace session started, buffer 64 MB").
msg(t2, 2, 0.001, 4120, 100, appserver,
    "Service starting, version 3.3.0.0").
msg(t2, 3, 0.005, 4120, 100, appserver,
    "Loading configuration C:\\App\\app.cfg").
msg(t2, 4, 0.006, 4120, 100, appserver,
    "Error: key proxy missing, default used").
msg(t2, 5, 0.012, 4120, 100, cachelib,
    "Cache initialized, capacity=4096").
msg(t2, 6, 0.016, 4120, 100, dbclient,
    "Connecting to db01:5432").
msg(t2, 7, 0.230, 4120, 100, dbclient,
    "Connected to db01:5432, session=72").
msg(t2, 8, 0.231, 4120, 100, appserver,
    "Worker pool started, threads=4").
msg(t2, 9, 0.900, 4120, 100, tester,
    "MARK: step 1 begin").
msg(t2, 10, 1.002, 4120, 100, appserver,
    "Request 1 accepted: GET /report/17").
msg(t2, 11, 1.003, 4120, 101, appserver,
    "Request 1 dispatched to worker 101").
msg(t2, 12, 1.300, 4120, 101, appserver,
    "Request 1 completed: status=200, 297 ms").
msg(t2, 13, 1.400, 4120, 100, tester,
    "MARK: step 1 end").
msg(t2, 14, 1.500, 4120, 100, appserver,
    "Service stoppinged to worker 101").
msg(t2, 15, 1.501, 4120, 100, tracer,
    "Trace session stopped, events lost: 0").

% trace_text(?Trace, ?N, ?TID, ?Component, ?Text):
% a view over trace1 (msg/6) and the in-chapter
% traces (msg/7).
trace_text(trace1, N, TID, C, Text) :-
    msg(N, _, _, TID, C, Text).
trace_text(T, N, TID, C, Text) :-
    msg(T, N, _, _, TID, C, Text).
```

```
% trace_invariant(+Trace, ?Inv): Inv is an
% invariant of Trace; each once.
trace_invariant(T, Inv) :-
    distinct(Inv, ( trace_text(T, _, _, _, Text),
                    invariant(Text, Inv) )).

% similarity(+S1, +S2, -Sim): the share of words
% the two strings have in common.
similarity(S1, S2, Sim) :-
    split_string(S1, " ", ",:", W1),
    split_string(S2, " ", ",:", W2),
    intersection(W1, W2, Shared),
    length(Shared, N),
    length(W1, L1),
    length(W2, L2),
    Sim is N / max(L1, L2).

% message_change(+Old, +New, -OldInv, -NewInv):
% NewInv occurs in New only, and OldInv is the
% invariant of Old only that is most similar to
% it, at least half its words.
message_change(Old, New, OldInv, NewInv) :-
    trace_invariant(New, NewInv),
    \+ trace_invariant(Old, NewInv),
    aggregate_all(max(S, I),
                  ( trace_invariant(Old, I),
                    \+ trace_invariant(New, I),
                    similarity(NewInv, I, S)
                  ),
                  max(S, OldInv)),
    S >= 0.5.

?- message_change(trace1, t2, Old, New).
Old = "Request # received: GET /report/#",
New = "Request # accepted: GET /report/#" ;
Old = "Request # completed: #, # ms",
New = "Request # completed: status=#, # ms" ;
false.
```

Two messages changed between the builds: `received` became `accepted`, and the completion message gained a field name. Every predicate of this chapter that tests for the prefix `"Request # received"` would find nothing in `t2`, and that is the pattern's practical content. `intersection/3` counts the shared words, and the witness form of `max/2` returns the invariant with the greatest share. The threshold of one half is arbitrary, and the measure is crude: word order is ignored, and a message rewritten entirely is reported as new rather than as changed, which is also what the tracer's and tester's messages are, correctly. The version message itself did not change, since the version number is blanked by the invariant; that the version differs is read from the **Identification Messages** (page 582), not from this pattern.

27.21 False Positive Error

A **False Positive Error** is a message marked as an error that does not indicate a problem: an expected failure handled by a fallback, a probe that is supposed to fail, an error logged at the wrong severity.

It is the complement of **Hidden Error** (page 584), and it is as common; an analyst who treats every `Error` as a lead wastes the time it would take to learn which errors are routine. The knowledge of which are routine cannot be computed from the trace, and the predicate states it as facts over invariants and applies them to any trace through the `trace_text/5` view.

```
% benign_error(?Inv): messages with the invariant
% Inv are marked as errors but are routine.
benign_error(
    "Error: key proxy missing, default used").

% false_positive_error(+Trace, ?N): message N of
% Trace is marked as an error and is benign.
false_positive_error(T, N) :-
    trace_text(T, N, _, _, Text),
    sub_string(Text, 0, _, _, "Error:"),
    invariant(Text, Inv),
    benign_error(Inv).

?- false_positive_error(trace1, N).
false.

?- false_positive_error(t2, N),
   trace_text(t2, N, _, _, Text).
N = 4,
Text = "Error: key proxy missing, default used" ;
false.
```

The missing configuration key in `t2` is logged as an error and resolved in the same message, and the knowledge base says so; none of the four marked errors of `trace1.pl` is in it. This is the honest shape of the pattern: the predicate recognizes nothing on its own, it applies what the analyst or the developers have recorded, and its value grows with the knowledge base. The limit is the inverse of its strength: a benign error becomes a real one in a context the invariant does not capture, a missing key that this time had no default, and the fact would then be wrong.

27.22 Diegetic Messages

The term is borrowed from narratology, where the diegesis is the world of the story and the diegetic is what belongs to it. **Diegetic Messages** are the messages written by the software under observation about its own activity; non-diegetic messages are written about the trace rather than from within it: the tracing tool's session headers and buffer statistics, a test harness's markers, an operator's notes. Both kinds are in the file, and the analysis of the software must set the second aside while the analysis of the trace itself, whether events were lost, when the session began, relies on it. The predicate classifies by component, which is where the distinction usually shows.

```
% non_diegetic_component(?C): component C writes
% about the trace, not from within the software.
non_diegetic_component(tracer).
non_diegetic_component(tester).
```

```
% diegetic_messages(+Trace, -Ns): the messages of
% Trace written by the software itself.
diegetic_messages(T, Ns) :-
    findall(N,
            ( trace_text(T, N, _, C, _),
              \+ non_diegetic_component(C)
            ),
            Ns).

?- diegetic_messages(t2, _Ns), length(_Ns, L),
   format("~w~n", [_Ns]).
[2,3,4,5,6,7,8,10,11,12,14]
L = 11.

?- forall(( trace_text(t2, N, _, C, Text),
            non_diegetic_component(C) ),
          format("~t~d~3| ~a ~s~n",
                 [N, C, Text])).
  1 tracer Trace session started, buffer 64 MB
  9 tester MARK: step 1 begin
 13 tester MARK: step 1 end
 15 tracer Trace session stopped, events lost: 0
true.

?- diegetic_messages(trace1, _Ns), length(_Ns, L).
L = 79.
```

Eleven of the fifteen messages of `t2` are diegetic; the four others frame them. The last message of the tracer, `events lost: 0`, is non-diegetic and indispensable: it says that the diegetic messages are all of them, which no diegetic message could say. `trace1.pl` has no non-diegetic component, and all 79 of its messages, the excerpt of Section 27.19 included, are diegetic. The component is a proxy for the distinction, and a proxy fails in both directions: a software component that logs its own tracing overhead writes non-diegetic messages under a diegetic name, and the test harness's marks, which Section 27.23 uses, are written in the software's process and might have been written by it.

27.23 Marked Message

A **Marked Message** is a message written into the trace on purpose as a landmark rather than as a record of the software's behavior: a tester's note at the start of each step of a reproduction, a marker injected by a tool when the operator pressed a key, a line the analyst asked to have added. Marks are anchors chosen before the trace exists rather than after, and the stretch between two marks is the trace of one step. The predicate recognizes the mark format of the test harness and the second predicate yields the messages between a step's beginning and end.

```
% marked_message(+Trace, ?N, -Label): message N of
% Trace is a mark with the given Label.
marked_message(T, N, Label) :-
    trace_text(T, N, _, _, Text),
    string_concat("MARK: ", Label, Text).
```

```
% between_marks(+Trace, +Step, -Ns): the messages
% between the begin and end marks of Step.
between_marks(T, Step, Ns) :-
    marked_message(T, N1, L1),
    string_concat(Step, " begin", L1),
    marked_message(T, N2, L2),
    string_concat(Step, " end", L2),
    findall(N,
            ( trace_text(T, N, _, _, _),
              N1 < N, N < N2
            ),
            Ns).

?- marked_message(t2, N, Label).
N = 9,
Label = "step 1 begin" ;
N = 13,
Label = "step 1 end" ;
false.

?- between_marks(t2, "step 1", Ns).
Ns = [10, 11, 12] ;
false.

?- marked_message(trace1, N, Label).
false.
```

Step 1 of the reproduction is three messages long, the complete lifecycle of request 1 in the new build, and nothing else; the marks have done their work by excluding the startup and the shutdown. `string_concat/3` with the first argument bound and the third bound extracts the label deterministically. `trace1.pl` has no marks, which is the ordinary case of a trace taken in production, and the pattern is then applied in the other direction: the analyst decides where marks would have been and uses **Anchor Messages** (page 577) instead. The format of a mark is a convention of one harness, and the predicate knows that one.

27.24 Palimpsest Messages

A palimpsest is a page that was written on twice, with the earlier writing showing through the later. **Palimpsest Messages** are messages in which the text of one message has been written over the text of another, usually because two writers shared a buffer or because a fixed-size record was reused without being cleared: the beginning of the message is the later writer’s and the end is the earlier one’s. Such a message matches no known invariant, and the recognizer looks for one whose invariant begins with a known invariant and ends with the tail of another.

```
% palimpsest_messages(+Trace, ?N, -Over, -Under):
% the invariant of message N of Trace is the known
% invariant Over followed by a tail of the known
% invariant Under.
palimpsest_messages(T, N, Over, Under) :-
    trace_text(T, N, _, _, Text),
    invariant(Text, Inv),
    \+ trace_invariant(trace1, Inv),
```

```
    trace_invariant(trace1, Over),
    string_concat(Over, Tail, Inv),
    Tail \== "",
    trace_invariant(trace1, Under),
    Under \== Over,
    string_concat(_, Tail, Under).

?- palimpsest_messages(t2, N, Over, Under).
N = 14,
Over = "Service stopping",
Under = "Request # dispatched to worker #" ;
false.

?- trace_text(t2, 14, _, _, Text).
Text = "Service stoppinged to worker 101".
```

Message 14 of `t2` reads `Service stoppinged to worker 101`: the main thread's `Service stopping` was written over a worker's `Request 2 dispatched to worker 101` from its first character, and the longer message's tail shows through. The predicate recovers both layers, the one on top exactly and the one below by kind, and the worker number survives in the tail. The known invariants are those of `trace1.pl`, which stands in for the vocabulary of the software; a palimpsest of two messages that are both unknown is not recognized, nor is one where the later writer started in the middle of the record, and a message that genuinely begins with another message's text, `Service stopping requested by operator`, would be reported falsely if the vocabulary happened to contain a message ending in `requested by operator`.

27.25 Translated Message

A **Translated Message** is a message whose text is in a language other than the one the analyst's searches assume: a log from a localized installation, where the strings were translated by the vendor and the numbers were not. Every search for `Error:` fails against `Fehler:`, and every invariant of this chapter is a different string, while the structure of the message, which fields it has and in what order, is unchanged. The recognizer needs a dictionary of invariants, and with it a translated message is mapped back to the text its untranslated counterpart would have, with the numbers put back in place.

```
:- multifile msg/7.

msg(de, 1, 0.000, 4120, 100, appserver,
    "Dienst startet, Version 3.2.1.0").
msg(de, 2, 1.002, 4120, 100, appserver,
    "Anforderung 1 empfangen: GET /report/17").
msg(de, 3, 1.311, 4120, 101, appserver,
    "Anforderung 1 abgeschlossen: 200, 309 ms").
msg(de, 4, 14.002, 4120, 109, dbclient,
    "Fehler: Kopie fehlgeschlagen, Code=0x3e6").
msg(de, 5, 14.010, 4120, 100, appserver,
    "Dienst wird beendet").

% translation(?Foreign, ?English): the invariant
% Foreign is the localized form of English.
```

```
translation(
    "Dienst startet, Version #",
    "Service starting, version #").
translation(
    "Anforderung # empfangen: GET /report/#",
    "Request # received: GET /report/#").
translation(
    "Anforderung # abgeschlossen: #, # ms",
    "Request # completed: #, # ms").
translation(
    "Fehler: Kopie fehlgeschlagen, Code=#",
    "Error: buffer copy failed, code=#").
translation(
    "Dienst wird beendet",
    "Service stopping").

% digit_tokens(+Text, -Tokens): the runs of Text
% that invariant/2 blanks, as strings, in order.
digit_tokens(Text, Tokens) :-
    string_codes(Text, Codes),
    digit_runs(Codes, Tokens).

digit_runs([], []).
digit_runs([C|Cs], Tokens) :-
    (   code_type(C, digit)
    ->  take_run([C|Cs], Run, Rest),
        string_codes(Token, Run),
        Tokens = [Token|Tokens1],
        digit_runs(Rest, Tokens1)
    ;   digit_runs(Cs, Tokens)
    ).

take_run(Cs, Run, Rest) :-
    (   Cs = [C|Cs1],
        (   code_type(C, alnum)
        ;   C == 0'.
        )
    ->  Run = [C|Run1],
        take_run(Cs1, Run1, Rest)
    ;   Run = [],
        Rest = Cs
    ).

% fill(+Inv, +Tokens, -Text): Text is Inv with
% each # replaced by the next of Tokens.
fill(Inv, Tokens, Text) :-
    string_codes(Inv, Codes),
    fill_codes(Codes, Tokens, Out),
    string_codes(Text, Out).
```

```
fill_codes([], [], []).
fill_codes([C|Cs], Tokens, Out) :-
    (   C == 0'#
    ->  Tokens = [Token|Tokens1],
        string_codes(Token, TCodes),
        append(TCodes, Out1, Out),
        fill_codes(Cs, Tokens1, Out1)
    ;   Out = [C|Out1],
        fill_codes(Cs, Tokens, Out1)
    ).

% translated_message(+Trace, ?N, -English):
% message N of Trace, translated back by its
% invariant.
translated_message(T, N, English) :-
    trace_text(T, N, _, _, Text),
    invariant(Text, Foreign),
    translation(Foreign, EnglishInv),
    digit_tokens(Text, Tokens),
    fill(EnglishInv, Tokens, English).

?- translated_message(de, 4, E).
E = "Error: buffer copy failed, code=0x3e6".

?- forall(translated_message(de, N, E),
          format("~t~d~3| ~s~n", [N, E])).
  1 Service starting, version 3.2.1.0
  2 Request 1 received: GET /report/17
  3 Request 1 completed: 200, 309 ms
  4 Error: buffer copy failed, code=0x3e6
  5 Service stopping
true.

?- translated_message(de, 4, _E),
   invariant(_E, Inv),
   message_invariant(N, Inv).
Inv = "Error: buffer copy failed, code=#",
N = 69 ;
false.
```

The five German messages come back as the five English ones they correspond to, and the last query closes the loop: the translation of message 4 has the invariant of message 69 of `trace1.pl`, so every predicate of this chapter can be applied to the localized log once it has been translated by kind. `digit_tokens/2` is `invariant/2` keeping what it blanks, and `fill/3` is the inverse of blanking; the two must agree on what a number is, which is why they share the definition of a run. The dictionary is the limit, as it is for any translation: a message whose invariant is not in it is not translated, and a localization that reorders the fields, as grammar sometimes requires, would put the numbers back in the wrong places.

27.26 Recovered Messages

Recovered Messages are messages that did not reach the trace file and were recovered from somewhere else: the tracing buffers found in a memory dump, a corrupted file read around its

damage, a circular log whose oldest records were partly overwritten. They arrive incomplete, often without a number, a time, or a thread, and sometimes without their beginning, and the first task is to say what kind of message each one is, which the invariants of the trace make possible even for a fragment. The excerpt here is three records recovered from the trace buffer of `appserver` in `dump1.pl`, represented as facts of their own, since they are not messages of the trace.

```
% recovered(?Id, ?Text): a record recovered from
% the tracing buffer in the memory dump.
recovered(1, "ispatched to worker 102").
recovered(2, "Query start: SELECT report 18").
recovered(3,
    "Warning: table resize, 524288 -> 1048576").

% recovered_messages(?Id, -Inv): the recovered
% record Id ends like the trace invariant Inv.
recovered_messages(Id, Inv) :-
    recovered(Id, Text),
    invariant(Text, Fragment),
    distinct(Inv, message_invariant(_, Inv)),
    sub_string(Inv, _, _, 0, Fragment).

?- recovered_messages(Id, Inv).
Id = 1,
Inv = "Request # dispatched to worker #" ;
Id = 2,
Inv = "Query start: SELECT report #" ;
Id = 3,
Inv = "Warning: table resize, # -> #" ;
false.

?- recovered(3, Text), numbers_in(Text, Ns).
Text = "Warning: table resize, 524288 -> 1048576",
Ns = [524288, 1048576].
```

The first record lost its beginning and is still identified as a dispatch; the second is a complete query start; the third is a resize warning whose numbers continue the series of Section 27.4 one step past message 36, which places it after the last message of worker 102 in the file and says that the table doubled at least once more before the dump was taken. The suffix test with `sub_string/5` and the length after the match bound to 0 is what tolerates the missing beginning. What the recovered records lack, the predicate cannot restore: the time and thread of record 3 are inferred from the series, not read, and the inference should be written down as an inference, which is the Message Annotation of the next section.

27.27 Message Annotations

Message Annotations are notes the analyst attaches to messages: what it means, what it corresponds to in another artifact, why it was selected, what was inferred from it. Annotations are the analyst’s layer over the trace, and they must be kept apart from the messages, so that a reader can tell what the software said from what the analyst concluded. The dynamic database of Chapter

11 is the natural home: `message_annotation/2` is declared dynamic, `annotate/2` adds a note with `assertz/1`, and a printer shows each message with its notes beneath it.

```
:- dynamic message_annotation/2.

% annotate(+N, +Note): attach Note to message N.
annotate(N, Note) :-
    assertz(message_annotation(N, Note)).

% print_annotated(+Ns): print each message with
% its annotations indented below it.
print_annotated(Ns) :-
    forall(member(N, Ns),
           ( print_msg(N),
             forall(message_annotation(N, Note),
                    format("~t~8|> ~s~n", [Note]))
           )).

?- annotate(43, "first symptom; cause is 38"),
   annotate(70, "matches exception_record/3"),
   annotate(70, "0xc0000005: access violation").
true.

?- print_annotated([43, 70]).
 43 100 Error: queue depth 3 exceeds threshold 2
        > first symptom; cause is 38
 70 109 Unhandled exception 0xc0000005, thread 109
        > matches exception_record/3
        > 0xc0000005: access violation
true.

?- aggregate_all(count, message_annotation(_, _),
                 C).
C = 3.
```

The notes are facts and can be queried like any other: every annotated message, every message annotated with a word, every message whose annotation names another message. The `dynamic/1` declaration is what allows `assertz/1` on a predicate that has no clauses in any file; without it the first `annotate/2` would create the predicate, in SWI-Prolog, but a query before it would raise an existence error instead of failing. The annotations live in the running toplevel and are lost with it, and an analysis that is to be kept writes them to a file, with `portray_clause/1` as Chapter 13 did, or uses `library(persistency)`. The pattern's limit is not in the mechanism: an annotation is only as good as the inference it records, and the predicate stores both the good and the bad with the same confidence.

27.28 Message Cover

A **Message Cover** is a family of message sets, each defined by a property, whose union is the whole trace or the whole of a region: a way of asking whether every message is accounted for by some kind the analyst understands. Unlike the **Trace Partition** (page 489) of Chapter 24 the sets may overlap, and the interesting output is not the cover but its complement, the messages no set claims,

which are the ones the analyst has not yet understood. The sets are named clauses of `message_set/2`, most of them built from this chapter's predicates, and `message_cover/2` yields the uncovered remainder.

```
% message_set(?Name, ?N): message N belongs to the
% named set.
message_set(lifecycle, N) :-
    prefix_of("Service", N).
message_set(identity, N) :-
    identification_messages(Ns),
    member(N, Ns).
message_set(workers, N) :-
    prefix_of("Worker", N).
message_set(request, N) :-
    message_key(N, "request"-_).
message_set(session, N) :-
    message_key(N, Key),
    key_word(Key, "session").
message_set(query, N) :-
    prefix_of("Query", N).
message_set(build, N) :-
    prefix_of("Build", N).
message_set(resize, N) :-
    prefix_of("Warning: table resize", N).
message_set(housekeeping, N) :-
    prefix_of("Housekeeping", N).
message_set(error, N) :-
    error_message(N).
message_set(error, N) :-
    hidden_error(N).
message_set(frames, N) :-
    frame_message(N, _).

% message_cover(+Names, -Uncovered): Uncovered are
% the messages that belong to none of the named
% sets.
message_cover(Names, Uncovered) :-
    findall(N,
            ( msg(N, _, _, _, _, _),
              \+ ( member(S, Names),
                   message_set(S, N) )
            ),
            Uncovered).

?- message_cover([request, error], _U),
   length(_U, L).
L = 46.

?- message_cover([lifecycle, identity, workers,
      request, session, resize, housekeeping,
      error], _U), length(_U, L).
L = 19.
```

```
?- message_cover([lifecycle, identity, workers,
      request, session, query, build, resize,
      housekeeping, error], U).
U = [72, 73, 74, 75, 76, 77, 78, 79].

?- message_cover([lifecycle, identity, workers,
      request, session, query, build, resize,
      housekeeping, error, frames], U).
U = [72].
```

Requests and errors alone leave 46 of the 79 messages unclaimed; eight sets leave nineteen, the query and build messages, which belong to requests but do not name them, and the stack trace excerpt; ten sets leave the excerpt; eleven sets leave one message, the header of the stack trace, which is neither a frame nor anything else the sets know. The sequence of queries is the analyst's work made visible: each uncovered remainder names a kind to be understood, and the cover is complete when the analyst can say of every message which kind it is, which here needs a twelfth set for a kind with one member. The predicate tests membership by calling the set's definition for each message, and the sets are as good as their definitions; a cover is a statement about the sets, not about the software, and a complete cover of a trace whose kinds were all defined by prefix says only that every message begins in a familiar way.

27.29 Defamiliarizing Effect

The last pattern of the chapter is a reading technique rather than a property of messages, and it has no predicate. The **Defamiliarizing Effect** is the name, borrowed from literary theory, for what happens when a trace that has become too familiar to see is presented in an unfamiliar form, so that the reader notices what routine had hidden: the same messages sorted by text rather than by time, the invariants without their numbers, a thread's messages alone on the page, the trace read backward from its last message, or the numbers of the messages hidden so that the eye cannot skip to the one it already knows. Several of this chapter's predicates produce such presentations as a by-product. The invariant table of Section 27.3 shows the trace as 34 kinds and makes the single occurrences visible; `thread_sequence/2` shows a thread alone; the translated log of Section 27.25 shows familiar messages in another language and forces every one to be read.

The technique is a corrective for the analyst's own habits, and that is why it is not formalized: what is familiar is a fact about the reader, not about the trace, and the presentation that defamiliarizes one reader bores another. The predicates of this chapter are nonetheless the means by which the presentations are made, and the toplevel, which answers each query in a form chosen by the query rather than by a viewer's layout, is itself a defamiliarizing instrument when it is used that way.

Chapter 28. Sequences, Motifs, and Correlation

The patterns of Chapter 27 looked at messages one at a time and at pairs linked by their text. This chapter looks at messages in sequence: the order they must have had, the phases that order falls into, the sub-sequences that recur, the sequences that two traces share, and the sequences that no trace could have. It then looks across artifacts, merging two traces into one and, in the last section, reading a trace and a memory dump together. The constructs are those the earlier chapters built up to, and two of them return here in force: the grammars of Chapter 12, which parse a sequence of message kinds as they parsed a sequence of characters, and the constraints of Chapter 14, which reconstruct an order from partial information and decide whether an order is possible at all.

The chapter works with the **kind** of a message rather than its text. A kind is a short atom, `received`, `dispatched`, `query_start`, derived from the message invariant of Chapter 27 by matching a prefix; reducing the 71 messages to a sequence of eleven kinds is what makes sequences short enough to compare and count. The reduction is a loss, and the closing remarks return to it. The helpers below define the kinds as a table of prefixes, `kind_prefix/2`, and one rule, `kind_of/2`, that classifies an invariant by the first prefix it begins with and assigns `other` to the rest; consult the trace; and provide the message printer and the run-length counter of Chapter 9 that several sections reuse. The flag that hides anonymous bindings is set for the chapter.

```
:- [trace1].

% text_of(?N, ?Text): the text of message N.
text_of(N, Text) :-
    msg(N, _, _, _, _, Text).

% invariant(+Text, -Inv): Text with every number
% replaced by #, as in Chapter 27.
invariant(Text, Inv) :-
    string_codes(Text, Codes),
    blank_numbers(Codes, Skeleton),
    string_codes(Inv, Skeleton).

blank_numbers([], []).
blank_numbers([C|Cs], Skeleton) :-
    (   code_type(C, digit)
    ->  skip_number(Cs, Rest),
        Skeleton = [0'#|Skeleton1],
        blank_numbers(Rest, Skeleton1)
    ;   Skeleton = [C|Skeleton1],
        blank_numbers(Cs, Skeleton1)
    ).
```

```
skip_number(Cs, Rest) :-
    (   Cs = [C|Cs1],
        (   code_type(C, alnum)
        ;   C == 0'.
        )
    ->  skip_number(Cs1, Rest)
    ;   Rest = Cs
    ).

% kind(?N, ?Kind): the kind atom of message N.
kind(N, Kind) :-
    text_of(N, Text),
    invariant(Text, Inv),
    kind_of(Inv, Kind).

% kind_prefix(?Kind, ?Prefix): a message whose
% invariant begins with Prefix is of kind Kind.
kind_prefix(received, "Request # received").
kind_prefix(dispatched, "Request # dispatched").
kind_prefix(completed, "Request # completed").
kind_prefix(query_start, "Query start").
kind_prefix(query_end, "Query end").
kind_prefix(build_start, "Build start").
kind_prefix(build_end, "Build end").
kind_prefix(resize, "Warning: table").
kind_prefix(queued, "No idle").
kind_prefix(error, "Error:").

% kind_of(+Inv, ?Kind): Kind is the kind whose
% prefix Inv begins with, or other if none does.
% The kind is found in a fresh variable and
% unified with Kind only after the commitment.
kind_of(Inv, Kind) :-
    (   kind_prefix(Kind0, Prefix),
        sub_string(Inv, 0, _, _, Prefix)
    ->  Kind = Kind0
    ;   Kind = other
    ).

% trace_thread(-TID): every thread, once each.
trace_thread(TID) :-
    distinct(TID, msg(_, _, _, TID, _, _)).

% thread_sequence(+TID, -Ns): message numbers of
% thread TID in order.
thread_sequence(TID, Ns) :-
    findall(N, msg(N, _, _, TID, _, _), Ns0),
    sort(Ns0, Ns).

% thread_kinds(+TID, -Kinds): the kinds of thread
% TID's messages in order.
thread_kinds(TID, Kinds) :-
    thread_sequence(TID, Ns),
    maplist(kind, Ns, Kinds).

% print_messages(+Ns): one line per message.
print_messages(Ns) :-
    forall(member(N, Ns), print_msg(N)).
```

```
print_msg(N) :-
    msg(N, _, _, TID, _, Text),
    format("~t~d~3| ~d ~s~n", [N, TID, Text]).

% print_kinds(+Ks): the kinds of a sequence, one
% per line with its position, so a long sequence
% stays within the page.
print_kinds(Ks) :-
    forall(nth1(I, Ks, K),
           format("~t~d~3| ~a~n", [I, K])).

% run_counts(+Sorted, -Counts): X-N per run of N
% equal elements X, as in Chapter 9.
run_counts([], []).
run_counts([X|Xs], [X-N|Counts]) :-
    run_length(Xs, X, 1, N, Rest),
    run_counts(Rest, Counts).

run_length(Xs, X, N0, N, Rest) :-
    (   Xs = [Y|Xs1],
        Y == X
    ->  N1 is N0 + 1,
        run_length(Xs1, X, N1, N, Rest)
    ;   N = N0,
        Rest = Xs
    ).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- thread_kinds(101, _Kinds), length(_Kinds, N).
N = 21.

?- kind(69, K).
K = error.
```

`kind/2` is called in two modes in this chapter: with the kind unbound, to enumerate, as `thread_kinds/2` does, and with the kind bound, to test membership, as `phase_boundaries/1` in Section 28.2 and `first_of_kind/3` in Section 28.12 do. The two must agree: `kind(N, K)` with `K` bound must succeed exactly when `kind(N, K0)` with `K0` unbound would bind `K0` to `K`. `kind_of/2` guarantees this by classifying into a fresh variable, `Kind0`, committing with if-then-else to the first prefix that matches, and unifying the result with the caller's argument only afterwards. A version written as one clause per kind, each ending in a cut, with a catch-all `kind_of(_, other)` last, would be correct in the first mode and wrong in the second: with `other` supplied, the heads of the specific clauses do not unify, their cuts are never reached, and the catch-all accepts every message, so that `kind(11, other)` succeeds although message 11 is a `received`. That is the non-steadfast shape of Chapter 7, output unification in the head before the cut. The check below exercises the bound mode, including the case that distinguishes the two versions, and then both modes over every message:

```
?- kind(11, K).
K = received.
```

```
?- kind(11, received).
true.

?- kind(11, other).
false.

?- aggregate_all(count, kind(_, other), C).
C = 22.

?- aggregate_all(count,
   ( kind(_, K), K == other ), C).
C = 22.

?- setof(K, N^kind(N, K), _Ks),
   forall(msg(N, _, _, _, _, _),
          ( aggregate_all(count, kind(N, _), 1),
            aggregate_all(count,
                ( member(K, _Ks), kind(N, K) ),
                1) )).
true.
```

Twenty-two messages are of kind `other` whether the kind is supplied or generated and then compared, and the last query says that every message has exactly one kind when enumerated and is accepted by exactly one of the eleven kinds when each is supplied; the version with cuts would have given 71 for the count with the kind supplied and accepted every message as `other` in the last query.

28.1 Event Sequence Order

The **Event Sequence Order** of a set of events is the order in which they occurred, which the analyst must often reconstruct from partial knowledge: a set of events, a set of constraints of the form "this happened before that", and the question of which total orders are consistent with the constraints. A trace gives the order of the messages of one thread directly, by their numbers; the hard case is the order of events across threads, or across traces with unsynchronized clocks, where only some orderings are known. This is a constraint problem, and Chapter 14's CLP(FD) solves it: a position variable per event, all distinct, with an inequality per known constraint.

```
:- use_module(library(clpfd)).

% event_before(?Earlier, ?Later): a known ordering
% constraint between two events.
event_before(connect, query).
event_before(query, build).
event_before(connect, send).
event_before(send, reply).

% event_sequence_order(-Events, -Order): Order
% is a total order of Events consistent with
% every event_before/2 constraint.
event_sequence_order(Events, Order) :-
    setof(E,
          X^( event_before(E, X)
```

```
            ; event_before(X, E) ),
          Events),
    pairs_keys_values(Pairs, Events, Vars),
    length(Events, L),
    Vars ins 1..L,
    all_distinct(Vars),
    findall(A-B, event_before(A, B), Edges),
    post_order(Edges, Pairs),
    label(Vars),
    pairs_keys_values(KV, Vars, Events),
    keysort(KV, Sorted),
    pairs_values(Sorted, Order).

% post_order(+Edges, +Pairs): post VA #< VB for
% each edge A-B, where Pairs maps events to vars.
post_order([], _).
post_order([A-B|Edges], Pairs) :-
    memberchk(A-VA, Pairs),
    memberchk(B-VB, Pairs),
    VA #< VB,
    post_order(Edges, Pairs).

?- event_sequence_order(_Es, Order).
Order = [connect, query, build, send, reply] ;
Order = [connect, query, send, build, reply] ;
Order = [connect, send, query, build, reply] ;
...
```

Six total orders satisfy the four constraints, enumerated on backtracking by `label/1`; the first puts the whole query-and-build chain before the send, the others interleave the two chains, and all agree that `connect` is first and that `send` precedes `reply`. The constraints must be posted by `post_order/2`, a direct recursion, and not by `forall/2`: a constraint posted inside the double negation of `forall/2` is retracted when the negation completes, and the predicate would then enumerate every permutation. This is the one place where the procedural reading of `\+` matters for CLP(FD), and it is a frequent error. The formalization reconstructs order from order constraints only; a constraint that one event is at a particular position, or that two events are simultaneous, needs a different posting, and the events here are named by hand, where a real reconstruction draws them from the trace.

28.2 Event Sequence Phase

An **Event Sequence Phase** is a maximal stretch of a trace in which one kind of activity dominates: a startup phase, a steady-serving phase, a saturation phase, a recovery phase, a shutdown phase. The phases are the coarsest useful description of a trace, the chapter headings of its story, and they are bounded by the significant events of Chapter 27. The predicate computes the boundaries from the trace rather than fixing them, each boundary the first message of a kind that marks a phase change, and assigns each message to the latest phase whose boundary it has reached.

```
% phase_boundaries(-Bs): the first message
% number of each phase after startup, in order.
phase_boundaries([B1, B2, B3, B4, B5]) :-
    aggregate_all(min(N), kind(N, received), B1),
```

```
    aggregate_all(min(N), kind(N, queued), B2),
    aggregate_all(min(N),
        ( text_of(N, T),
          sub_string(T, 0, _, _, "Housekeeping")
        ), B3),
    aggregate_all(min(N), msg(N, _, _, 109, _, _),
                  B4),
    aggregate_all(min(N),
        ( text_of(N, T),
          sub_string(T, 0, _, _,
                     "Service stopping")
        ), B5).

% message_phase(?N, -Phase): N's phase.
message_phase(N, Phase) :-
    phase_boundaries([B1, B2, B3, B4, B5]),
    msg(N, _, _, _, _, _),
    (   N >= B5 ->  Phase = shutdown
    ;   N >= B4 ->  Phase = failure
    ;   N >= B3 ->  Phase = drain
    ;   N >= B2 ->  Phase = saturation
    ;   N >= B1 ->  Phase = serving
    ;   Phase = startup
    ).

% phase_extent(?Phase, -Lo-Hi, -Count): the first
% and last message of Phase and how many it holds.
phase_extent(Phase, Lo-Hi, Count) :-
    distinct(Phase, message_phase(_, Phase)),
    findall(N, message_phase(N, Phase), Ns),
    min_list(Ns, Lo),
    max_list(Ns, Hi),
    length(Ns, Count).

?- phase_extent(Phase, Extent, Count).
Phase = startup,
Extent = 1-10,
Count = 10 ;
Phase = serving,
Extent = 11-37,
Count = 27 ;
Phase = saturation,
Extent = 38-47,
Count = 10 ;
Phase = drain,
Extent = 48-66,
Count = 19 ;
Phase = failure,
Extent = 67-70,
Count = 4 ;
Phase = shutdown,
Extent = 71-71,
Count = 1.
```

The six phases are the whole trace in outline: ten messages of startup, twenty-seven of serving while the first requests run and the report build of request 2 begins to thrash, ten of saturation once the pool is full and requests queue, nineteen of drain once the watchdog runs and worker 101 empties

the queue, four of failure on thread 109, and the one shutdown message. The phases here happen not to overlap, because each boundary is strictly later than the one before, which a trace with a recovery that relapses would violate; `message_phase/2` would then assign by the last boundary reached and lose the relapse. The boundaries are also chosen by hand as the kinds that matter in this trace, and a different analysis draws different phases from the same messages.

28.3 Macrofunction

A **Macrofunction** is an activity that contains other activities, recognized in a trace as a start/end pair that brackets further start/end pairs: a request that contains a query and a build, a transaction that contains several statements. The nesting is a tree, and recovering it is parsing: the sequence of open and close events is a sentence of a bracket language, and a definite clause grammar (Chapter 12) is its parser. Each message is first mapped to an `open`, `close`, or `leaf` token by its kind, in the same form as `kind_of/2`, the token chosen by if-then-else and unified with the argument afterwards, and the grammar folds the tokens into a forest.

```
% open_close(?Open, ?Close): Open begins and Close
% ends a bracketed activity.
open_close(dispatched, completed).
open_close(query_start, query_end).
open_close(build_start, build_end).

% ev(+N, ?Token): the bracket token of message N:
% open for a kind that begins a bracket, close for
% one that ends it, leaf for any other.
ev(N, Token) :-
    kind(N, K),
    (   open_close(K, _)
    ->  Token = open(K, N)
    ;   open_close(K0, K)
    ->  Token = close(K0, N)
    ;   Token = leaf(K, N)
    ).

% macrofunction(+TID, -Trees): the nesting forest
% of thread TID's activities.
macrofunction(TID, Trees) :-
    thread_sequence(TID, Ns),
    maplist(ev, Ns, Events),
    phrase(forest(Trees), Events).

forest([T|Ts]) --> tree(T), forest(Ts).
forest([]) --> [].

tree(node(K, N, Children)) -->
    [open(K, N)],
    forest(Children),
    [close(K, _)].
tree(leaf(K, N)) -->
    [leaf(K, N)].
```

```
% print_forest(+Trees, +Indent): print the forest
% with two spaces of indentation per level.
print_forest(Trees, Indent) :-
    forall(member(T, Trees),
           print_tree(T, Indent)).

print_tree(leaf(K, N), Indent) :-
    format("~*c~a ~d~n", [Indent, 0' , K, N]).
print_tree(node(K, N, Children), Indent) :-
    format("~*c~a ~d~n", [Indent, 0' , K, N]),
    Deeper is Indent + 2,
    print_forest(Children, Deeper).

?- once(macrofunction(101, _Trees)),
   print_forest(_Trees, 0).
other 7
dispatched 12
  query_start 13
  build_start 15
dispatched 53
  query_start 54
  build_start 56
dispatched 59
dispatched 61
dispatched 63
dispatched 65
true.
```

Worker 101's forest shows the structure the kind sequence hides: requests 1 and 5 are dispatches that contain a query and a build, and requests 6 to 9, drained from the queue, are dispatches that contain nothing, because the queue drain reused the already-built reports. The grammar is the difference-list translation of Chapter 12 doing real work: `tree//1` consumes a matching `open` and `close` with a `forest//1` between them, and the recursion gives the nesting for free. The parse here is unambiguous because an `open` token can begin only a `node` and a `leaf` token only a `leaf`; a trace whose closes do not match its opens, a `completed` with no `dispatched`, fails to parse entirely, which is correct but unhelpful, since the pattern of an unbalanced bracket is exactly what an analyst wants located, not merely reported as a failure.

28.4 Motif

A **Motif** is a short sequence of message kinds that recurs through a trace, whether or not the analyst knew to look for it: the signature of a repeated operation, discovered by counting rather than by matching a template. Where the **Characteristic Message Block** (page 574) of Chapter 27 is a known sequence sought in the trace, a motif is an unknown sequence found by enumerating the subsequences of a given length and counting their occurrences. The enumeration is `append/3` twice, as in Chapter 5, over each thread's kinds; the counting is the run-length histogram of Chapter 9.

```
% window(+Len, +List, -Sub): Sub is a contiguous
% sub-sequence of List of length Len.
window(Len, List, Sub) :-
    length(Sub, Len),
    append(_, Rest, List),
    append(Sub, _, Rest).

% all_windows(+Len, -Windows): every length-Len
% window of every thread's kind sequence.
all_windows(Len, Windows) :-
    findall(Sub,
            ( trace_thread(TID),
              thread_kinds(TID, Kinds),
              window(Len, Kinds, Sub)
            ),
            Windows).

% motif(+Len, -Motif, -Count): Motif is a
% length-Len sequence of kinds occurring Count
% times, Count at least two.
motif(Len, Motif, Count) :-
    all_windows(Len, Windows),
    msort(Windows, Sorted),
    run_counts(Sorted, Counts),
    member(Motif-Count, Counts),
    Count >= 2.

?- findall(C-M, motif(3, M, C), _Pairs),
   sort(0, @>=, _Pairs, _Sorted),
   forall(member(C-M, _Sorted),
          format("~t~d~3| ~w~n", [C, M])).
  7 [other,other,other]
  4 [resize,resize,resize]
  4 [completed,dispatched,completed]
  3 [received,received,received]
  3 [query_start,query_end,build_start]
  3 [dispatched,query_start,query_end]
  3 [dispatched,completed,dispatched]
  2 [received,queued,received]
  2 [queued,received,queued]
  2 [query_end,build_start,build_end]
  2 [other,dispatched,query_start]
  2 [other,dispatched,other]
  2 [error,received,error]
  2 [dispatched,other,other]
  2 [build_start,build_end,completed]
  2 [build_end,completed,dispatched]
true.
```

The motifs rank the trace's repetitions. The run of three `other` messages is the startup chatter and the common noise; `[completed, dispatched, completed]` is the queue drain, where a worker finishes one request and immediately starts the next; `[resize, resize, resize]` is the thrashing report build; `[dispatched, query_start, query_end]` is the head of a served request. None was specified in advance, and the frequent motifs are the trace's routines made visible. The motif is defined within a thread, so a motif split across threads, which a request handed from one worker to another would be, is not found; the length is fixed per query, and a motif longer than the window is

seen only as its overlapping pieces. A motif is also only a sequence of kinds, and two occurrences with the same kinds but different data are counted as one recurrence, which is the point when the recurrence is what matters and a loss when the data is.

28.5 Dominant Event Sequence

The **Dominant Event Sequence** of a trace is the sequence of its principal events with the incidental ones removed: the spine of the story, which Chapter 26 identified informally and which this section computes. The book's operational definition divides the trace into fixed windows, the **Trace Frames** (page 493) of Chapter 24, and takes the most frequent kind of each window; the sequence of these modal kinds is the dominant sequence, coarse enough to read at a glance and faithful to where the trace spent its messages.

```
% trace_frame(+Size, ?Index, -Ns): the messages of
% the Index-th window of Size messages.
trace_frame(Size, Index, Ns) :-
    aggregate_all(max(N), msg(N, _, _, _, _, _),
                  Max),
    Frames is (Max + Size - 1) // Size,
    between(1, Frames, Index),
    Lo is (Index - 1) * Size + 1,
    Hi is Index * Size,
    findall(N,
            ( between(Lo, Hi, N),
              msg(N, _, _, _, _, _)
            ),
            Ns).

% modal_kind(+Ns, -Kind): the most frequent kind
% among the messages Ns.
modal_kind(Ns, Kind) :-
    findall(K, ( member(N, Ns), kind(N, K) ), Ks),
    msort(Ks, Sorted),
    run_counts(Sorted, Counts),
    sort(2, @>=, Counts, [Kind-_|_]).

% dominant_event_sequence(+Size, -Seq): the modal
% kind of each non-empty frame of Size messages.
dominant_event_sequence(Size, Seq) :-
    findall(K,
            ( trace_frame(Size, _, Ns),
              Ns \== [],
              modal_kind(Ns, K)
            ),
            Seq).
```

```
?- dominant_event_sequence(10, _Seq),
   print_kinds(_Seq).
  1 other
  2 dispatched
  3 other
  4 resize
  5 error
  6 completed
  7 completed
  8 other
true.

?- dominant_event_sequence(18, _Seq),
   print_kinds(_Seq).
  1 other
  2 resize
  3 other
  4 completed
true.
```

At a window of ten the dominant sequence is the trace in eight strokes: startup chatter, the first dispatches, more startup and the request-2 setup, the resize thrashing, the queue errors, two windows of the drain's completions, and the failure and shutdown as a last `other`. At a window of eighteen it is four strokes: the startup chatter, the resize thrashing, the saturated middle where requests queue, and the completing drain. The sequence depends on the window, which is the parameter the analyst tunes, and the modal kind of a window with no clear majority is arbitrary among the ties, broken here by the standard order of the counts, so a window split evenly between two kinds reports whichever sorts first. `sort/4` with `@>=` on the second argument keeps the ties and takes the first; `sort(2, @>, ...)` would drop all but one count per value and is wrong here, since two kinds can share a count.

28.6 Inter-Correlation and Intra-Correlation

Intra-Correlation relates the messages of a trace to each other: which kinds tend to occur together, which follow which, how the trace is bound internally. **Inter-Correlation** relates the messages of two traces: which activities they share, where they agree and diverge. The two are the same operation at different scopes, and the book formalizes the first as the co-occurrence of message kinds on a thread and the second as the sharing of a thread identifier between two traces, which Section 28.7 provides.

```
% intra_correlation(?K1, ?K2, -Count): kinds K1
% and K2, with K1 before K2 in standard order,
% occur together on Count threads, Count >= 2.
intra_correlation(K1, K2, Count) :-
    findall(A-B,
            ( trace_thread(TID),
              aggregate_all(set(K),
                  ( msg(N, _, _, TID, _, _),
                    kind(N, K) ),
                  Ks),
              member(A, Ks),
```

```
              member(B, Ks),
              A @< B
            ),
            Pairs),
    msort(Pairs, Sorted),
    run_counts(Sorted, Counts),
    member((K1-K2)-Count, Counts),
    Count >= 2.

?- findall(C-(K1-K2),
      intra_correlation(K1, K2, C), _Ps),
   sort(0, @>=, _Ps, _S),
   forall(limit(5, member(C-P, _S)),
          format("~t~d~3| ~w~n", [C, P])).
  4 dispatched-other
  2 query_end-query_start
  2 other-received
  2 other-query_start
  2 other-query_end
true.
```

The strongest intra-correlation is between `dispatched` and `other`, on four threads, because every worker both serves requests and writes a startup line; the next ranks pair `other` with the request kinds for the same reason, and `query_end` with `query_start` on the two threads that ran full requests. The measure is co-occurrence on a thread, a coarse correlation that ignores order and distance; a finer one counts how often one kind immediately follows another, which is the motif of Section 28.4 at length two, and a finer one still measures the delay between them, which is Chapter 25's **Time Delta** (page 511). Correlation is not causation here any more than elsewhere: `dispatched` and `other` co-occur because every worker does both, not because one causes the other.

28.7 Split Trace and Master Trace

A **Split Trace** is one activity recorded across several trace files, because each component logged separately, or each machine, or each tracing session. A **Master Trace** is the single ordered trace reconstructed by merging them, which an analyst must build before the activity can be read as one. The merge is by time when the clocks agree and by whatever ordering constraints are available when they do not; this section merges by time. The traces are given in the `msg/7` form of Chapter 16, with a trace identifier first, declared `multifile` as in Chapter 27 so that they extend one predicate.

```
:- multifile msg/7.

% trace a: the appserver log alone.
msg(a, 1, 0.000, 4120, 100, appserver,
    "Service starting, version 3.2.1.0").
msg(a, 2, 1.002, 4120, 100, appserver,
    "Request 1 received: GET /report/17").
msg(a, 3, 1.311, 4120, 101, appserver,
    "Request 1 completed: 200, 309 ms").
msg(a, 4, 14.010, 4120, 100, appserver,
    "Service stopping").
```

```
% trace b: the dbclient log, same run.
msg(b, 1, 0.115, 4120, 100, dbclient,
    "Connecting to db01:5432").
msg(b, 2, 0.312, 4120, 100, dbclient,
    "Connected to db01:5432, session=71").
msg(b, 3, 1.104, 4120, 101, dbclient,
    "Query start: SELECT report 17").
msg(b, 4, 14.102, 4120, 109, dbclient,
    "Error: copy failed, code=0x3e6").

% trace_source(?T): a split trace to be merged.
trace_source(a).
trace_source(b).

% master_trace(-Rows): the split traces merged
% into one time-ordered list of row/5 terms.
master_trace(Rows) :-
    findall(row(Time, T, N, TID, Text),
            ( trace_source(T),
              msg(T, N, Time, _, TID, _, Text)
            ),
            Rows0),
    predsort(by_time, Rows0, Rows).

% by_time(-Order, +Row1, +Row2): order rows by
% time, breaking ties by trace then number so
% that no two distinct rows compare equal.
by_time(Order, row(T1, Tr1, N1, _, _),
               row(T2, Tr2, N2, _, _)) :-
    (   compare(O0, T1, T2), O0 \== (=)
    ->  Order = O0
    ;   compare(O1, Tr1, Tr2), O1 \== (=)
    ->  Order = O1
    ;   compare(Order, N1, N2)
    ).

% print_master(+Rows): one line per merged row.
print_master(Rows) :-
    forall(member(row(Time, T, N, TID, Text),
                  Rows),
           format("~3f ~a~d ~d ~s~n",
                  [Time, T, N, TID, Text])).

?- master_trace(_Rows), print_master(_Rows).
0.000 a1 100 Service starting, version 3.2.1.0
0.115 b1 100 Connecting to db01:5432
0.312 b2 100 Connected to db01:5432, session=71
1.002 a2 100 Request 1 received: GET /report/17
1.104 b3 101 Query start: SELECT report 17
1.311 a3 101 Request 1 completed: 200, 309 ms
14.010 a4 100 Service stopping
14.102 b4 109 Error: copy failed, code=0x3e6
true.
```

The two four-message logs merge into the eight-message story: the service starts, the database connection is made and the session opened, request 1 arrives, its query runs, it completes, and much later the service stops after the buffer error. `predsort/3` does the merge, and `keysort/2` could not,

since it keeps duplicate keys but gives no way to break their ties; and the comparator is written to be a total order: it breaks a tie in time by the trace identifier and then by the message number, so that two rows never compare equal. This matters because `predsort/3` drops elements its comparator calls equal, as Chapter 9 warned, and a comparator that compared only the time would silently discard the second of two messages logged in the same millisecond. The merge is only as trustworthy as the clocks; with skewed clocks the time order is wrong, and the merge must fall back on the **Event Sequence Order** (page 603) of Section 28.1, using the cross-trace constraints, shared session keys and request identifiers, that the two logs do share.

28.8 Quotient Trace

A **Quotient Trace** is the trace seen through an equivalence relation: the messages grouped into classes that are treated as one, so that the trace of 71 messages becomes a trace of a dozen kinds, or of a handful of threads, or of two components. Forming the quotient is how an analyst changes the grain of a trace without losing track of what each coarse element stands for, and the natural equivalence here is equality of kind. The predicate groups the message numbers by kind with `keysort/2` and `group_pairs_by_key/2` of Chapter 9.

```
% quotient_trace(-Classes): the message numbers
% grouped into Kind-Members classes.
quotient_trace(Classes) :-
    findall(K-N, kind(N, K), Pairs),
    keysort(Pairs, Sorted),
    group_pairs_by_key(Sorted, Classes).

?- quotient_trace(_Classes),
   forall(member(_K-_Ns, _Classes),
          ( length(_Ns, L),
            format("~a ~d~n", [_K, L]) )).
build_end 2
build_start 3
completed 6
dispatched 9
error 4
other 22
query_end 3
query_start 3
queued 3
received 10
resize 6
true.
```

The quotient is the trace as eleven classes with their sizes, and the sizes are already a diagnosis: nine dispatches but only six completions, ten receiveds, four errors, and twenty-two messages of no recognized kind, which measures how much of the trace the kinds leave unexplained and which Chapter 27's **Message Cover** (page 597) set out to reduce. Each class keeps its members, so the quotient can be refined back to the messages at any point; `group_pairs_by_key/2` requires the pairs sorted by key, which `keysort/2` does stably, keeping the members of each class in message order. The equivalence is a choice, and equality of kind is one of many: by thread the quotient is seven

classes, by component four, by phase six, and each answers a different question about the same trace.

28.9 Impossible Trace

An **Impossible Trace** is a trace, or a set of ordering claims about a trace, that could not have arisen from any real execution, because its constraints are jointly unsatisfiable: a reply before its request, a lock acquired twice without release, an effect before its cause. Recognizing impossibility is the same constraint computation as **Event Sequence Order** (page 603), with the opposite outcome of interest: the order exists if and only if the constraints are satisfiable, so an impossible trace is one for which the constraint solver finds no solution. The predicate takes the ordering claims as edges and succeeds when an order exists.

```
% orderable(+Edges, -Order): Order is a total
% order of the events in Edges satisfying every
% edge; fails when the edges are contradictory.
orderable(Edges, Order) :-
    setof(E,
          X^( member(E-X, Edges)
            ; member(X-E, Edges) ),
          Events),
    pairs_keys_values(Pairs, Events, Vars),
    length(Events, L),
    Vars ins 1..L,
    all_distinct(Vars),
    post_order(Edges, Pairs),
    label(Vars),
    pairs_keys_values(KV, Vars, Events),
    keysort(KV, Sorted),
    pairs_values(Sorted, Order).

?- orderable([connect-query, query-send,
              send-reply], Order).
Order = [connect, query, send, reply].

?- orderable([connect-query, query-send,
              send-reply, reply-connect], Order).
false.
```

The first set of edges is a chain and has one order; the second adds `reply-connect`, closing the chain into a cycle, and `orderable/2` fails, because no assignment of distinct positions can make every event both before and after another around a cycle. The failure is the recognition: a trace asserting those four orderings is impossible. CLP(FD) decides this by propagation and search, and for the small cycle here propagation alone suffices, `all_distinct/1` and the strict inequalities leaving no value for `label/1` to try. The method proves impossibility only relative to the constraints given; a trace that is impossible for a reason not expressed as an ordering constraint, a counter that decreases where it must only increase, needs that reason stated as a constraint too, and the absence of a solution is only as strong as the model is complete.

28.10 Data Reversal

A **Data Reversal** is a place in a trace where data appears to flow backward in time: a value received before it was sent, a result read before it was written, a reply logged before its request. It is usually an artifact of buffering or of unsynchronized logging rather than a real causal inversion, but it must be recognized, because it breaks every analysis that assumes the trace order is the causal order. The trace `trace1.pl` has no reversal, which is worth confirming; a small reordered excerpt shows one, two messages in which a reply for a session precedes its send on the same thread.

```
:- multifile msg/7.

% a buffer-reordered excerpt: the reply was
% flushed before the send that caused it.
msg(rev, 1, 5.000, 4120, 120, dbclient,
    "Reply received: session=90, bytes=256").
msg(rev, 2, 5.001, 4120, 120, dbclient,
    "Send start: session=90, bytes=1024").

% session_token(+Text, -Tok): Text's session=
% token.
session_token(Text, Tok) :-
    split_string(Text, " ,", " ,", Words),
    member(Tok, Words),
    sub_string(Tok, 0, 8, _, "session=").

% data_reversal(+Trace, -Key, -Reply, -Send): in
% Trace a reply for Key on a thread precedes the
% send for Key on the same thread.
data_reversal(Trace, Key, Reply, Send) :-
    msg(Trace, Reply, _, _, TID, _, RText),
    sub_string(RText, 0, _, _, "Reply received"),
    session_token(RText, Key),
    msg(Trace, Send, _, _, TID, _, SText),
    sub_string(SText, 0, _, _, "Send start"),
    session_token(SText, Key),
    Reply < Send.

?- data_reversal(rev, Key, Reply, Send).
Key = "session=90",
Reply = 1,
Send = 2 ;
false.

?- data_reversal(rev, _, _, _).
true ;
false.
```

The excerpt's reply, at record 1, precedes the send that produced it, at record 2, on the same thread and for the same session, and the predicate names the key and the two records. The reversal is almost certainly the tracing buffer flushing the two records out of order, not the network delivering a reply before the request, and the predicate cannot tell the two apart: it recognizes the inversion in the trace, and whether the inversion is in the world or only in the record is a judgment the trace

alone does not support. The recognizer is also tied to one send/reply vocabulary and one key; a reversal of a different operation, or across threads, needs the corresponding opposition and scope, as in Chapter 27's **Opposition Messages** (page 572).

28.11 Dialogue

A **Dialogue** is an exchange of messages between two parties that take turns: a client and a server, two threads passing a resource, a request and its response seen as a conversation. Reading a trace as a dialogue means following the turns between the parties rather than the messages of either alone, and the parties are identified by a shared datum, here a session key, that both write. The predicate collects the messages bearing a key, in order, tagged with their thread, and reports the keys that more than one thread touches.

```
% keyed_message(?N, -TID, -Key): message N on
% thread TID bears the session Key.
keyed_message(N, TID, Key) :-
    msg(N, _, _, TID, _, Text),
    session_token(Text, Key).

% dialogue(?Key, -Turns): the turns of the
% dialogue on Key, as N-TID pairs in order, when
% at least two threads take part.
dialogue(Key, Turns) :-
    distinct(Key, keyed_message(_, _, Key)),
    aggregate_all(set(TID),
                  keyed_message(_, TID, Key),
                  TIDs),
    TIDs = [_, _|_],
    findall(N-TID, keyed_message(N, TID, Key),
            Turns0),
    sort(Turns0, Turns).

?- dialogue(Key, Turns).
Key = "session=71",
Turns = [5-100, 26-103, 29-104, 68-109] ;
false.

?- dialogue(Key, _Turns),
   pairs_values(_Turns, _TIDs),
   sort(_TIDs, Parties).
Key = "session=71",
Parties = [100, 103, 104, 109] ;
false.
```

Session 71 is the trace's one dialogue, and it is a dialogue of four: the main thread opens it at message 5, worker 103 sends on it at message 26, worker 104 receives a reply on it at message 29, and the failing thread 109 receives on it at message 68. The turns are not a tidy alternation of two parties, which is the finding: a session that should have belonged to one request is touched by four threads, among them the two that deadlock and the one that faults, and the dialogue view is how that sharing is read as a sequence. The predicate's notion of a turn is the appearance of the key, which conflates the parties' roles; who spoke and who listened is in the verb of each message, `Send`

against `received`, which Chapter 27's **Opposition Messages** (page 572) distinguishes and this predicate does not.

28.12 Minimal Trace

A **Minimal Trace** is the smallest selection of messages that still exhibits some property of the whole: every kind that occurs, every component, every thread, every error, so that a reader or a test can work from a few messages instead of thousands. Constructing one is a covering problem; the book takes the simple and common case of covering every kind with its first occurrence, which gives one representative message per kind.

```
% minimal_trace(-Ns): the first message of each
% kind, in message order.
minimal_trace(Ns) :-
    setof(K, N^kind(N, K), Kinds),
    foldl(first_of_kind, Kinds, [], Ns0),
    sort(Ns0, Ns).

% first_of_kind(+K, +Acc, -Acc1): add the first
% message of kind K to the accumulator.
first_of_kind(K, Acc, [N|Acc]) :-
    aggregate_all(min(M), kind(M, K), N).

?- minimal_trace(_Ns), print_messages(_Ns).
  1 100 Service starting, version 3.2.1.0
 11 100 Request 1 received: GET /report/17
 12 101 Request 1 dispatched to worker 101
 13 101 Query start: SELECT report 17
 14 101 Query end: 1 row, 44 ms
 15 101 Build start: report 17
 16 101 Build end: report 17, 2611 nodes, 261 ms
 17 101 Request 1 completed: 200, 309 ms
 31 102 Warning: table resize, 8192 -> 16384
 38 100 No idle worker, request 5 queued
 43 100 Error: queue depth 3 exceeds threshold 2
true.
```

Eleven messages, one per kind, reproduce the trace's vocabulary in eleven of its seventy-one messages, about a sixth of its length, and they read as a plausible miniature run: a service starts, a request arrives and is dispatched, queried, built, and completed, a table resizes, a request queues, and an error is logged. `foldl/4` of Chapter 10 threads the accumulator through the kinds, and `aggregate_all(min(M), ...)` takes the earliest message of each, calling `kind/2` with the kind bound, the mode the check at the start of the chapter exercises. The first message of the trace is itself of kind `other`, so this transcript could not reveal a classification that accepted every message as `other`; the count of `other` messages in that check can, which is why the check is made where the helper is defined and not here. The minimality here is one representative per kind, which is minimal for the property "every kind occurs" and not for others: it does not preserve the order relationships, so the deadlock, which is a relationship between messages 25 and 30 and the absence of their

completions, does not survive, and a minimal trace that must preserve a pattern has to be built against that pattern, not against the vocabulary.

28.13 Visitor Trace

A **Visitor Trace** is the trace of a single agent, a thread or a request, as it moves through the components of a system, recording where it went rather than what happened at each place: the itinerary of a visitor through the parts of the program. It is read to see the path an activity took and to compare the paths of activities that should be alike. The predicate collects the components a thread wrote to.

```
% visitor_trace(?TID, -Components): the set of
% components thread TID wrote to, when it
% visited at least three.
visitor_trace(TID, Components) :-
    trace_thread(TID),
    aggregate_all(set(C),
                  msg(_, _, _, TID, C, _),
                  Components),
    length(Components, L),
    L >= 3.

?- visitor_trace(TID, Components).
TID = 100,
Components = [appserver, cachelib, dbclient] ;
TID = 101,
Components = [appserver, dbclient, reportengine] ;
TID = 102,
Components = [appserver, dbclient, reportengine] ;
false.
```

Three threads visited three components each. The main thread 100 touched the cache at startup, which no worker did; workers 101 and 102 took the same path through the application server, the database client, and the report engine, which is the itinerary of a report request. The identical component sets of 101 and 102 are a correlation worth the analyst’s attention precisely because their outcomes differed, one completing and one thrashing, and a difference in outcome on an identical path points at the data, not the path. The predicate records the set of components and loses the order and the repetition; a visitor trace that preserved the order, `appserver, dbclient, reportengine, appserver`, would be the thread’s kind sequence projected onto components, which the thread of activity of Chapter 26 already gives and which `aggregate_all(set(...))` here deliberately collapses.

28.14 Semantic Mapping

A **Semantic Mapping**, a pattern added to the catalog after the fifth edition of the reference (Trace Analysis Patterns, Part 260), assigns each message kind to a term in a vocabulary of meaning: a request arrival, database input/output, computation, back-pressure, a fault. It lifts the trace from

the words a programmer happened to log to the concepts an analysis reasons about, and it is the join between a trace and a body of knowledge expressed in those concepts. The mapping is facts from kind to semantic class, and a message's class is its kind's class.

```
% semantic(?Kind, ?Class): the meaning class of a
% message kind.
semantic(received, request_arrival).
semantic(dispatched, request_arrival).
semantic(completed, request_done).
semantic(query_start, db_io).
semantic(query_end, db_io).
semantic(build_start, cpu_work).
semantic(build_end, cpu_work).
semantic(resize, cpu_work).
semantic(queued, back_pressure).
semantic(error, fault).
semantic(other, other).

% mapped(?N, -Class): message N's semantic
% class.
mapped(N, Class) :-
    kind(N, Kind),
    semantic(Kind, Class).

% class_count(?Class, -Count): how many
% messages map to Class.
class_count(Class, Count) :-
    distinct(Class, semantic(_, Class)),
    aggregate_all(count, mapped(_, Class), Count).

?- class_count(Class, Count).
Class = request_arrival,
Count = 19 ;
Class = request_done,
Count = 6 ;
Class = db_io,
Count = 6 ;
Class = cpu_work,
Count = 11 ;
Class = back_pressure,
Count = 3 ;
Class = fault,
Count = 4 ;
Class = other,
Count = 22.
```

The trace in six meaning classes and a remainder says what the run was made of: nineteen messages of requests arriving and being dispatched against six of them finishing, eleven of computation, six of database input/output, three of back-pressure, and four faults. The gap between nineteen arrivals and six completions is the trace's central fact restated in semantic terms, and it is now in a vocabulary an analysis can share with a dump or with another trace. The mapping is a model, and its classes are chosen for a purpose; `resize` is mapped to computation here because the thrashing is the report build's work, and a capacity-planning analysis would map it to memory pressure

instead. The same kind, under two analyses, is two meanings, and the mapping is where that choice is recorded and made inspectable.

28.15 Karnaugh Map

A **Karnaugh Map** of a trace, likewise a later addition to the catalog (Trace Analysis Patterns, Part 261), is a table of the presence or absence of each message kind, or each semantic class, across the parts of a partition: a grid of ones and zeros that shows at a glance which activities each thread or component engaged in and, read down its columns, which parts share an activity. The name is borrowed from the Boolean minimization table, and the use is the same, to see structure in a field of bits. The predicate builds the grid of semantic classes against threads.

```
% semantic_classes(-Classes): the semantic classes
% in standard order.
semantic_classes(Classes) :-
    setof(C, K^semantic(K, C), Classes).

% present(+TID, +Class, -Bit): 1 if thread TID
% has a message of Class, else 0.
present(TID, Class, Bit) :-
    (   once(( msg(N, _, _, TID, _, _),
               mapped(N, Class) ))
    ->  Bit = 1
    ;   Bit = 0
    ).

% karnaugh_map: print the thread-by-class grid,
% one row per thread, the bits in class order.
karnaugh_map :-
    semantic_classes(Classes),
    forall(trace_thread(TID),
           ( findall(Bit,
                     ( member(C, Classes),
                       present(TID, C, Bit) ),
                     Bits),
             atomic_list_concat(Bits, ' ', Row),
             format("~t~d~5|  ~w~n", [TID, Row])
           )).

?- semantic_classes(_Cs),
   forall(member(C, _Cs), writeln(C)).
back_pressure
cpu_work
db_io
fault
other
request_arrival
request_done
true.
```

```
?- karnaugh_map.
  100  1 0 0 1 1 1 0
  101  0 1 1 0 1 1 1
  102  0 1 1 0 1 1 0
  103  0 0 0 0 1 1 0
  104  0 0 0 0 1 1 0
  108  0 0 0 0 1 0 0
  109  0 0 0 1 1 1 0
true.
```

The columns are the seven classes in the order the first query prints them, and each row is a thread. The grid separates the threads by what they did: 101 is the only complete worker, the only row with a one in the last column, `request_done`; 102 did the same work but never finished, its row 101's with the last bit cleared; 103 and 104 are identical, each only arriving and doing `other`, the two deadlocked workers that got no further; 100 carries the only back-pressure bit, from the queue it managed, and the two fault bits fall on 100 and 109. Reading down the `request_done` column, exactly one thread completed anything, which is the trace's failure in one bit. The map is Boolean and loses all counts, so a thread with one error and a thread with a hundred are the same bit; it is a map of what occurred, not how much, and its value is exactly that compression.

28.16 State Dump

A **State Dump** is a message, or a block of messages, in which a component writes out its internal state: a configuration, a table of counters, a snapshot of a data structure, dumped into the trace at a checkpoint or on an error. It is dense and irregular where the surrounding trace is a stream of events, and it is read differently, as a structure to be parsed rather than an event to be placed. In `trace1.pl` the nearest thing is the resize warnings, each of which dumps the old and new sizes of the report engine's table; the predicate extracts the numeric state from such a message.

```
% numbers_in(+Text, -Numbers): Text's numbers.
numbers_in(Text, Numbers) :-
    split_string(Text, " ,->", " ,->", Parts),
    convlist([P, V]>>number_string(V, P), Parts,
             Numbers).

% state_dump(?N, -Before, -After): message N
% dumps a table resize from Before to After.
state_dump(N, Before, After) :-
    kind(N, resize),
    text_of(N, Text),
    numbers_in(Text, [Before, After]).

?- state_dump(N, Before, After).
N = 31,
Before = 8192,
After = 16384 ;
N = 32,
Before = 16384,
After = 32768 ;
```

```
N = 33,
Before = 32768,
After = 65536 ;
...

?- findall(After, state_dump(_, _, After), _As),
   max_list(_As, Max).
Max = 524288.
```

The six resize messages dump the table's size at six checkpoints, and read together they are the state of one data structure over time: a table doubling from 8192 to 524288 entries in two and a half seconds, which is the report build of request 2 failing to converge. `convlist/3` with `number_string/2` extracts the two sizes, and the `After` of each message is the `Before` of the next, which a consistency check could confirm and which the pattern leaves implicit. A true state dump is richer than two numbers, a nested structure serialized into text, and extracting it is parsing with the grammars of Chapter 12; the resize message is the simplest state dump, a snapshot of a single counter, and the principle, that the dump is read as structure and not as event, is the same at every size.

28.17 Small DA+TA

The last pattern of the chapter, and of Part IV, reads a memory dump and a software trace together. **Small DA+TA** is the author's name for a combined analysis in which a small memory dump and a small trace, each incomplete on its own, corroborate and complete each other: the dump shows a state and the trace shows how it was reached, and a fact that is a guess in one is a certainty in the other. The shared files are the pair for which they were designed, and both are consulted here. The dump is the state of the process at the moment thread 109 faulted; the trace is the sixteen seconds leading to it.

```
:- [dump1].
```

Three correspondences tie the two artifacts together, and each is a predicate that joins a fact of the trace to a fact of the dump. The first is the fault itself. The trace's message 70 reports an unhandled exception with a code in its text; the dump's `exception_record/3` fact records the same thread, code, and the faulting address. Reading the code from the trace text is the number extraction of Chapter 27, and the join is on the thread and the code.

```
% trace_exception(-TID, -Code): a thread
% reported an unhandled exception with Code.
trace_exception(TID, Code) :-
    msg(_, _, _, TID, _, Text),
    split_string(Text, " ,", " ,", Parts),
    member(P, Parts),
    sub_string(P, 0, 2, _, "0x"),
    number_string(Code, P),
    Code >= 0xc000_0000.
```

```
% exception_corroborated(-TID, -Code, -Address):
% the trace's exception on TID matches the
% dump's exception_record/3, giving the Address.
exception_corroborated(TID, Code, Address) :-
    trace_exception(TID, Code),
    exception_record(TID, Code, Address).

?- forall(exception_corroborated(T, C, A),
          format("thread ~d code ~16r at ~16r~n",
                 [T, C, A])).
thread 109 code c0000005 at 7ffb8d1031aa
true.
```

The trace says that thread 109 took an access violation and stops there; the dump says where, at an address in `dbclient`, in the frame `Buffer::Copy` that Chapter 19 read as a NULL Pointer with the offset 0x18. Neither artifact alone gives both the history and the faulting address; together they give the whole fault. The second correspondence is the deadlock. The housekeeping thread's messages 49 and 50 report that workers 103 and 104 have been waiting for about seven seconds; the dump's `waits_for/3` facts say what they wait on, and that the two waits form the cycle Chapter 20 identified as a **Deadlock (Critical Sections)** (page 388).

```
% housekeeping_wait(-TID, -Ms): the watchdog
% reported worker TID waiting Ms milliseconds.
housekeeping_wait(TID, Ms) :-
    msg(_, _, _, 108, _, Text),
    split_string(Text, " ", "", Words),
    nth1(3, Words, WTID),
    number_string(TID, WTID),
    nth1(5, Words, WMs),
    number_string(Ms, WMs).

% deadlock_corroborated(-TID, -Ms, -Addr): a
% worker the watchdog reported waiting is, in
% the dump, blocked on critical section Addr.
deadlock_corroborated(TID, Ms, Addr) :-
    housekeeping_wait(TID, Ms),
    waits_for(TID, critical_section, Addr).

?- forall(deadlock_corroborated(T, Ms, A),
          format("~d waited ~d ms on ~16r~n",
                 [T, Ms, A])).
103 waited 7001 ms on 7ff61d13a140
104 waited 6900 ms on 7ff61d13a0c0
true.
```

The trace reports the symptom, two workers stuck for seven seconds, and the dump reports the mechanism, each holding the lock the other wants: worker 103 waits on the section 104 owns and 104 on the section 103 owns, the two `waits_for/3` facts whose addresses cross. The trace alone could not say the workers were deadlocked rather than merely slow, and the dump alone could not say how long they had been stuck or that the watchdog had already noticed; the Small DA+TA says both. The third correspondence is the spiking worker. Message 51 reports worker 102 at 100% CPU,

and the dump's `thread_time/2` gives its accumulated CPU time, the **Spiking Thread** (page 335) of Chapter 18.

```
% spiking_corroborated(-TID, -Ms): a worker the
% watchdog reported at full CPU has high CPU time
% in the dump.
spiking_corroborated(TID, Ms) :-
    msg(_, _, _, 108, _, Text),
    sub_string(Text, _, _, _, "cpu 100%"),
    split_string(Text, " ", "", Words),
    nth1(3, Words, WTID),
    number_string(TID, WTID),
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

?- forall(spiking_corroborated(T, Ms),
          format("thread ~d: ~d ms~n", [T, Ms])).
thread 102: 9680 ms
true.
```

The trace's watchdog saw worker 102 at full CPU; the dump confirms it with 9680 milliseconds of accumulated time, far above any other thread, and the trace's resize warnings of Section 28.16 explain what the CPU was spent on, a report table doubling without converging. The three correspondences together reconstruct the incident that neither artifact holds alone: a report build on worker 102 that never converged and spun a core, two workers deadlocked on a pair of critical sections while the watchdog counted their wait, a queue backing up behind the stalled pool, and finally a thread faulting on a null pointer in the database client. The join is possible only because the two artifacts share identifiers, the thread numbers, the exception code, the component names, and a knowledge base that did not share them, a trace with different thread numbering than its dump, would need those identifiers reconciled first, which is the limit of the method: the correspondence is only as sound as the shared keys are trustworthy, and a mismatched key joins two facts that do not belong together as confidently as it joins two that do.

Part V. Engineering Diagnostic Knowledge

Part V turns from the patterns to the programs that hold them: meta-interpreters that explain why a recognizer succeeded and turn a proof into a diagnostic narrative; the techniques that keep knowledge bases of millions of facts tractable, from indexing and tabling to streaming parsers and compiled fact files; and the ways a Prolog knowledge base is connected to the rest of a diagnostic workflow, as a command-line tool, as a service, as an engine embedded in a debugger, and as a verifier for hypotheses proposed by people or by language models.

Chapter 29. Meta-Interpreters and Explanation

A recognizer of Parts III and IV answers yes or no, or enumerates instances, and the analyst who receives the answer has to trust it. The evidence form of Chapter 17, `evidence(Pattern, Instance, Facts)`, was written by hand for each pattern: the author of the recognizer decided which facts to report. This chapter obtains the evidence from the recognizer itself. A **meta-interpreter** is a Prolog program that executes Prolog programs: it reads the clauses of a predicate as data, with `clause/2`, and carries out the resolution of Chapter 4 itself instead of leaving it to the system. Because the interpreter performs every step, it can record them, and the record of the steps that led to an answer is a **proof tree**: the answer, the clause that produced it, and the proof trees of the clause's body goals, down to the facts of the knowledge base and the built-in tests that succeeded. Printed with indentation, a proof tree is an explanation of why a pattern instance was recognized, and its leaves are the evidence.

The chapter builds the interpreter in stages. The vanilla interpreter `solve/1` behaves as Prolog does. `solve/2` produces the proof tree, and `why/1` prints it. From the tree, the facts a recognizer relied on are collected automatically, which gives the evidence form of Chapter 17, under the contract of Section 17.3, without a hand-written list. The remaining sections vary the interpreter: a bound on the depth of the search that makes a cyclic wait-for graph safe to traverse, a count of the inferences in a proof, a tracer that prints the four ports of Chapter 4, and an interpreter that attaches certainty factors to heuristic conclusions, with the caution such numbers require. The last section relates these mechanisms to the author's work on analysis pattern networks. The interpreters of this chapter read the clauses of module `user`; Section 31.9 adds the module context that lets the same technique read the clauses of the module `patterns_dump` of Chapter 23.

29.1 Clauses as Data

A Prolog program is a set of clauses, and clauses are terms. `clause/2` makes the connection available to the program: `clause(Head, Body)` succeeds once for each clause of the predicate of `Head` whose head unifies with `Head`, binding `Body` to the body of that clause, in the order of the clauses. A fact has the body `true`. The first argument must be bound at least to a term with a known functor, since `clause/2` has to know which procedure to look in; an unbound first argument raises an **instantiation error**.

The interpreted program of this chapter is a small set of recognizers over `dump1.pl`, restated from Parts II and III so that `clause/2` finds them in `user`: **Blocked Thread (Software)** (page 336) and **Spiking Thread** (page 335) with the clauses Chapter 18 gave them, the two-thread **Deadlock (Critical Sections)** (page 388) of Chapter 17 with a helper `owner/2` that reads the owner of a critical section, and **Handle Leak** (page 409) with the clause of Chapter 21. The thresholds are not restated. They are imported from the module `patterns_dump` of Chapter 23 with `use_module/2` and an import list, so that the interpreted recognizers compare against the same values as Chapters 18 to 21 and recognize the same instances; the import list keeps the module's other exports, among them

recognizers with the names used here, out of `user`. The flag that hides the bindings of variables beginning with an underscore is set for the chapter, as in Chapter 9.

```
:- [dump1].
:- use_module(patterns_dump, [threshold/2]).

% blocked_thread(?TID): the dump records an object
% TID waits for.
blocked_thread(TID) :-
    waits_for(TID, _, _).

% owner(?Addr, ?TID): TID owns the critical
% section at Addr.
owner(Addr, TID) :-
    critical_section(Addr, TID, _, _),
    TID =\= 0.

% blocks(?Blocker, ?Blocked): Blocked waits for a
% critical section that Blocker owns.
blocks(Blocker, Blocked) :-
    waits_for(Blocked, critical_section, Addr),
    owner(Addr, Blocker).

% deadlock(?T1, ?T2): T1 and T2 block each other.
deadlock(T1, T2) :-
    blocks(T1, T2),
    blocks(T2, T1),
    T1 < T2.

% cpu_time(?TID, -Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% total_cpu(-Total): the CPU time of all threads.
total_cpu(Total) :-
    aggregate_all(sum(Ms), cpu_time(_, Ms),
                  Total).

% cpu_share(?TID, -Share): the CPU time of TID as
% a percentage of the total.
cpu_share(TID, Share) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Total > 0,
    Share is 100.0 * Ms / Total.

% spiking_thread(?TID): the CPU share of TID is at
% least the threshold.
spiking_thread(TID) :-
    cpu_share(TID, Share),
    threshold(spiking_share, Min),
    Share >= Min.
```

```
% handle_leak(?Type): the count of handles of Type
% exceeds the limit for one type.
handle_leak(Type) :-
    handle_count(_, Type, Count),
    threshold(handle_type_count, Max),
    Count > Max.


?- set_prolog_flag(toplevel_print_anon, false).
true.

?- clause(deadlock(A, B), Body).
Body = (blocks(A, B), blocks(B, A), A<B).

?- clause(threshold(spiking_share, V), Body).
V = 25,
Body = true.

?- clause(blocks(A, B), _Body),
   portray_clause((blocks(A, B) :- _Body)).
blocks(A, B) :-
    waits_for(B, critical_section, C),
    owner(C, A).
true.

?- clause(waits_for(101, K, Addr), Body).
K = critical_section,
Addr = 140695026507968,
Body = true.
```

The body of `deadlock/2` is the conjunction term `','(blocks(A, B), ','(blocks(B, A), A<B))`, which the toplevel writes with operators; the variables of the clause are fresh copies, shared between the head and the body, which is why `A` and `B` in the body are the `A` and `B` of the query. The second query reads a clause of an imported predicate: `clause/2` follows the import, and in this edition the clause of `threshold/2` comes back as it would for a local predicate, without a module qualifier. The third query reassembles a clause from its parts and prints it with `portray_clause/1`, the inverse of reading. The fourth reads a fact of the dump: `clause/2` does not distinguish a fact of the knowledge base from a rule of the program, and this is what will let the interpreter treat `dump1.pl`, the recognizers, and the imported thresholds as one program. The toplevel prints the address in decimal, as Chapter 2 said it would; Section 29.4 changes that.

ISO Prolog restricts `clause/2` to public procedures, those declared `dynamic`, and prescribes a **permission error** for the others. SWI-Prolog is more permissive: in this edition (9.0.4), `clause/2` reads the clauses of static user predicates, such as every predicate above, unless the flag `protect_static_code` is set to `true`, in which case the ISO permission error is raised for them. The flag is `false` by default. Built-in predicates are protected regardless of the flag, as is `true/0`; a predicate without clauses makes `clause/2` fail rather than raise the existence error that calling it would.

```
?- current_prolog_flag(protect_static_code, F).
F = false.
```

```
?- catch(clause(write(_), _),
         error(permission_error(Op, Type, PI), _),
         true).
Op = access,
Type = private_procedure,
PI = write/1.

?- catch(clause(true, _),
         error(permission_error(Op, Type, PI), _),
         true).
Op = access,
Type = private_procedure,
PI = true/0.

?- clause(no_such_predicate(_), _).
false.

?- clause((a, b), Body).
Body = call((a, b)).
```

The last answer is a SWI-Prolog particularity that matters below. The conjunction `','/2` is defined in the system as a predicate whose single clause calls its arguments, and `clause/2` shows that clause. An interpreter that reaches `clause/2` with a conjunction does not get an error; it gets `call((a, b))`, and if it then interprets `call/1` the same way it gets `call((a, b))` again, forever.

What a program may call that `clause/2` cannot read is decided with `predicate_property/2` of Chapter 11. The property `built_in` holds for the predicates of the system, among them the arithmetic comparisons, `format/2`, and the control constructs; `imported_from(Module)` holds for a predicate that the current module obtained from another, which is how library predicates such as `member/2` and `sum_list/2` appear in module `user`, and how `threshold/2` appears there after the import above. The two kinds of import are told apart by the module they come from: `module_property/2` with the property `class(Class)` reports `library` for a module of the SWI-Prolog library and `user` for a module loaded from the analyst's own files. Library predicates have clauses, but the clauses belong to the library and are not part of the program under study; the clauses of `patterns_dump` are.

```
?- predicate_property(A < B, built_in).
true.

?- predicate_property(member(_, _),
                      imported_from(M)).
M = lists.

?- predicate_property(threshold(_, _),
                      imported_from(M)).
M = patterns_dump.

?- module_property(lists, class(C1)),
   module_property(patterns_dump, class(C2)).
C1 = library,
C2 = user.
```

```
?- predicate_property(deadlock(_, _), built_in).
false.

?- predicate_property(deadlock(_, _),
                      imported_from(_)).
false.

?- predicate_property(deadlock(_, _),
                      number_of_clauses(N)).
N = 1.
```

29.2 The Vanilla Meta-Interpreter

The textbook **vanilla meta-interpreter** has three clauses: `true` is proved; a conjunction is proved by proving its parts; and any other goal is proved by finding a clause whose head unifies with it and proving the body.

```
naive_solve(true).
naive_solve((A, B)) :-
    naive_solve(A),
    naive_solve(B).
naive_solve(G) :-
    clause(G, Body),
    naive_solve(Body).
```

This is a correct description of SLD resolution, and it does not run correctly in SWI-Prolog. The third clause is not restricted to the goals the first two leave out: on backtracking, `naive_solve(true)` reaches `clause(true, Body)` and raises the permission error of Section 29.1, and `naive_solve((A, B))` reaches `clause((A, B), Body)`, obtains `call((A, B))`, and loops. Nor does it handle the built-ins, which have no clauses. The working version, `solve/1`, keeps the shape and adds what is missing: a guard that keeps control constructs out of the last clause, a test for goals the system executes, clauses for the control constructs that occur in recognizer bodies, and clauses for the two **collecting goals** the recognizers of this book use, `findall/3` and `aggregate_all/3`, whose inner goal is interpreted so that the facts it consults are seen by the interpreter.

```
% solve(+Goal): Goal is provable from the program,
% by interpretation of its clauses.
solve(true).
solve((A, B)) :-
    solve(A),
    solve(B).
solve((If -> Then ; Else)) :-
    (   solve(If)
    ->  solve(Then)
    ;   solve(Else)
    ).
solve((A ; B)) :-
    A \= (_ -> _),
    (   solve(A)
    ;   solve(B)
    ).
solve(\+ A) :-
```

```
    \+ solve(A).
solve(findall(T, G, L)) :-
    findall(T, solve(G), L).
solve(aggregate_all(Spec, G, R)) :-
    aggregate_all(Spec, solve(G), R).
solve(G) :-
    \+ control(G),
    \+ collecting(G),
    (   system_goal(G)
    ->  call(G)
    ;   clause(G, Body),
        solve(Body)
    ).

% control(@Goal): Goal is a control construct that
% solve/1 handles by a clause of its own.
control(true).
control((_, _)).
control((_ ; _)).
control(\+ _).

% collecting(@Goal): Goal collects the solutions
% of an inner goal, which solve/1 interprets.
collecting(findall(_, _, _)).
collecting(aggregate_all(_, _, _)).

% system_goal(+Goal): Goal is executed by the
% system, not interpreted: a built-in or a
% predicate of a library module.
system_goal(G) :-
    predicate_property(G, built_in).
system_goal(G) :-
    predicate_property(G, imported_from(M)),
    module_property(M, class(library)).
```

The last clause of `solve/1` is the one that does the work. Its guards `\+ control(G)` and `\+ collecting(G)` fail for the constructs the earlier clauses handle, so that a conjunction is never passed to `clause/2`; this is the role a cut plays in the textbook versions, done with a test so that each clause of `solve/1` stands on its own. The if-then-else inside the clause separates goals the system executes from goals the interpreter resolves: a built-in or library goal is called, and its success or failure is taken as given; a user goal, which includes the imported `threshold/2`, is resolved against its clauses, and the body of each matching clause is interpreted in turn. The clause for `;/2` excludes the if-then-else term, which the clause before it handles, because `(If -> Then ; Else)` is the term `;((If -> Then), Else)` and would otherwise match twice. The clauses for the collecting goals hand the system a goal of the interpreter itself: `findall(T, solve(G), L)` collects the same instances of `T` as `findall(T, G, L)`, because `solve/1` proves what Prolog proves, and the difference is that every step beneath is now the interpreter's.

```
?- solve(deadlock(A, B)).
A = 103,
B = 104 ;
false.
```

```
?- solve(blocked_thread(T)).
T = 101 ;
T = 103 ;
T = 104 ;
false.

?- solve((deadlock(A, B), \+ spiking_thread(A))).
A = 103,
B = 104 ;
false.

?- solve(spiking_thread(T)).
T = 102 ;
T = 107 ;
false.

?- solve(member(X, [a, b])).
X = a ;
X = b.
```

The answers are those the recognizers themselves give, with one visible difference. Called directly, `deadlock(A, B)` answers `A = 103, B = 104.` and leaves no choice point, because SWI-Prolog's clause indexing recognizes when the fact it is trying is the last that can match; `clause/2` hands the interpreter one clause at a time and cannot tell it that the clause is the last, so the interpreted query ends with `;` and then `false.` when the remaining choice points are exhausted. The spiking threads are 102 and 107, as in Chapter 18, since the threshold is Chapter 18's. `member/2` is a library predicate and is executed, not interpreted, which the last query shows by ending with `.` as the direct call would.

Three things the interpreter does not do. A **cut** in an interpreted body reaches the last clause as the goal `!`, which is a built-in, and `call(!)` cuts nothing outside itself: a clause that relies on a cut is interpreted as if the cut were `true`. **Meta-predicates** other than the two collecting goals, such as `forall/2` and `maplist/3`, are built-in or library predicates and run natively, so the goals inside them are not interpreted and will not appear in a proof tree. And a goal for a predicate that does not exist makes `solve/1` fail, since `clause/2` fails for it, where `call/1` would raise an existence error. The recognizers of this book avoid cuts, collect with `findall/3` and `aggregate_all/3`, and are checked by `check/0` of Chapter 13 for undefined predicates, so the three limitations cost little.

29.3 Proof Trees

The interpreter decides at each step which clause it uses. `solve/2` records the decisions. A proof tree is the term `proof(Goal, SubProofs)`, where `Goal` is the goal as it was proved, with the bindings the proof made, and `SubProofs` is the list of proof trees of the body goals of the clause that proved it. A fact has no body goals and its list is empty; so has a built-in, whose proof is its success. A collecting goal has one sub-proof per solution of its inner goal, which `collect/2` produces while it computes the collected result. The body of a clause is interpreted by `solve_body/2`, which produces a list of proofs, one per goal of the body, flattening the conjunction.

```
% solve(+Goal, -Proof): Proof is a proof tree of
% Goal.
solve(G, proof(G, Subs)) :-
    \+ control(G),
    (   collecting(G)
    ->  collect(G, Subs)
    ;   system_goal(G)
    ->  call(G),
        Subs = []
    ;   clause(G, Body),
        solve_body(Body, Subs)
    ).

% collect(+Goal, -Subs): Goal is a collecting goal
% whose inner goal has the proofs Subs, one per
% solution; the collected result is computed from
% the solutions as the goal itself computes it.
collect(findall(T, G, L), Subs) :-
    findall(T-P, solve(G, P), Pairs),
    pairs_keys_values(Pairs, L, Subs).
collect(aggregate_all(Spec, G, R), Subs) :-
    findall(Spec-P, solve(G, P), Pairs),
    pairs_keys_values(Pairs, Specs, Subs),
    aggregate_all(Spec, member(Spec, Specs), R).

% solve_body(+Body, -Proofs): Proofs is the list
% of proof trees of the goals of Body.
solve_body(true, []).
solve_body((A, B), Proofs) :-
    solve_body(A, PA),
    solve_body(B, PB),
    append(PA, PB, Proofs).
solve_body((If -> Then ; Else), Proofs) :-
    (   solve_body(If, PI)
    ->  solve_body(Then, PT),
        append(PI, PT, Proofs)
    ;   solve_body(Else, Proofs)
    ).
solve_body((A ; B), Proofs) :-
    A \= (_ -> _),
    (   solve_body(A, Proofs)
    ;   solve_body(B, Proofs)
    ).
solve_body(\+ A, [proof(\+ A, [])]) :-
    \+ solve_body(A, _).
solve_body(G, [Proof]) :-
    \+ control(G),
    solve(G, Proof).
```

The clause for `aggregate_all/3` deserves a word. Its inner goal is interpreted once per solution, and each solution leaves a copy of the specification, `sum(160)`, `sum(20)`, and so on, paired with its proof. The result is then computed by `aggregate_all/3` itself over those copies, with `member/2` as the generator: `aggregate_all(sum(Ms), member(sum(Ms), Specs), R)` sums the same values the original goal would have summed. The same line serves `count`, `max`, `min`, `bag`, and `set`, since every specification `aggregate_all/3` accepts is a term that `member/2` can unify with its copies.

```
?- solve(deadlock(A, B), proof(G, _Subs)),
   length(_Subs, N).
A = 103,
B = 104,
G = deadlock(103, 104),
N = 3 ;
false.

?- solve(handle_leak(T), proof(_, _Subs)),
   member(proof(G, Subs), _Subs).
T = event,
G = handle_count(4120, event, 12040),
Subs = [] ;
T = event,
G = threshold(handle_type_count, 5000),
Subs = [] ;
T = event,
G = (12040>5000),
Subs = [] ;
false.

?- solve(spiking_thread(107), proof(_, _Subs)),
   forall(member(proof(G, _), _Subs),
          ( print(G), nl )).
cpu_share(107,29.006013441811106)
threshold(spiking_share,25)
29.006013441811106>=25
true ;
false.

?- solve(total_cpu(T),
         proof(_, [proof(aggregate_all(_, _, _),
                         _Inner)])),
   length(_Inner, N).
T = 14135,
N = 10.

?- solve(blocked_thread(108), P).
false.
```

A proof of a negated goal is a leaf: **negation as failure** proves `\+ A` by the absence of a proof of `A`, and there is no sub-proof to record. The if-then-else records the proofs of the condition and the branch taken, and nothing about the branch not taken. The proof of `handle_leak(event)` shows the three leaves an analyst would want to see: the fact that gives the count, the fact that gives the threshold, and the comparison, with its arguments already evaluated by unification but not yet by arithmetic, since `12040>5000` is the goal as it was called. The proof of `spiking_thread(107)` has the same three kinds of sub-proof, but the first is a tree rather than a leaf, since `cpu_share/2` is a rule; and beneath it, as the fourth query shows, the proof of `total_cpu/1` has one sub-proof, the collecting goal, which has ten of its own, one per thread whose time went into the total. The query binds the one sub-proof to the pattern of a collecting goal and counts its children; the goal as recorded keeps its template and inner goal unbound, since each solution bound a copy.

29.4 Printing Proofs as Explanations

A proof tree as a term is for programs. For the analyst it is printed as an indented **explanation**: the goal, the word `because`, and the sub-proofs indented beneath it; a leaf prints as the goal alone when it is a fact of the knowledge base, and with the word `holds` when it is a built-in test or a negation. The addresses in the facts are long in decimal, and the explanation is more readable with them in hexadecimal. SWI-Prolog's hook for this is `portray/1`: when a predicate `portray/1` is defined in module `user`, `print/1`, the `~p` directive of `format/2`, and `write_term/2` with the option `portray(true)` call it for every subterm before writing it, and if it succeeds its output stands in for the subterm.

```
% portray(+Term): hook. Integers above 0xffff are
% written in hexadecimal.
portray(X) :-
    integer(X),
    X > 0xffff,
    format("0x~16r", [X]).

?- print(waits_for(101, cs, 0x7ff6_1d13_a0c0)),
   nl.
waits_for(101,cs,0x7ff61d13a0c0)
true.

?- waits_for(101, Kind, Addr).
Kind = critical_section,
Addr = 0x7ff61d13a0c0.

?- format("~p ~w~n",
          [0x7ff6_1d13_a0c0, 0x7ff6_1d13_a0c0]).
0x7ff61d13a0c0 140695026507968
true.
```

The second query shows that the hook reaches further than `print/1`: the toplevel writes answers with `portray(true)` among its options (flag `answer_write_options`), so from here on addresses in answers appear in hexadecimal too. The `~w` directive does not call the hook. The hook is crude, since it cannot see where in a term an integer stands and treats every integer above 65535 as an address; a heap size would be shown the same way. In this edition, `portray_clause/1` and `listing/1` write integers in decimal regardless of the hook.

The printer `why/1` walks the tree with a depth counter and uses the column stop `~t~*|` of Chapter 6 for indentation. Variables that a proof left unbound, such as the waiter and recursion counts in a proof of `owner/2`, would print with system-generated names; `numbervars/3` of Chapter 8 inside a **double negation** gives them letters for the duration of the printing and undoes the binding afterwards, with the option `singletons(true)` writing a variable that occurs once as `_`.

```
% why(+Goal): print a proof of Goal as an indented
% explanation; one answer per proof.
why(G) :-
    solve(G, Proof),
    print_proof(Proof, 0).

print_proof(proof(G, Subs), Depth) :-
    Indent is Depth * 2,
    \+ \+ ( numbervars(G, 0, _,
                       [singletons(true)]),
            format("~t~*|~p", [Indent, G]) ),
    (   Subs == []
    ->  (   leaf_fact(G)
        ->  nl
        ;   format(" holds~n")
        )
    ;   format(" because~n"),
        Deeper is Depth + 1,
        forall(member(Sub, Subs),
               print_proof(Sub, Deeper))
    ).

% leaf_fact(+Goal): a leaf that is a fact of the
% knowledge base, not a built-in or a negation.
leaf_fact(G) :-
    G \= (\+ _),
    \+ system_goal(G).

?- why(handle_leak(T)).
handle_leak(event) because
  handle_count(4120,event,12040)
  threshold(handle_type_count,5000)
  12040>5000 holds
T = event ;
false.

?- why(deadlock(A, B)).
deadlock(103,104) because
  blocks(103,104) because
    waits_for(104,critical_section,0x7ff61d13a0c0)
    owner(0x7ff61d13a0c0,103) because
      critical_section(0x7ff61d13a0c0,103,2,1)
      103=\=0 holds
  blocks(104,103) because
    waits_for(103,critical_section,0x7ff61d13a140)
    owner(0x7ff61d13a140,104) because
      critical_section(0x7ff61d13a140,104,1,1)
      104=\=0 holds
  103<104 holds
A = 103,
B = 104 ;
false.
```

The explanation of the deadlock is the argument an analyst would write in a report: thread 104 waits for the critical section at `0x7ff61d13a0c0`, which 103 owns; thread 103 waits for the one at `0x7ff61d13a140`, which 104 owns. It was produced from the recognizer alone, and if the recognizer is changed, the explanation changes with it, which a hand-written evidence clause does not. The

condition `103<104`, which the recognizer uses to report each pair once, appears too, because the proof records everything the clause required; an explanation for a reader would omit it, and Section 29.5 shows how the leaves are selected. The explanation of a **Spiking Thread** (page 335) is not shown: beneath `total_cpu/1` it lists the ten threads' times with the addition that computed each, three lines per thread, which is correct and is more than a reader wants; Section 29.5 reduces it to the facts.

```
?- why((deadlock(A, B), \+ spiking_thread(A))).
false.

?- why(blocked_thread(108)).
false.
```

`why/1` takes one goal, not a conjunction, because `solve/2` does, and it fails without printing when there is no proof. Failure is the right answer to the second query, and it is also the limit of the method: a meta-interpreter explains success, and the absence of a proof of `blocked_thread(108)` has no tree to print. Thread 108 is not blocked under the recognizer's definition, which is all the failure says.

29.5 Evidence from Proofs

Chapter 17 introduced the evidence form `evidence(Pattern, Instance, Facts)`, with `Facts` chosen by the author of each recognizer under the contract of Section 17.3: ground goals that hold, a fact of the knowledge base or a threshold or a derived goal with its arguments bound, and together sufficient for the instance. The leaves of a proof tree that are facts are exactly the facts the recognizer consulted, and collecting them, which `evidence/2` does, turns any recognizer into its own explainer. The **evidence** of an instance is then a projection of its proof. Two small tables complete the term: `pattern_name/2` gives the catalog name of the pattern each interpreted recognizer recognizes, since the contract asks for the name and not the functor; and the instance is the recognizer's argument, or the pair of its two arguments, as Chapter 17 fixed for pairs.

```
% pattern_name(Functor, Name): the catalog name of
% the pattern the recognizer Functor recognizes.
pattern_name(blocked_thread,
             'Blocked Thread (Software)').
pattern_name(deadlock,
             'Deadlock (Critical Sections)').
pattern_name(spiking_thread, 'Spiking Thread').
pattern_name(handle_leak, 'Handle Leak').
```

```
% proof_facts(+Proof, -Facts): the facts of the
% knowledge base among the leaves of Proof.
proof_facts(proof(G, []), Facts) :-
    (   leaf_fact(G)
    ->  Facts = [G]
    ;   Facts = []
    ).
proof_facts(proof(_, [Sub|Subs]), Facts) :-
    maplist(proof_facts, [Sub|Subs], Lists),
    append(Lists, Facts).

% evidence(+Goal, -Evidence): Evidence is
% evidence(Pattern, Instance, Facts) for one proof
% of the recognizer goal Goal: Pattern its catalog
% name, Instance its argument or its two arguments
% as a pair, Facts the facts of the proof, once.
evidence(Goal, Evidence) :-
    Evidence = evidence(Pattern, Instance, Facts),
    solve(Goal, Proof),
    Goal =.. [Functor | Args],
    pattern_name(Functor, Pattern),
    instance_of(Args, Instance),
    proof_facts(Proof, Facts0),
    list_to_set(Facts0, Facts).

instance_of([Instance], Instance).
instance_of([A, B], A-B).

% print_evidence(+Goal): one line per fact.
print_evidence(Goal) :-
    evidence(Goal, evidence(P, I, Facts)),
    format("~a: ~p~n", [P, I]),
    forall(member(F, Facts),
           format("  ~p~n", [F])).

?- print_evidence(deadlock(A, B)).
Deadlock (Critical Sections): 103-104
  waits_for(104,critical_section,0x7ff61d13a0c0)
  critical_section(0x7ff61d13a0c0,103,2,1)
  waits_for(103,critical_section,0x7ff61d13a140)
  critical_section(0x7ff61d13a140,104,1,1)
A = 103,
B = 104 ;
false.

?- print_evidence(handle_leak(T)).
Handle Leak: event
  handle_count(4120,event,12040)
  threshold(handle_type_count,5000)
T = event ;
false.

?- print_evidence(spiking_thread(107)).
Spiking Thread: 107
  thread_time(107,3200,900)
  thread_time(100,120,40)
  thread_time(101,15,5)
  thread_time(102,8450,1230)
  thread_time(103,30,10)
```

```
  thread_time(104,25,10)
  thread_time(105,20,5)
  thread_time(106,10,5)
  thread_time(108,0,0)
  thread_time(109,40,20)
  threshold(spiking_share,25)
true ;
false.
```

The four facts of the deadlock are the four lines an analyst would quote from the debugger's `!locks` and `~*k` output, and `list_to_set/2` removes a fact that two branches of the proof both used, keeping the first occurrence. The threshold fact appears in the evidence of the **Handle Leak** (page 409) and of the **Spiking Thread** (page 335), which is correct: the recognition depends on it, and a report that quotes the counts should quote the limit they were compared with. The evidence of the **Spiking Thread** (page 335) is longer than the list Chapter 18 wrote by hand, which names the derived goals `cpu_time(107, 4100)` and `total_cpu(14135)` where this one lists the ten facts beneath them. Both lists are under the contract. The hand-written one is shorter because its author chose the level at which to stop; the derived one stops at the facts because the interpreter stops there, and it says, which the shorter one leaves to the reader, that the share of thread 107 was computed against every other thread's time.

The first two parts of the contract are checked as Section 17.3 checks them: every element is a ground goal that holds, so `maplist(call, Facts)` succeeds, and the test below applies it to every instance of every interpreted recognizer. The third part, sufficiency, is not re-derived here with `entails/2`, whose conditions are written over the vocabulary of the hand-written lists, `cpu_share/2` and the threshold; over the derived list the condition would have to recompute the total from the ten facts, and the proof tree is that computation. Sufficiency holds for derived evidence by construction where it holds at all: a proof tree is a derivation of the goal, and its facts, together with the built-in tests among its leaves, are what the derivation used, so the shape follows from the list without recourse to the rest of the knowledge base. The exception is negation. A recognizer that finds an instance by `\+` consulted facts whose absence mattered, and absence leaves no fact to record; the evidence of such a recognizer is then true and insufficient, and a re-derivation would fail on it. None of the four recognizers of this chapter negates, and a recognizer that does should say in its hand-written evidence what it found absent.

```
?- forall(( member(G, [ deadlock(_, _),
                        blocked_thread(_),
                        spiking_thread(_),
                        handle_leak(_) ]),
            evidence(G, evidence(_, _, _Facts)) ),
          maplist(call, _Facts)).
true.

?- evidence(blocked_thread(101), _E),
   _E = evidence(P, I, _Facts),
   length(_Facts, N).
P = 'Blocked Thread (Software)',
I = 101,
N = 1 ;
false.
```

Two qualifications. The evidence is the evidence of one proof; a goal with several proofs, such as a thread blocked in two ways, has several evidence terms, one per answer of `evidence/2`. And the facts collected are those the recognizer consulted, not those that would convince a reader: a recognizer that tests `TID =\= 0` consulted no fact for it, and the comparison appears in the proof tree and not in the evidence. The automatic evidence is a floor, not a ceiling, for what a report should contain.

29.6 Depth-Bounded Interpretation

Chapter 11 wrote the **Wait Chain (Critical Sections)** (page 384) as a transitive closure and used tabling to make it terminate on a cyclic wait-for graph. The naive closure, written without a visited list and without tabling, does not terminate on `dump1.pl`, because the deadlock is a cycle.

```
% blocked_by(?T, ?B): T waits, directly or through
% other threads, for a critical section B owns.
% Deliberately without a visited list.
blocked_by(T, B) :-
    blocks(B, T).
blocked_by(T, B) :-
    blocks(X, T),
    blocked_by(X, B).

?- blocked_by(101, B).
B = 103 ;
B = 104 ;
B = 103 ;
B = 104 ;
...
```

Thread 101 waits for 103, which waits for 104, which waits for 103, and the second clause follows the cycle forever, producing the same two answers in turn. The query has infinitely many answers and the toplevel will give them as long as it is asked. A **depth-bounded interpreter** makes the search finite without changing the program. `solve_depth(Goal, Depth)` interprets as `solve/1` does but spends one unit of depth on each clause resolution along a branch of the proof, and fails when none is left; built-ins cost nothing, since they do not recurse through the interpreter. The bounded interpreter, and the counting and tracing interpreters of the next sections, treat the collecting goals as library goals and run them natively, which is enough for the goals they are applied to here.

```
% solve_depth(+Goal, +Depth): Goal has a proof in
% which no branch uses more than Depth clauses.
solve_depth(true, _).
solve_depth((A, B), D) :-
    solve_depth(A, D),
    solve_depth(B, D).
solve_depth(\+ A, D) :-
    \+ solve_depth(A, D).
solve_depth(G, D) :-
    \+ control(G),
    (   system_goal(G)
    ->  call(G)
```

```
    ;   D > 0,
        D1 is D - 1,
        clause(G, Body),
        solve_depth(Body, D1)
    ).

?- solve_depth(blocked_by(101, B), 6).
B = 103 ;
B = 104 ;
B = 103 ;
false.

?- solve_depth(blocked_by(101, B), 3).
false.

?- findall(B, solve_depth(blocked_by(101, B), 6),
           Bs),
   sort(Bs, Set).
Bs = [103, 104, 103],
Set = [103, 104].
```

The depth is the length of the longest branch of the proof tree, measured in clauses. A proof of `blocked_by(101, 103)` by the first clause uses `blocked_by/2`, `blocks/2`, `owner/2`, and the fact `critical_section/4`, four clauses deep, so depth 3 finds nothing and depth 6 finds the chain two steps further before the bound stops it. The repeated answer is the cycle made visible: a chain that returns to a thread it has already named is the **Deadlock (Critical Sections)** (page 388) seen from thread 101, and the bounded interpreter reports it as a repetition rather than as a hang.

Iterative deepening, which runs the bounded interpreter with depths 1, 2, 3 and so on, finds the shortest proof first and is complete where depth-first search is not; it repeats the shallow work at every round, which costs a constant factor, and it finds each answer once per depth at which it has a proof, so the answers are deduplicated afterwards. SWI-Prolog has a built-in that does the bookkeeping: `call_with_depth_limit/3` runs a goal with a limit on the recursion depth and reports the depth reached, or the atom `depth_limit_exceeded` when the limit stopped it. Its unit is the system's own notion of depth, which counts the frames of built-ins and library predicates too, and its numbers are therefore not those of `solve_depth/2`; a companion, `call_with_inference_limit/3`, limits the number of inferences instead, and Section 31.9 uses it to bound a verifier.

```
?- call_with_depth_limit(blocked_by(101, B), 6,
                         R).
B = 103,
R = 4 ;
B = 104,
R = 5 ;
B = 103,
R = 6 ;
R = depth_limit_exceeded.
```

29.7 Counting Inferences

The length of a proof is a measure of how much a recognizer had to do to reach its conclusion, and an interpreter that threads a counter through the resolution computes it. `solve_count(Goal, N)` counts one for each goal resolved, built-ins included, along the proof that was found. The accumulator pattern of Chapter 5 carries the count; the count of a negated goal is one, for the negation itself, since the failed search beneath it is not part of the proof.

```
% solve_count(+Goal, -N): N is the number of goals
% resolved in the proof of Goal that was found.
solve_count(G, N) :-
    solve_count(G, 0, N).

solve_count(true, N, N).
solve_count((A, B), N0, N) :-
    solve_count(A, N0, N1),
    solve_count(B, N1, N).
solve_count(\+ A, N0, N) :-
    \+ solve_count(A, 0, _),
    N is N0 + 1.
solve_count(G, N0, N) :-
    \+ control(G),
    N1 is N0 + 1,
    (   system_goal(G)
    ->  call(G),
        N = N1
    ;   clause(G, Body),
        solve_count(Body, N1, N)
    ).

?- solve_count(blocked_thread(101), N).
N = 2 ;
false.

?- solve_count(deadlock(A, B), N).
A = 103,
B = 104,
N = 12 ;
false.

?- solve_count(handle_leak(T), N).
T = event,
N = 4 ;
false.

?- solve_count(spiking_thread(107), N).
N = 11 ;
false.
```

Twelve goals prove the deadlock: the goal itself, two instances of `blocks/2` with five goals each, and the final comparison. Eleven prove the **Spiking Thread** (page 335), and the count stops at the collecting goal, which this interpreter runs natively as one goal. This is the **size of the proof**, not the cost of finding it. The counter is a term that backtracking undoes, so the goals resolved on branches

that failed, the wait chains tried and abandoned before the pair `103`, `104` was found, leave no trace in `N`. The cost of the search is what SWI-Prolog's inference counter measures: `statistics/2` with the key `inferences` gives the number of procedure calls since the system started, and the difference across a goal is the number the goal cost.

```
?- deadlock(_, _),
   statistics(inferences, I0),
   deadlock(_, _),
   statistics(inferences, I1),
   Cost is I1 - I0.
I0 = <...>,
I1 = <...>,
Cost = 29.
```

The goal is run once before the measurement so that the second run pays nothing for one-time work such as building the clause index. Twenty-nine inferences for a proof of twelve goals is the price of the failed branches, and the ratio between the two numbers is a measure of how well the recognizer's clause order guides the search; Chapter 30 takes up the cost side.

29.8 A Tracing Meta-Interpreter

Chapter 4 described the four **ports** of the debugger's tracer, call, exit, redo, and fail, and showed its output as an abridged excerpt because the interactive tracer cannot be replayed. A meta-interpreter that prints the ports is a **tracing meta-interpreter** that can, and it is a few lines longer than `solve/1`. The goal is printed at its call port; if a step succeeds, at its exit port; when a later failure backtracks into it, at its redo port, after which the interpreter looks for the next solution; and when no solution remains, at its fail port. The disjunction in `trace_goal/2` is the mechanism: the first branch prints the exit and, on backtracking, the redo and then fails into `step/2` for another solution; the second branch is reached when `step/2` has none left.

```
% trace_solve(+Goal): solve Goal, printing the
% four ports of each goal, indented by depth.
trace_solve(G) :-
    trace_goal(G, 0).

trace_goal(true, _).
trace_goal((A, B), D) :-
    trace_goal(A, D),
    trace_goal(B, D).
trace_goal(\+ A, D) :-
    \+ trace_goal(A, D).
trace_goal(G, D) :-
    \+ control(G),
    port(call, D, G),
    (   step(G, D),
        (   port(exit, D, G)
        ;   port(redo, D, G),
            fail
        )
    ;   port(fail, D, G),
```

```
        fail
    ).

% step(+Goal, +Depth): one resolution step.
step(G, D) :-
    (   system_goal(G)
    ->  call(G)
    ;   clause(G, Body),
        D1 is D + 1,
        trace_goal(Body, D1)
    ).

port(Port, Depth, G) :-
    Indent is Depth * 2,
    \+ \+ ( numbervars(G, 0, _,
                       [singletons(true)]),
            format("~t~*|~a ~p~n",
                   [Indent, Port, G]) ).
```

```
?- trace_solve(handle_leak(file)).
call handle_leak(file)
  call handle_count(_,file,_)
  exit handle_count(4120,file,38)
  call threshold(handle_type_count,_)
  exit threshold(handle_type_count,5000)
  call 38>5000
  fail 38>5000
  redo threshold(handle_type_count,5000)
  fail threshold(handle_type_count,_)
  redo handle_count(4120,file,38)
  fail handle_count(_,file,_)
fail handle_leak(file)
false.
```

The trace reads as the debugger's would. The process holds 38 file handles, the comparison with the limit of 5000 fails, and the failure backtracks through the two facts before it, which have no alternatives, to the failure of the goal. An unbound argument is printed as _ at the call port and with its binding at the exit port, so the two lines of each goal together show what the step found out. The redo port is where the debugger's tracer is hardest to read and where this one helps most: a goal printed at its redo port is being asked for another solution, and the fail line that follows says it had none.

```
?- trace_solve(owner(A, 103)).
call owner(_,103)
  call critical_section(_,103,_,_)
  exit critical_section(0x7ff61d13a0c0,103,2,1)
  call 103=\=0
  exit 103=\=0
exit owner(0x7ff61d13a0c0,103)
A = 0x7ff61d13a0c0 ;
```

```
redo owner(0x7ff61d13a0c0,103)
  redo 103=\=0
  fail 103=\=0
  redo critical_section(0x7ff61d13a0c0,103,2,1)
  fail critical_section(_,103,_,_)
fail owner(_,103)
false.
```

Here the toplevel's request for a second answer is visible in the trace: after `A = 0x7ff61d13a0c0 ;` the redo ports fire from the innermost goal outwards, and the fail ports follow because the critical section facts have no second owner for thread 103. The goal at the fail port is printed with its arguments unbound again, because the bindings the step made have been undone, which is also what the debugger shows and what Chapter 3 said backtracking means.

The tracer shares the limitations of `solve/1` and adds one: it prints the goal at the redo port before it knows whether another solution exists, so a redo line is eventually followed by an exit or a fail line for the same goal, after whatever the goals beneath it print. Built-ins show two ports, since the interpreter does not see inside them. The tracer is a tool for understanding why a recognizer gives the answers it gives on a particular dump, and a recognizer that produces a long trace on a small dump is one whose clause order deserves attention.

29.9 Certainty Factors

Every answer so far has been a theorem: the facts are taken as true, the clauses as definitions, and a proof is a proof. Some diagnostic reasoning is not of this kind. Eight identical frames of `Node::Visit` on thread 107 make a **Stack Overflow (User Mode)** (page 366) a hypothesis worth checking and nothing more; the same frames with an exception record whose code is the stack overflow status `0xc00000fd` would make it nearly certain. Expert systems of the MYCIN tradition expressed such gradations with **certainty factors**, numbers between 0 and 1 attached to rules, and a meta-interpreter is the natural way to compute with them, because the rules are data and the combination is a matter of interpretation.

The rules are kept apart from the program, as facts `rule(Hypothesis, Condition, Weight)`, so that `clause/2` is not involved and the weights are visible. A **hypothesis** is a term `suspect(TID, What)` whose `What` is a plain label, not a pattern name; the conditions are recognizers and facts of the knowledge base. The combination follows the usual conventions: the certainty of a conjunction is the minimum of its parts; the certainty of a hypothesis derived by a rule is the certainty of its condition multiplied by the weight of the rule; a fact or a recognized pattern instance has certainty 1; and when several rules support the same hypothesis the largest result stands.

```
% frame_regularity(?TID): the top four frames
% repeat one function.
frame_regularity(TID) :-
    frame(TID, 0, M, F, _),
    frame(TID, 1, M, F, _),
    frame(TID, 2, M, F, _),
    frame(TID, 3, M, F, _).

% rule(Hypothesis, Condition, Weight)
rule(suspect(T, stack_overflow),
     frame_regularity(T), 0.3).
rule(suspect(T, stack_overflow),
     ( frame_regularity(T),
       exception_record(T, 0xc00000fd, _) ),
     0.95).
rule(suspect(T, infinite_loop),
     spiking_thread(T), 0.6).
rule(suspect(T, infinite_loop),
     ( spiking_thread(T),
       thread_state(T, running) ), 0.8).

% solve_cf(+Goal, -CF): Goal holds with
% certainty CF.
solve_cf(true, 1.0).
solve_cf((A, B), CF) :-
    solve_cf(A, CA),
    solve_cf(B, CB),
    CF is min(CA, CB).
solve_cf(G, CF) :-
    \+ control(G),
    (   \+ rule(G, _, _)
    ->  solve(G),
        CF = 1.0
    ;   rule(G, Condition, Weight),
        solve_cf(Condition, CC),
        CF is CC * Weight
    ).

% certainty(+Hypothesis, -CF): the best certainty
% any rule gives Hypothesis; fails if none does.
certainty(H, CF) :-
    aggregate_all(max(C), solve_cf(H, C), CF).

?- solve_cf(suspect(T, What), CF).
T = 107,
What = stack_overflow,
CF = 0.3 ;
T = 102,
What = infinite_loop,
CF = 0.6 ;
T = 107,
What = infinite_loop,
CF = 0.6 ;
T = 102,
What = infinite_loop,
CF = 0.8 ;
T = 107,
What = infinite_loop,
CF = 0.8 ;
```

```
false.

?- certainty(suspect(102, infinite_loop), CF).
CF = 0.8.

?- certainty(suspect(107, stack_overflow), CF).
CF = 0.3.

?- certainty(suspect(107, infinite_loop), CF).
CF = 0.8.

?- certainty(suspect(105, infinite_loop), CF).
false.
```

Thread 107 is suspected of a stack overflow at 0.3, because only the weak rule applies: the dump has no exception record for it, and the strong rule's condition fails. Threads 102 and 107 are suspected of an infinite loop at 0.8 by the rule that also looks at their state, since both are Spiking Threads under the threshold of Chapter 18 and both are running. Thread 107 therefore carries two hypotheses with different numbers, and the numbers order them: a report that lists the suspicions with the strongest first is a reasonable report, and for thread 107 it names the loop before the overflow.

The numbers must not be read as more than an ordering. They are not probabilities; nothing in their derivation makes 0.8 mean that eight of ten such threads loop forever. They are the rule author's judgment written as a decimal, and the combination rules, minimum for conjunction and product for the rule, are conventions chosen for their algebra rather than derived from any model of the evidence. Two rule bases written by two analysts will assign different numbers to the same dump and both will be internally consistent. The pattern catalogs this book formalizes do not assign weights to patterns, and this chapter does not propose that they should. A certainty factor is a device for ranking hypotheses inside one program, to be reported, if at all, as "stronger" and "weaker", and never as a percentage.

29.10 Meta-Interpreters and Analysis Pattern Networks

The author's work on **analysis pattern networks** describes diagnostic analysis as a layered process in which each layer enriches the artifact it receives rather than replacing it, so that what a later layer concludes remains traceable to what an earlier layer observed; the author calls this **provenance-preserving enrichment**. The proof tree of this chapter is a mechanism of that kind at the scale of a single query: the conclusion `deadlock(103, 104)` is kept together with the intermediate conclusions `blocks/2` and `owner/2` and the facts they rest on, and nothing along the way is discarded. The correspondence should be stated with care. A proof tree records the **provenance** of one answer under one recognizer; an analysis pattern network, as the author describes it, is an architecture for many patterns, artifacts, and analysts over time, and includes much that no meta-interpreter sees, such as the choice of which patterns to apply and the judgment that reconciles them. What the chapter supports is the narrower claim that provenance can be obtained from recognizers mechanically, as a by-product of running them, instead of being written by hand, and that the evidence form of Chapter 17 is one projection of it.

Chapter 30. Scale and Performance

The shared trace of this book has 71 messages and the shared dump ten threads, and every query in the preceding chapters answered in less time than it took to read the answer. A production log has millions of messages, a complete memory dump thousands of threads, and the same queries, written the same way, may take seconds, minutes, or not finish. This chapter is about the difference. It covers how SWI-Prolog finds clauses, which is the question of **indexing**; how to measure what a query costs, with `time/1` and `statistics/2`; which representation of a collection to choose when facts, association lists, and dicts are all possible; tabling as memoization; how to avoid materializing a large relation as a list; how to read a large file without holding it in memory; what the Prolog stacks are and how they are limited; and how to store a large knowledge base so that it loads in a fraction of the time it takes to consult.

The measurements are made on a synthetic trace of 100,000 messages generated in the chapter with `between/3` and `assertz/1`, and the benchmark is the **Discontinuity** (page 513) recognizer of Chapter 25 run over it. Every timing in the transcripts is written with the `<...>` wildcard, because a time depends on the machine, and the prose quotes the figures observed on one machine as orders of magnitude only. Inference counts, which do not depend on the machine, are quoted exactly where they make a point.

30.1 Measuring

`time/1` runs a goal and prints, to the standard error stream, the number of **inferences** it made, the CPU time and wall-clock time it took, and the rate in logical inferences per second (Lips). An inference, in SWI-Prolog's accounting, is a procedure call: each goal that is resolved against a predicate, built-in or user-defined, counts one. The count is the machine-independent measure of work, and the first thing to compare between two formulations of the same query.

The first example is the **Discontinuity** (page 513) recognizer of Chapter 25 over `trace1.pl` in two formulations. The first finds two messages and tests, with arithmetic comparison, that their numbers are consecutive; the second computes the number of the next message and looks it up.

```
:- [trace1].

% threshold(Pattern, Value)
threshold(discontinuity, 1.0).

% slow_discontinuity(?N, -Gap): pairs of messages,
% then a test. Quadratic in the trace length.
slow_discontinuity(N1, Gap) :-
    msg(N1, T1, _, _, _, _),
    msg(N2, T2, _, _, _, _),
    N2 =:= N1 + 1,
    Gap is T2 - T1,
    threshold(discontinuity, Min),
    Gap > Min.
```

```
% discontinuity(?N, -Gap): the next message is
% looked up by number. Linear in the trace length.
discontinuity(N1, Gap) :-
    msg(N1, T1, _, _, _, _),
    N2 is N1 + 1,
    msg(N2, T2, _, _, _, _),
    Gap is T2 - T1,
    threshold(discontinuity, Min),
    Gap > Min.

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- time(aggregate_all(count,
                      slow_discontinuity(_, _),
                      C)).
% <...> inferences, <...> CPU in <...>
C = 1.

?- time(aggregate_all(count, discontinuity(_, _),
                      C)).
% <...> inferences, <...> CPU in <...>
C = 1.

?- discontinuity(N, Gap).
N = 66,
Gap = 4.691000000000001 ;
false.
```

Both find the one gap of more than a second, between messages 66 and 67. The times printed are both a fraction of a millisecond and say nothing; the inference counts say everything. `statistics/2` with the key `inferences` reads the counter `time/1` uses, and the difference across a goal is the goal's cost; running the goal once before measuring removes the one-time cost of building indexes, which Section 30.2 explains.

```
?- aggregate_all(count, slow_discontinuity(_, _),
                 _),
   statistics(inferences, I0),
   aggregate_all(count, slow_discontinuity(_, _),
                 _),
   statistics(inferences, I1),
   Slow is I1 - I0.
I0 = <...>,
I1 = <...>,
Slow = 5331.

?- aggregate_all(count, discontinuity(_, _), _),
   statistics(inferences, I0),
   aggregate_all(count, discontinuity(_, _), _),
   statistics(inferences, I1),
   Fast is I1 - I0.
I0 = <...>,
I1 = <...>,
Fast = 290.
```

The slow formulation makes 71 × 71 pairs and tests each, about 5,000 inferences for 71 messages; the fast one makes one lookup per message. On the trace of this chapter, with 100,000 messages, the first would need ten billion inferences, which at the rate `time/1` reports, several million per second, is the better part of an hour, and the second needs a few hundred thousand. The difference between the two is not in the arithmetic but in what the second `msg/6` goal asks: with `N2` bound, SWI-Prolog goes to the one clause whose first argument is `N2`, and with `N2` unbound it tries all 71.

Other keys of `statistics/2` used in this chapter are `cputime`, the process CPU time in seconds as a float; `runtime`, a list of the CPU time in milliseconds and the time since the key was last read; `walltime`, the same for wall-clock time; and the memory keys of Section 30.9.

```
?- statistics(cputime, T), T > 0.
T = <...>.

?- statistics(runtime, [Total|_]).
Total = <...>.
```

30.2 Indexing

A procedure with ten thousand clauses is not searched clause by clause. SWI-Prolog maintains **clause indexes**, hash tables from the value of an argument to the clauses whose head has that value in that position, and uses one whenever a call has the indexed argument bound. **First-argument indexing** is the traditional guarantee of Prolog systems and is why the schema of Part II puts the message number first in `msg/6` and the thread number first in `frame/5`: a call with the first argument bound is a lookup, not a scan. SWI-Prolog goes further with **just-in-time indexing** (JIT): when a call has some other argument bound, or a combination of arguments, and the procedure is large enough for a scan to be costly, the system assesses the arguments, builds a hash table on the most selective one or on a combination, and keeps it for later calls. Indexes are created on demand, on the first call that would profit from them, and this includes the first argument.

`jiti_list/1` prints the indexes a predicate has at the moment. Its output is a table whose header and widths exceed the page, so the transcripts below use the `...` and `<...>` wildcards and the prose says what the columns hold.

```
:- [dump1].

?- jiti_list(frame/5).
...
true.

?- frame(107, 3, M, F, O).
M = reportengine,
F = 'Node::Visit',
O = 107.
```

```
?- jiti_list(frame/5).
...
user:frame/5<...>1+2<...>
true.

?- aggregate_all(count,
                 frame(_, _, reportengine, _, _),
                 C).
C = 13.

?- jiti_list(frame/5).
...
user:frame/5<...>1+2<...>
<...>3<...>
true.
```

Before any call, `frame/5` has no index and `jiti_list/1` prints only the header. The call `frame(107, 3, M, F, O)` binds the first two arguments; neither alone identifies a clause, since a thread has many frames and frame number 3 occurs in every thread, and the system builds a **multi-argument index** on the pair, which the table reports as `1+2`. The call with the module bound builds a second index on argument 3. For each index the table gives the number of hash buckets and an estimate of the speedup over trying every clause. `predicate_property/2` with `indexed(List)` gives the same information as a term.

Indexing has consequences for how a recognizer is written. A goal whose indexed argument is bound should come before goals that merely test; the fast `discontinuity/2` of Section 30.1 computes `N2` before the second `msg/6` goal for exactly this reason. Atoms, integers, and strings hash well; floats are hashed too, but calls that look up a message by its timestamp are rare, and the message number remains the key.

30.3 A Synthetic Trace

The trace of this chapter is generated rather than loaded, so that the chapter does not depend on a file of several megabytes. `gen_trace/1` asserts `N` messages into a dynamic predicate `smsg/6` with the argument order of `msg/6`: the message number, a timestamp that advances by one millisecond per message and jumps by two seconds at every thousandth message, the process, one of five threads, one of three components, and a text. A failure-driven loop over `between/3` with `assertz/1` is the simplest way to produce a large number of facts, and it is fast, since each `assertz/1` is a single operation on the clause store; the jumps give the **Discontinuity** (page 513) recognizer something to find.

```
:- dynamic smsg/6.
```

```
% gen_trace(+N): assert N synthetic messages.
gen_trace(N) :-
    retractall(smsg(_, _, _, _, _, _)),
    forall(between(1, N, I),
           ( Ms is I + 2000 * (I // 1000),
             T is Ms / 1000.0,
             TID is 100 + I mod 5,
             component(I, C),
             assertz(smsg(I, T, 4120, TID, C,
                          "Request")) )).

component(I, C) :-
    K is I mod 3,
    nth0(K, [appserver, dbclient, reportengine],
         C).

?- time(gen_trace(100000)).
% <...> inferences, <...> CPU in <...>
true.

?- aggregate_all(count, smsg(_, _, _, _, _, _),
                 C).
C = 100000.

?- smsg(999, T1, _, _, _, _),
   smsg(1000, T2, _, _, _, _).
T1 = 0.999,
T2 = 3.0.

?- jiti_list(smsg/6).
...
user:smsg/6<...>1<...>
true.
```

Generating 100,000 messages took about a quarter of a second on the author's machine, ten inferences per message. The lookups by number created the first-argument index, with a bucket count that `jiti_list/1` shows and a speedup equal to the number of clauses, since a lookup by number tries one clause instead of 100,000. A call with the thread bound and the number unbound is a different matter.

```
?- time(aggregate_all(count,
                      smsg(_, _, _, 104, _, _),
                      C)).
% <...> inferences, <...> CPU in <...>
C = 20000.

?- time(aggregate_all(count,
                      smsg(_, _, _, 104, _, _),
                      C)).
% <...> inferences, <...> CPU in <...>
C = 20000.
```

```
?- jiti_list(smsg/6).
...
user:smsg/6<...>1<...>
<...>4<...>
true.
```

The first call paid for building an index on argument 4; the second ran several times faster and found the 20,000 messages of thread 104 without touching the other 80,000. A trace with five threads gives a speedup of five on this argument, which `jiti_list/1` reports; a trace with a thousand threads would give a thousand. The **Thread of Activity** (page 537) of Chapter 26, which is exactly this call, is therefore fast on a large trace after its first use, and nothing in the recognizer had to change.

30.4 Discontinuity at Scale

The fast formulation of Section 30.1 is now run over `smsg/6`. The recognizer is the same up to the name of the message predicate.

```
% big_discontinuity(?N, -Gap): Discontinuity over
% the synthetic trace.
big_discontinuity(N1, Gap) :-
    smsg(N1, T1, _, _, _, _),
    N2 is N1 + 1,
    smsg(N2, T2, _, _, _, _),
    Gap is T2 - T1,
    threshold(discontinuity, Min),
    Gap > Min.

?- time(aggregate_all(count,
                      big_discontinuity(_, _),
                      C)).
% <...> inferences, <...> CPU in <...>
C = 100.

?- big_discontinuity(N, Gap), N > 50000,
   format("~d ~3f~n", [N, Gap]).
50999 2.001
N = 50999,
Gap = 2.0010000000000048 ;
51999 2.001
N = 51999,
Gap = 2.0010000000000048 ;
...
```

```
?- aggregate_all(count, big_discontinuity(_, _),
                 _),
   statistics(inferences, I0),
   aggregate_all(count, big_discontinuity(_, _),
                 _),
   statistics(inferences, I1),
   Cost is I1 - I0.
I0 = <...>,
I1 = <...>,
Cost = 400204.
```

A hundred discontinuities, one at every thousandth message, found in about a tenth of a second and four inferences per message: the first `smsg/6` goal enumerates, the arithmetic computes the next number, the second goal is a lookup, and the comparison tests. This is the linear cost that first-argument indexing buys, and the second query shows the recognizer producing answers one at a time, with the toplevel asked for more; the gap prints as `2.001` with `~3f` and as `2.0010000000000048` as a float, which is the arithmetic of Chapter 6 and not an error in the trace. The quadratic formulation is not run on this trace, for the reason given in Section 30.1.

30.5 Facts, Association Lists, and Dicts

A collection that a program builds and looks up has three natural representations in SWI-Prolog, and the choice is a matter of scale and of lifetime. **Facts** in the dynamic database are global, indexed, and persist across queries; they are the representation of the artifacts themselves and of anything derived from them that several queries will consult. **Association lists** from `library(assoc)` of Chapter 10 are AVL trees: a term, built and looked up in logarithmic time, that lives in a clause body and is discarded with it, and that backtracking undoes. **Dicts** are SWI-Prolog's record type: a flat array sorted by key, looked up by binary search, and copied whole by `put_dict/4`.

```
% lookup_facts(+N): N lookups by message number.
lookup_facts(N) :-
    forall(between(1, N, I),
           smsg(I, _, _, _, _, _)).

% lookup_assoc(+N): build an assoc of N pairs and
% look each up.
lookup_assoc(N) :-
    findall(I-I, between(1, N, I), Pairs),
    list_to_assoc(Pairs, Assoc),
    forall(between(1, N, I),
           get_assoc(I, Assoc, _)).

% lookup_dict(+N): the same with a dict built in
% one step.
lookup_dict(N) :-
    findall(I-I, between(1, N, I), Pairs),
    dict_pairs(Dict, t, Pairs),
    forall(between(1, N, I),
           get_dict(I, Dict, _)).
```

```
% grow_assoc(+N), grow_dict(+N): build one element
% at a time.
grow_assoc(N) :-
    numlist(1, N, Keys),
    empty_assoc(A0),
    foldl([K, A1, A2]>>put_assoc(K, A1, K, A2),
          Keys, A0, _).

grow_dict(N) :-
    numlist(1, N, Keys),
    foldl([K, D1, D2]>>put_dict(K, D1, K, D2),
          Keys, _{}, _).

?- time(lookup_facts(100000)).
% <...> inferences, <...> CPU in <...>
true.

?- time(lookup_assoc(100000)).
% <...> inferences, <...> CPU in <...>
true.

?- time(lookup_dict(100000)).
% <...> inferences, <...> CPU in <...>
true.

?- time(grow_assoc(10000)).
% <...> inferences, <...> CPU in <...>
true.

?- time(grow_dict(10000)).
% <...> inferences, <...> CPU in <...>
true.

?- time(grow_dict(20000)).
% <...> inferences, <...> CPU in <...>
true.
```

On the author’s machine, 100,000 lookups in the fact base took a few hundredths of a second, two inferences each; the dict built in one step from a pair list and looked up 100,000 times took about the same; the association list, built and looked up, took three to four times longer, a million inferences, since each `get_assoc/3` walks about seventeen levels of a tree. The last three queries show the trap. Growing an association list by 10,000 insertions costs tenths of a second and grows as the logarithm says it should; growing a dict by 10,000 insertions costs more than half a second, and doubling the size quadruples the time, because every `put_dict/4` copies the whole array. A dict is a record with a handful of fields, a message as a term with named parts, and not a table; a table built incrementally is an association list, and a table that outlives the query is a set of facts. The figures will differ on other machines, and the relationships between them will not.

30.6 Distinct Values and library(nb_set)

The distinct threads of a trace are `setof/3` over the messages, which collects every solution, sorts, and removes duplicates: linear in the trace for the collection and logarithmic-linear for the sort.

`library(nb_set)` offers the alternative of a set that is updated destructively as the solutions are produced. `empty_nb_set/1` creates it, `add_nb_set/2` adds an element, `add_nb_set/3` adds and reports in its third argument whether the element was new, and `nb_set_to_list/2` reads the set as a sorted list. The prefix `nb` is for non-backtrackable, as in `nb_setval/2` of Chapter 11: the additions survive the failure of the loop that makes them.

```
:- use_module(library(nb_set)).

% threads_nb(-TIDs): distinct threads via nb_set.
threads_nb(TIDs) :-
    empty_nb_set(Set),
    forall(smsg(_, _, _, TID, _, _),
           add_nb_set(TID, Set)),
    nb_set_to_list(Set, TIDs).

% threads_setof(-TIDs): the same via setof/3.
threads_setof(TIDs) :-
    setof(TID, N^T^P^C^X^smsg(N, T, P, TID, C, X),
          TIDs).

% first_message_of_each(-Ns): the number of the
% first message of every thread, in one pass.
first_message_of_each(Ns) :-
    empty_nb_set(Seen),
    findall(N, ( smsg(N, _, _, TID, _, _),
                 add_nb_set(TID, Seen, true) ),
            Ns).

?- time(threads_nb(Ts)).
% <...> inferences, <...> CPU in <...>
Ts = [100, 101, 102, 103, 104].

?- time(threads_setof(Ts)).
% <...> inferences, <...> CPU in <...>
Ts = [100, 101, 102, 103, 104].

?- first_message_of_each(Ns).
Ns = [1, 2, 3, 4, 5].
```

The measurement is not the advertisement one might expect: `setof/3` was faster, by a factor of about three, since its collection is a single built-in pass and its sort is done in C, while `add_nb_set/2` is a Prolog predicate hashing each of 100,000 elements. The use of `library(nb_set)` is not speed. It is the third predicate: a question that `setof/3` cannot ask, since it needs to know, while enumerating, whether a value has been seen, and a set that grows under a `findall/3` is the way to ask it. The same device serves when the solutions come from a file that is read once, as in Section 30.8, and cannot be enumerated twice.

30.7 Tabling as Memoization

Tabling, introduced in Chapter 11 to make left-recursive closures terminate, has a second use: a tabled predicate computes each answer once and serves it from the table afterwards. A recognizer

that is expensive and is asked the same question repeatedly, by a report that mentions it in several places or by a composite pattern that uses it as a goal, profits from `:- table` as from a cache.

```
:- table thread_messages/2.

% thread_messages(+TID, -Count): messages of TID.
thread_messages(TID, Count) :-
    aggregate_all(count, smsg(_, _, _, TID, _, _),
                  Count).

?- time(thread_messages(102, C)).
% <...> inferences, <...> CPU in <...>
C = 20000.

?- time(thread_messages(102, C)).
% <...> inferences, <...> CPU in <...>
C = 20000.

?- abolish_all_tables.
true.
```

The first call made 40,000 inferences and the second a handful: the answer came from the table. The table is keyed on the call, so `thread_messages(103, C)` computes again, and a call with the thread unbound computes for all threads and tables each. The cost is the caution. A table is a copy of the answers at the time of the call, and if the knowledge base changes afterwards, by asserting more messages or loading a second trace, the table is stale and the predicate answers the old question. `abolish_all_tables/0` discards every table and is the simplest remedy after a change; SWI-Prolog's **incremental tabling**, declared with `:- table p/2 as incremental` together with `:- dynamic q/6 as incremental` for the facts it depends on, keeps tables consistent with the dynamic predicates automatically at the price of bookkeeping on every change, and is the right tool when the knowledge base is updated while recognizers run. Memoization pays when the same call recurs; a recognizer called once per analysis gains nothing from a table and pays for building it.

30.8 Collecting Less

`findall/3` copies every solution into a list, and Chapter 9 said so; on a relation of 100,000 messages the copy is 100,000 terms on the global stack. Much of what a report needs from a large relation is not the list but a number, and `aggregate_all/3` computes the number without the list. Where a list is unavoidable, `limit/2` of Chapter 9 bounds its length, and where the solutions are consumed one by one, a failure-driven loop or `forall/2` consumes them without storing any.

```
?- time(( findall(T, smsg(_, T, _, _, _, _), _Ts),
          length(_Ts, N) )).
% <...> inferences, <...> CPU in <...>
N = 100000.
```

```
?- time(aggregate_all(count,
                      smsg(_, _, _, _, _, _), N)).
% <...> inferences, <...> CPU in <...>
N = 100000.

?- time(aggregate_all(max(T),
                      smsg(_, T, _, _, _, _),
                      Last)).
% <...> inferences, <...> CPU in <...>
Last = 300.0.

?- limit(3, smsg(N, _, _, 103, _, _)).
N = 3 ;
N = 8 ;
N = 13.
```

The two counts cost about the same number of inferences, since both visit every message; the difference is the 100,000-element list the first builds and the garbage collector later reclaims. The maximum of the timestamps, which the **Last Activity** (page 532) of Chapter 25 needs, is one pass with no list at all. The principle is to ask for the aggregate when an aggregate is wanted, and to ask for the list only when the list itself is the result.

30.9 Streaming from Files

A trace file of a hundred megabytes should not be read into a list of codes before parsing begins. `phrase_from_file/2` from `library(pure_input)`, introduced in Chapter 12, parses a file with a DCG while reading it in blocks: the list the grammar walks is a **lazy list**, whose tail is an attributed variable that is bound to the next block of the file when the grammar reaches it, and whose head is released to the garbage collector when nothing refers to it. A grammar that carries its state in arguments, instead of building a list of everything it saw, therefore runs in memory bounded by the block size. The log for the demonstration is written from the synthetic trace in the format of Chapter 12, one message per line.

```
:- use_module(library(dcg/basics)).
:- use_module(library(pure_input)).

% write_log(+File): the synthetic trace as text.
write_log(File) :-
    setup_call_cleanup(
        open(File, write, S),
        forall(smsg(N, T, P, TID, C, Text),
               format(S,
                      "~d ~3f [~d:~d] ~a: ~s~n",
                      [N, T, P, TID, C, Text])),
        close(S)).

% log_line(-N, -Time)//: one line; the rest of the
% line is skipped.
log_line(N, Time) -->
    integer(N), " ", number(Time), " ",
    string_without("\n", _), "\n".
```

```
% gaps(+Prev, +C0, -C)//: count the gaps of more
% than a second between consecutive lines.
gaps(Prev, C0, C) -->
    log_line(_, T),
    { (   Prev = time(T0), T - T0 > 1.0
      ->  C1 is C0 + 1
      ;   C1 = C0
      ) },
    gaps(time(T), C1, C).
gaps(_, C, C) -->
    eos.

count_gaps(File, Count) :-
    once(phrase_from_file(gaps(none, 0, Count),
                          File)).

?- time(write_log('big.log')).
% <...> inferences, <...> CPU in <...>
true.

?- time(count_gaps('big.log', Count)).
% <...> inferences, <...> CPU in <...>
Count = 100.
```

The grammar found the hundred discontinuities in the text without a `smsg/6` fact in sight, in a few seconds on the author's machine. That is slower than the same count over facts by more than an order of magnitude, and the comparison is fair: the lazy list is built of individual character codes, and `library(dcg/basics)` is Prolog, not C. For a line-oriented format, the stream predicates of Chapter 13 are faster. `read_line_to_string/2` reads a line as a string in C, and a loop over it with `split_string/4` or a DCG applied per line with `string_codes/2` keeps the purity of the grammar where it matters, in the line, and leaves the file handling to the stream.

```
% count_lines(+File, -N): lines of File.
count_lines(File, N) :-
    setup_call_cleanup(
        open(File, read, S),
        count_lines(S, 0, N),
        close(S)).

count_lines(S, N0, N) :-
    read_line_to_string(S, Line),
    (   Line == end_of_file
    ->  N = N0
    ;   N1 is N0 + 1,
        count_lines(S, N1, N)
    ).

?- time(count_lines('big.log', N)).
% <...> inferences, <...> CPU in <...>
N = 100000.
```

A fact file is read the same way, one term at a time. `read_term/2` reads the next clause from a stream as a term and gives the atom `end_of_file` at the end, and a loop over it is the way to process a file of facts that is too large to consult, or to filter it into a smaller one. Section 30.11 writes such a file.

`library(fastrw)` provides `fast_write/2` and `fast_read/2`, which write and read terms in a binary format that is faster to read than text and is not meant to be edited; the stream must be opened with `type(binary)`.

```
?- open('t.fast', write, _S, [type(binary)]),
   fast_write(_S, smsg(1, 0.001, 4120, 101, a,
                       "x")),
   close(_S).
true.

?- open('t.fast', read, _S, [type(binary)]),
   fast_read(_S, Term),
   close(_S).
Term = smsg(1, 0.001, 4120, 101, a, "x").
```

30.10 Stacks and Garbage Collection

A running Prolog program uses three stacks. The **global stack** holds compound terms and strings; the **local stack** holds the environments of the clauses being executed and the choice points; the **trail** records the bindings that backtracking must undo. SWI-Prolog allocates them on demand and limits their combined size by the flag `stack_limit`, whose default in this edition is one gigabyte on 64-bit systems. A program that exceeds it receives the exception `error(resource_error(stack), Context)`, where `Context` is a dict describing the stacks at the moment of the overflow. The limit is a safeguard against runaway recursion, and it can be raised when a legitimate computation needs more, by the flag or by the command-line option `--stack-limit=4g`.

```
?- current_prolog_flag(stack_limit, Limit).
Limit = 1073741824.

?- set_prolog_flag(stack_limit, 1 000 000),
   catch(length(_L, 100 000 000),
         error(resource_error(R), _), true).
R = stack.

?- set_prolog_flag(stack_limit, 1 073 741 824).
true.
```

The list of a hundred million elements does not fit in a megabyte of stack, and the error names the resource. `statistics/2` reports the stacks: `globalused`, `localused`, and `trailused` give the bytes in use on each, `stack` their total, and `garbage_collection` a list that begins with the number of collections, the bytes they reclaimed, and the time they took. `garbage_collect/0` runs a collection at once, which is useful before a measurement.

```
?- garbage_collect, statistics(globalused, G),
   G < 1 000 000.
G = <...>.

?- statistics(garbage_collection, [Count|_]),
   Count > 0.
Count = <...>.
```

Facts are not on these stacks. The 100,000 messages of `smsg/6` live in the clause store, which is heap memory outside the limit, and `statistics(clauses, N)` counts them with every other clause in the system. The stacks are used by the queries over the facts, and a `findall/3` of 100,000 timestamps is what moves the global stack, not the facts themselves. This is one more reason the artifact is facts and the analysis is queries: a knowledge base of millions of messages is a matter of memory, and a query over it is a matter of stack only to the extent that it collects.

30.11 Fact Files and Quick-Load Files

A knowledge base generated by a parser, or by `gen_trace/1`, is kept between sessions as a file of facts. `portray_clause/2` writes a term to a stream as a clause, quoted and terminated with a full stop, in the layout `listing/1` uses, and a loop over the facts writes the file.

```
% write_facts(+File): the synthetic trace as facts
% of tmsg/6, so that loading it does not redefine
% the dynamic smsg/6.
write_facts(File) :-
    setup_call_cleanup(
        open(File, write, S),
        forall(smsg(N, T, P, TID, C, Text),
               portray_clause(S,
                   tmsg(N, T, P, TID, C, Text))),
        close(S)).

% count_terms(+File, -N): clauses in a fact file,
% read one at a time.
count_terms(File, N) :-
    setup_call_cleanup(
        open(File, read, S),
        count_terms(S, 0, N),
        close(S)).

count_terms(S, N0, N) :-
    read_term(S, Term, []),
    (   Term == end_of_file
    ->  N = N0
    ;   N1 is N0 + 1,
        count_terms(S, N1, N)
    ).

?- time(write_facts('big.pl')).
% <...> inferences, <...> CPU in <...>
true.

?- time(count_terms('big.pl', N)).
% <...> inferences, <...> CPU in <...>
N = 100000.

?- time(consult(big)).
% <...> inferences, <...> CPU in <...>
true.
```

```
?- tmsg(1000, T, _, _, _, _).
T = 3.0.
```

Writing the file with `portray_clause/2` took longer than generating the facts, about a second and a half, since the predicate lays out each clause with care; `format/3` with `~q` and a full stop is about three times faster when the layout does not matter. Consulting the file took about a second and a half as well: the reader parses each clause, the compiler translates it, and the clause is added to the store. For a knowledge base that is loaded many times, SWI-Prolog's **quick-load files** save the compilation. `qcompile/1` compiles a source file and writes the result next to it with the extension `.qlf`; `consult/1` of the `.qlf` file, or `load_files/2` with the option `qcompile(auto)`, which compiles when the `.qlf` is missing or older than the source and loads it otherwise, restores the clauses without parsing.

```
?- time(qcompile(big)).
% <...> inferences, <...> CPU in <...>
true.

?- exists_file('big.qlf').
true.

?- time(consult('big.qlf')).
% <...> inferences, <...> CPU in <...>
true.

?- time(load_files(big, [qcompile(auto)])).
% <...> inferences, <...> CPU in <...>
true.

?- aggregate_all(count, tmsg(_, _, _, _, _, _),
                 N).
N = 100000.
```

Loading the quick-load file took a few hundredths of a second, some thirty times less than consulting the source, and a few hundred inferences in place of ten million. The format is specific to the SWI-Prolog version that wrote it and is not portable across versions, so the source is kept and the `.qlf` is treated as a cache, which `qcompile(auto)` makes automatic. A fact file of several hundred megabytes loads in seconds this way, and a workflow in which the debugger log is parsed once, written as facts, and compiled once gives the analyst a knowledge base that is ready before the debugger would have finished printing.

30.12 Profiling

When a query is slow and the inference count does not say where, `profile/1` runs the goal under SWI-Prolog's statistical profiler and shows where the time went. The profiler samples the executing predicate at a fixed interval and attributes each sample to the predicate running and to the chain of predicates that called it; at the end it presents, per predicate, the number of calls and redos and the time spent in the predicate itself and inclusive of what it called, sorted by the latter. With the graphical environment available the presentation is a window; without it, `show_profile/1` prints a

table of the same figures to the terminal. The output is not shown in a transcript because every column of it depends on the machine and the moment. What to look for does not. A predicate with a high own time and many calls is a candidate for indexing, as the `msg/6` goal of the slow **Discontinuity** (page 513) would be; a predicate with a high inclusive time and few calls is doing too much per call, often a `findall/3` where an aggregate would serve; and a predicate whose redo count is close to its call count is being backtracked into, which the tracer of Chapter 29 shows in detail on a small example. Profiling confirms a hypothesis about where the cost is. The inference counts and the indexing table of this chapter are what suggest it.

Chapter 31. Integration: Scripts, Embedding, and Debugger Extensions

The preceding chapters ran at the toplevel, which is where an analyst explores. The recognizers, once written, are more useful where the artifacts are: in a script that a build system calls on every crash dump, in a service that other tools query, inside the debugger that produced the dump, and behind an agent that proposes hypotheses faster than a human can check them. This chapter covers the ways a SWI-Prolog knowledge base is connected to its surroundings. Command-line tools are written with `library(main)`; data is exchanged as JSON and CSV; a service is built with the HTTP libraries; a C program hosts the Prolog engine through the embedding interface; and the Machine Query Interface connects a Python process to a running Prolog. A debugger extension that exposes a dump to Prolog while the debugger is still attached to it is treated twice: Section 31.6 is the design a reader of the embedding interface draws first, with the engine inside the debugger and the dump exposed on demand, and Section 31.7 is the extension the author built, PrologWinDbg, which takes the other side of two of that design's decisions; the two are compared, point by point, with what each buys and what it costs. The last section is a design: the use of a Prolog knowledge base as the verifier of hypotheses proposed by a language-model agent, with what such verification can and cannot establish.

Two kinds of block appear in this chapter, and they are labeled where they occur. The executable Prolog examples and their transcripts were run, as in every other chapter, in a toplevel whose working directory holds the shared files. The HTTP service of Section 31.4 and the C of Section 31.5 are **illustrative**: the Prolog of the service is checked for syntax and not run, since a server has no transcript, and the C examples were not compiled or executed. The debugger extension of Section 31.6 is a design, with one illustrative Prolog block. Section 31.7 describes an extension that exists, as its repository documents it: the extension's own Prolog module is reproduced in the book's package and is executed here, against a synthetic snapshot, with transcripts; the debugger session shown there is reproduced from the repository's validation log and was not run in the environment of this book, which has no Windows debugger. The Machine Query Interface of Section 31.8 is described in prose with one executed transcript.

31.1 Command-Line Tools

A Prolog file becomes a program with `library(main)`, introduced in Chapter 13. The directive `:- initialization(main, main).` declares that `main/0` is the entry point when the file is run as a script, and `library(main)` defines `main/0` to collect the command-line arguments and call the user's `main/1` with them as a list of atoms, halting with exit code 0 when `main/1` succeeds and a nonzero code when it fails or raises an exception. `argv_options/3` parses the list. In this edition it has two modes. Without further declarations it recognizes long options only: `--name=value` becomes `name(value)`, with numbers converted, `--name` becomes `name(true)`, and `--no-name` becomes `name(false)`; everything else is positional. When the program defines `opt_type/3`, parsing is guided by it: each clause `opt_type(Opt, Name, Type)` declares an option, a one-character `Opt` is a short option and a longer one a long option, and `Type` is checked and converted, `float`, `integer`, `boolean`,

`atom`, `file`, and `oneof(List)` among others. `opt_help/2` supplies the text that `argv_usage/1` prints for `--help`, which the guided mode handles without further code.

The tool below finds the **Time Delta** (page 511) anomalies of a trace, the gaps between consecutive messages above a threshold, which is the **Discontinuity** (page 513) of Chapter 25 with the threshold as a command-line option. The trace file is a positional argument and is consulted at run time.

```
:- use_module(library(main)).
:- set_prolog_gc_thread(false).
:- initialization(main, main).

% opt_type(Opt, Name, Type)
opt_type(threshold, threshold, float).
opt_type(t, threshold, float).
opt_type(csv, csv, boolean).

opt_help(threshold, "Minimum gap in seconds").
opt_help(csv, "Comma-separated output").
opt_help(help(usage), " -- trace.pl [options]").

main(Argv) :-
    argv_options(Argv, [File], Options),
    option(threshold(Min), Options, 1.0),
    option(csv(CSV), Options, false),
    consult(File),
    forall(( gap(N1, N2, Gap), Gap > Min ),
           report(CSV, N1, N2, Gap)).

% gap(?N1, -N2, -Gap): time between consecutive
% messages.
gap(N1, N2, Gap) :-
    msg(N1, T1, _, _, _, _),
    N2 is N1 + 1,
    msg(N2, T2, _, _, _, _),
    Gap is T2 - T1.

report(false, N1, N2, Gap) :-
    format("~d -> ~d: ~3f s~n", [N1, N2, Gap]).
report(true, N1, N2, Gap) :-
    format("~d,~d,~3f~n", [N1, N2, Gap]).
```

The second directive concerns the collector. SWI-Prolog of this edition reclaims atoms and clauses in a thread of its own, named `gc`, and a script that halts a few milliseconds after it started can halt while that thread is in the middle of a collection; `halt/0` then waits briefly for it and prints `% The following threads wouldn't die: [gc]` on the error stream, harmlessly, in about one run in sixty on the machine this book was drafted on. `set_prolog_gc_thread(false)` disables the dedicated collector, stops it if it is already running, which it may be by the time the libraries above are loaded, and sets the flag `gc_thread` to `false`; the collections that follow run in the thread that requests them, the main thread in these scripts. Setting the flag alone, with `set_prolog_flag/2`, changes the scheduling of later collections but leaves a collector that has already started alive, which is why the predicate is used. The setting addresses the shutdown message of this edition and nothing more: it changes when and where collection runs for the rest of the process, it remains in effect when the

file is consulted into a toplevel, as the harness does, and it is not a guarantee of identical output or of unchanged performance for an arbitrary program. The `deltas.pl` of Chapter 13 sets it for the same reason. Consulting the file into the toplevel, as the harness does, defines its predicates and does not run `main/0`, since the `main` initialization fires only when the file is loaded as a script. The parsing can therefore be tried on a list directly, and `shell/1` runs the script from the toplevel as a separate process whose output appears in the transcript.

```
?- argv_options(['trace1.pl', '--threshold=0.5'],
                Positional, Options).
Positional = ['trace1.pl'],
Options = [threshold(0.5)].

?- argv_options(['trace1.pl', '-t', '0.5',
                 '--no-csv'],
                Positional, Options).
Positional = ['trace1.pl'],
Options = [threshold(0.5), csv(false)].

?- shell('swipl gaps.pl -- trace1.pl -t 0.5').
10 -> 11: 0.786 s
38 -> 39: 0.998 s
40 -> 41: 0.999 s
43 -> 44: 0.998 s
45 -> 46: 0.999 s
66 -> 67: 4.691 s
true.

?- shell('swipl gaps.pl -- trace1.pl -t 2 --csv').
66,67,4.691
true.
```

The `--` in the command line matters. In this edition, `swipl` consults every leading argument that names a Prolog source file, so `swipl gaps.pl trace1.pl` would load the trace as a second program file and leave `main/1` with no positional argument; after `--`, the arguments belong to the program. A script that begins with the line `#!/usr/bin/env swipl` and is made executable runs as `./gaps.pl -- trace1.pl`, with the same proviso. The short option `-t 2` is converted to the float `2.0` because its declared type is `float`, and a value that does not convert is an error that the guided mode reports itself, with exit code 1, which `shell/2` returns.

```
?- shell('swipl gaps.pl -- trace1.pl -t abc',
         Status).
ERROR: Option -t requires an argument<...>
Status = 1.

?- shell('stty cols 80; swipl gaps.pl --help').
Usage: swipl gaps.pl -- trace.pl [options]

Options:
-h, -?, --help<...>
-t FLOAT, --threshold=FLOAT Minimum gap in seconds
--csv                       Comma-separated output
true.
```

The usage text is generated from the `opt_type/3` and `opt_help/2` clauses, and `-h`, `-?`, and `--help` are bound to it by default; `library(main)` wraps the text to the width of the terminal, which the `stty` command sets for the harness's terminal before the script runs. Exit codes are the interface between a Prolog tool and the shell script or build job that calls it: `main/1` that fails exits with 1, an uncaught exception exits with 2 after printing the error, and `halt/1` with an explicit code is the way to signal, for instance, that a pattern was recognized, so that a job can fail a build when a dump shows a **Deadlock (Critical Sections)** (page 388).

31.2 JSON

JSON is the interchange format of the tools a diagnostic service talks to, and `library(http/json)` reads and writes it. The library has two representations of a JSON object; the one used here, and recommended for new code, is the SWI-Prolog dict of Chapter 10. `json_write_dict/2` and `json_write_dict/3` write a term to a stream as JSON: a dict is an object, a list is an array, a number is a number, a string or an atom is a string, and the atoms `true`, `false`, and `null` are the JSON literals. `json_read_dict/2` reads one JSON value from a stream, producing dicts whose keys are atoms and whose strings are SWI-Prolog strings. `atom_json_dict/3` converts between an atom holding JSON text and a dict in both directions.

```
:- use_module(library(http/json)).
:- [dump1].

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- json_write_dict(current_output,
       _{pattern:"Spiking Thread",
         threads:[102]}).
{"pattern":"Spiking Thread", "threads": [102 ]}
true.

?- json_write_dict(current_output,
       _{pattern:"Spiking Thread", threads:[102],
         exception:null, verified:true},
       [width(20)]).
{
  "exception":null,
  "pattern":"Spiking Thread",
  "threads": [102 ],
  "verified":true
}
true.

?- atom_json_dict('{"hypothesis": "deadlock(103)",
                   "id": 7}', D, []).
D = _{hypothesis:"deadlock(103)", id:7}.

?- atom_json_dict('{"id": 7}', _D, []),
   get_dict(id, _D, Id).
Id = 7.
```

The `width(W)` option sets the line width at which the writer switches from the horizontal to the vertical layout; the default is 72, and `width(0)` forces a single line, which is what a program on the other end of a pipe wants. The keys of a dict are written in the standard order of terms, not in the order they were given, which JSON permits. Prolog terms that are not JSON values, such as the goal `deadlock(critical_section, C)` or an address written as a hexadecimal integer, are carried as strings: `term_string/2` of Chapter 8 converts in both directions, and Section 31.9 uses it to receive hypotheses as text and send evidence back as text. A file is written and read back with the stream forms.

```
?- setup_call_cleanup(
       open('finding.json', write, _S),
       json_write_dict(_S, _{tid:109,
                             code:0xc0000005}),
       close(_S)).
true.

?- setup_call_cleanup(
       open('finding.json', read, _S),
       json_read_dict(_S, D),
       close(_S)).
D = _{code:3221225477, tid:109}.
```

The exception code `0xc0000005` is the integer 3221225477, and JSON has no hexadecimal notation; a consumer that expects the hexadecimal text receives it as a string produced with `format(string(S), "0x~16r", [Code])`.

31.3 CSV

Comma-separated values are how traces leave logging systems and spreadsheets, and `library(csv)` reads and writes them. `csv_read_file/2` and `csv_read_file/3` read a whole file into a list of terms `row(F1, F2, ...)`, one per line, with the arity of each row the number of fields on its line; fields that look like numbers are converted to numbers by default and the others are atoms. `csv_read_file_row/3` reads one row at a time on backtracking, which is the form for a file too large to hold as a list, as Chapter 30 argued; its option `line(L)` reports the line number. `csv_write_file/2` and `csv_write_file/3` write a list of rows. Options include `separator(Code)` for a tab-separated file, `functor(Name)` for rows with another principal functor, `convert(false)` to keep every field as an atom, and `skip_header(Char)` to ignore comment lines.

```
:- use_module(library(csv)).

% thread_rows(-Rows): CPU times as CSV rows with a
% header row.
thread_rows([Header|Rows]) :-
    Header = row(tid, user_ms, kernel_ms),
    findall(row(T, U, K), thread_time(T, U, K),
            Rows).
```

```
?- thread_rows(_Rows),
   csv_write_file('threads.csv', _Rows).
true.

?- shell('head -3 threads.csv').
tid,user_ms,kernel_ms
100,120,40
101,15,5
true.

?- csv_read_file('threads.csv', _Rows),
   length(_Rows, N),
   _Rows = [Header|_].
N = 11,
Header = row(tid, user_ms, kernel_ms).

?- csv_read_file_row('threads.csv', row(T, U, K),
                     [line(L)]),
   integer(U),
   U + K > 5000.
T = 102,
U = 8450,
K = 1230,
L = 4 ;
false.
```

The header row reads as a row of atoms, and a program that converts rows to facts skips it or tests the type of a field as the last query does. Converting rows to the schema of Part II is one clause per row shape: a `row/6` from a logging system with the columns of `msg/6` becomes a fact with `assertz/1`, or is written to a fact file with `portray_clause/2` as in Chapter 30, and the message text, which `library(csv)` reads as an atom, is converted with `atom_string/2` if the schema wants a string. The grammar `csv//1` of the same library parses CSV text held in a code list, for the case where the data arrives in a message rather than a file.

31.4 An HTTP Service

A knowledge base that other programs query over a socket is a small HTTP service, and SWI-Prolog's HTTP libraries make one in a few clauses. `library(http/http_server)` bundles the pieces: `http_server/1` starts a multi-threaded server on a port, `http_handler/3` registers a predicate for a path, and with `library(http/http_json)` the handler reads its request body as a dict with `http_read_json_dict/2` and answers with `reply_json_dict/1`. The older `library(http/thread_httpd)` with `http_dispatch/1` is the same machinery assembled by hand. The block below is illustrative: it is checked for syntax and was not run, since a server that waits for requests has no transcript. It assumes the verifier `check_hypothesis/2` and the conversion `verdict_dict/3` of Section 31.9, and the module `patterns_dump` that the verifier reads.

```
:- use_module(library(http/http_server)).
:- use_module(library(http/http_json)).
:- [dump1].
:- use_module(patterns_dump).
```

```
% POST /check with a request body such as
% {"hypothesis": "deadlock(critical_section, C)"}
:- http_handler(root(check), check_handler,
                [method(post)]).

check_handler(Request) :-
    http_read_json_dict(Request, In),
    get_dict(hypothesis, In, Text),
    check_hypothesis(Text, Verdict),
    verdict_dict(Text, Verdict, Out),
    reply_json_dict(Out).

% serve: listen on the loopback interface, port
% 8080, and wait forever.
serve :-
    http_server([port(localhost:8080)]),
    thread_get_message(_).
```

Each request runs in a worker thread of its own, so the knowledge base is read concurrently; SWI-Prolog's clause store is safe for concurrent reading, and a service that also writes, by asserting annotations, serializes the writes with `with_mutex/2` or confines them to one thread. Two cautions belong to any such service. The hypotheses it receives are text from outside, and Section 31.9 shows the checks that keep a goal from doing what `shell/1` does, from running without end, and from being proved without its evidence. And the port is bound to the loopback interface, `port(localhost:8080)`, so that only processes on the same machine reach it; the address is removed only when the service is meant for the network, in which case the HTTP libraries' authentication and TLS support are the next things to read about.

31.5 Embedding the Engine in C

SWI-Prolog is a library, `libswipl`, before it is an executable, and the executable is a short C program that calls it. The **foreign language interface** in `SWI-Prolog.h` lets any C or C++ program do the same: start the engine, load a knowledge base, run queries, and read the answers. The code in this section is illustrative C in text blocks. It was not compiled or executed; the names are those of the interface as documented for this edition, and a reader who builds the program should expect to adjust it to the toolchain and to the header of the installed version.

The engine is started with `PL_initialise`, which takes the arguments `swipl` itself would take. Terms are handled through **term references**, handles of type `term_t` obtained from `PL_new_term_ref` and `PL_new_term_refs`, which are valid until the enclosing foreign frame or query is closed. A goal as text is converted with `PL_chars_to_term` and run with `PL_call`, which succeeds or fails as the goal does and gives no access to alternative answers.

```
#include <SWI-Prolog.h>
#include <stdio.h>

int main(int argc, char **argv)
{ char *av[] = { argv[0], "-q", NULL };

  if ( !PL_initialise(2, av) )
    PL_halt(1);

  { term_t goal = PL_new_term_ref();
    PL_chars_to_term("[dump1], use_module(patterns_dump)",
                     goal);
    if ( !PL_call(goal, NULL) )
      fprintf(stderr, "loading failed\n");
  }
  ...
```

Enumerating answers uses a **query** object. `PL_predicate` looks up a predicate by name, arity, and module; `PL_open_query` opens a query on it with an array of argument term references; `PL_next_solution` advances to the next answer, binding the arguments, and returns false when there are none; `PL_close_query` closes it and discards the bindings, where `PL_cut_query` keeps them. Integers, atoms, and strings are read from term references with `PL_get_integer`, `PL_get_atom_chars`, and `PL_get_chars`. The query below enumerates the **Spiking Thread** (page 335) instances of the module loaded above, as `spiking_thread(T)` does at the toplevel.

```
  { predicate_t p = PL_predicate("spiking_thread", 1, "user");
    term_t args = PL_new_term_refs(1);
    qid_t q = PL_open_query(NULL, PL_Q_NORMAL, p, args);

    while ( PL_next_solution(q) )
    { int tid;

      if ( PL_get_integer(args, &tid) )
        printf("Spiking thread %d\n", tid);
    }
    PL_close_query(q);
  }

  PL_halt(0);
}
```

The program is compiled against the header and linked with the library, for which the distribution provides `swipl-ld` and the flags in `swipl --dump-runtime-variables`; the details are those of the platform's C toolchain. The reverse direction, Prolog calling C, is a **foreign predicate**: a C function with one `term_t` per argument that returns true or false, unifying its outputs with `PL_unify_integer`, `PL_unify_atom_chars`, and their relatives, and registered with `PL_register_foreign`. The function below is the shape of what the next section needs: a predicate `dbg_execute/2` that hands a command to some host and unifies the second argument with the host's reply.

```
static foreign_t
pl_dbg_execute(term_t cmd, term_t out)
{ char *s;

  if ( !PL_get_chars(cmd, &s, CVT_ATOM|CVT_STRING|REP_UTF8) )
    return PL_type_error("text", cmd);
  const char *reply = host_execute(s);   /* the host's API */
  if ( !reply )
    return FALSE;
  return PL_unify_atom_chars(out, reply);
}

install_t
install_dbgpl(void)
{ PL_register_foreign("dbg_execute", 2, pl_dbg_execute, 0);
}
```

A shared library with such an `install` function is loaded from Prolog with `use_foreign_library/1`, after which `dbg_execute/2` is an ordinary predicate. Foreign predicates must observe one rule that Prolog programmers rarely think about: the engine is not re-entrant across threads, and a C thread that wants to call Prolog must either be the one that initialized the engine or attach an engine of its own with `PL_thread_attach_engine`.

31.6 A Debugger Extension: A Design

The pipeline of Chapter 15 runs the debugger, logs its output, parses the log into facts, and consults the facts. It works, and it has a cost: the analyst must decide in advance which commands to run, and a question that needs a command that was not run means a second pass. A debugger extension that hosts the Prolog engine removes the pass. This section presents such an extension as a design: the one that follows most directly from the embedding interface of the previous section, with the engine inside the debugger and the dump exposed to it on demand. The next section presents the extension the author built, which decided two of the questions below the other way, and compares the two.

The Windows debuggers load extensions as DLLs, and an extension receives the debugger engine's interfaces, through which it can read memory, enumerate threads and modules, walk stacks, resolve symbols, and execute debugger commands and capture their output. The extension of this design links `libswipl`, calls `PL_initialise` when it is loaded, and registers two kinds of things with each side. With the debugger it registers extension commands: one that consults a Prolog file, one that runs a query and prints its answers in the debugger's command window, and one that prints the explanation of Chapter 29 for a goal. With Prolog it registers foreign predicates, and these are the heart of the design: the predicates of the schema of Part II, implemented not as facts but as foreign predicates that ask the engine.

`frame/5` is the example. A foreign `frame/5` called with the thread bound and the frame number unbound asks the engine for that thread's stack, converts each frame to the schema's arguments, the module name lowercased, the function name as the symbol server returned it, the offset as an

integer, and produces them one at a time as nondeterministic foreign predicates can, through `PL_retry` and the control argument the interface provides for the purpose. Called with both bound, it yields one frame; called with the thread unbound, it enumerates the threads first. The dump is thereby exposed to Prolog **on demand**: nothing is translated that no query asks for, and a query over a complete memory dump with thousands of threads touches the stacks of the threads it needs. The same construction gives `module/3`, `thread_state/2`, `critical_section/4` from the engine's view of the critical section list, and `exception_record/3` from the exception record of the last event. Where the engine offers no interface for something a command prints, the foreign predicate executes the command, captures the output, and parses it with the DCGs of Chapter 12; `dbg_execute/2` of the previous section is that path exposed to Prolog directly.

Two refinements follow. The first is caching. An engine call is slower than a clause lookup by orders of magnitude, and a recognizer such as the **Stack Trace Collection (Unmanaged Space)** (page 330) of Chapter 18 visits every frame of every thread several times. The foreign predicate therefore asserts what it translates, as facts in a dynamic predicate, and the schema predicate is defined in Prolog as the union of the cached facts and the foreign enumerator for what has not been fetched, with a per-thread flag that records completeness. Chapter 11's dynamic database and Chapter 30's indexing do the rest, and the cache is discarded when the debugger moves to another dump or another process. The second is the other direction. A Prolog program can drive the debugger: a recognizer that needs the raw stack of a thread, as the **Execution Residue (User Space)** (page 350) of Chapter 18 does, calls `dbg_execute/2` with the command that dumps it, and an analysis script runs the debugger through a sequence of commands conditioned on what earlier answers showed, which is the automation of the analyst's own procedure.

```
% Illustrative: the Prolog side of the extension.
% dbg_frame/5 and dbg_execute/2 are foreign.
:- dynamic cached_frame/5, frames_complete/1.

frame(TID, N, M, F, O) :-
    (   frames_complete(TID)
    ->  true
    ;   fetch_frames(TID)
    ),
    cached_frame(TID, N, M, F, O).

fetch_frames(TID) :-
    forall(dbg_frame(TID, N, M, F, O),
           assertz(cached_frame(TID, N, M, F,
                                O))),
    assertz(frames_complete(TID)).
```

The constraints of the design are those of the host. Debugger extensions are called on the debugger's thread and must not block it for long, so a long analysis is started on a Prolog thread of its own and its result collected later. Symbol resolution is the debugger's, with the uncertainty that Chapter 18 discussed under **Coincidental Symbolic Information** (page 347), and a foreign `frame/5` cannot make a symbol more reliable than the engine makes it; it can record, in a `source/3` fact as Chapter 15 proposed, which engine call produced each fact. And the engine's view is of a dump at a moment, or of a live process that is moving, and a cache built from a live target is stale as soon as the target

runs; the extension invalidates it on every event the engine reports. With these constraints respected, the result is a debugger in which the analyst types a query where a command would go, and a Prolog program in which the dump is a knowledge base that was never written to a file.

31.7 PrologWinDbg: The Extension as Built

The extension exists. **PrologWinDbg** is an extension for the Windows debuggers, WinDbg and CDB, written by the author of this book and published in source form at:

github.com/DumpAnalysis/PrologWinDbg;

the version this section describes is 0.2.0, an experimental release of 20 October 2026, for native x64 and ARM64 user-mode targets, dumps and stopped live processes. What follows is what the repository documents and what its Prolog module does, and the epistemic status of each part is stated as the book has stated it throughout. The extension was not run in the environment in which this book was drafted, which has no Windows debugger. The Prolog side of it, the module `wd` that every rule imports, is reproduced verbatim in the book's package as `wd.pl` and is executed below against a synthetic snapshot, so the transcripts establish what that module does with the facts it is given, and nothing about the DLL that gives them; the debugger session shown later is reproduced from the repository's own validation log of an x64 run under CDB, abridged, and is the repository's evidence, not the book's.

The design of Section 31.6 put the engine inside the debugger and exposed the dump through foreign predicates that fetch on demand. The extension as built does neither. It runs SWI-Prolog as a **separate, persistent worker process**, started on first use from the installed `swipl.exe`, and the DLL talks to it over local pipes in a line protocol of tab-separated, hex-encoded fields. Each `!prolog.query` command takes a **snapshot** on the debugger's command thread: the threads the engine enumerates, up to a limit, each thread's stack up to a per-thread frame limit, the symbol of every captured instruction address, the loaded modules, the target's kind and process identifier, the processor architecture, and a coverage statement for each of these, saying what was captured and what the engine did not certify. The snapshot is sent to the worker with the goal as a list of ground facts of a fixed vocabulary, which the module validates before asserting them: a fact of another shape, a directive, or a clause is rejected with a domain error, so that the fact channel cannot carry code. The goal then runs under a budget, a time limit, an inference limit, and an answer limit, and its answers come back one at a time, printed in the command window as `bindings(Names, Residual)`, followed by a status that says whether the search was exhausted, failed, reached a limit, timed out, or raised. Two requests go the other way during the query: a memory read and a symbol lookup are relayed to the DLL, serviced on the command thread against the live engine, and cached for the duration of the query. When the query ends, the facts and the caches are cleared; the consulted rules persist. A query that exhausts its time or inference budget leaves the worker and its consulted rules in place. A transport failure, or an interruption from the debugger while the extension waits on the worker, stops the worker; the next command starts a fresh one, and the rules must be consulted again. A Windows job object bounds the worker's memory.

The predicates of `wd` are these. Every fact carries the snapshot identifier as its first argument, an integer local to the loaded extension; `current_snapshot/1` returns the one of the running query and raises an existence error outside one.

Predicate	Meaning
`target(S, Kind, PID, PointerBytes)`	`dump` or `live`; the process identifier; 8
`architecture(S, A)`	`amd64` or `arm64`
`thread(S, T)`	`T` is the engine's thread number
`thread_id(S, T, TID)`	the Windows thread identifier of `T`
`stack_frame(S, T, Depth, IP)`	a captured frame, `Depth` from 0, `IP` an integer address
`stack_pointer(S, T, Depth, SP)`	its stack pointer
`symbol(S, IP, Module, Function, Displacement)`	the symbol of a captured address, names as the debugger returned them
`symbol_status(S, IP, Status)`	`resolved` or `unavailable(Reason)`
`module(S, Name, Base, Size)`	a loaded module; unloaded modules are not captured
`coverage(S, Area, Status)`	what the capture holds, per area
`memory(S, Address, Length, Result)`	a read of at most 64 KiB: `ok(Bytes)`, `partial(Bytes, Length, Reason)`, `unavailable(Reason)`
`bytes(S, Address, Length, Bytes)`	the same read, raising if it is not complete
`pointer(S, Address, Value)`	one target pointer, read little-endian
`lookup(S, Name, Result)`	an address for `'module!symbol'`: `ok(Address)` or `unavailable(Reason)`

The coverage facts are the extension's own version of Section 15.5. The thread list is `enumerated(N)` or `truncated(Captured, Total)`; the module list is `enumerated(N)`, `partial(Captured, Total)` or `unavailable(Reason)`; and a stack is `captured_prefix(N, unverified_end)` when the engine returned `N` frames and stopped, `captured_prefix(N, frame_limit)` when the per-thread limit was reached, or `unavailable(Reason)`. The documentation says of the first that an enumerated list is what the engine returned for that stop and not proof that a limited dump holds every thread that existed, and of the stacks that the status deliberately makes no completeness claim: the engine does not certify an unwind, and optimized, inlined or tail-called frames may be absent. These are the **Truncated Stack Trace** (page 346) and **Incorrect Stack Trace** (page 345) of Chapter 18, stated as facts of the capture rather than left for the analyst to suspect. The same documentation says what Part II said of negation: with incomplete evidence, `\+ wd:stack_frame(S, T, _, IP)` means that no such fact was captured and not that the function was never on the stack, and it recommends what Section 17.3 required, an explicit result term in place of a bare failure.

The commands are few.

Command	Effect
`.load prolog.dll`	loads the extension

Command	Effect
`!prolog.runtime "path/swipl.exe"`	names the runtime; stops a running worker
`!prolog.info`	starts the worker and reports its version
`!prolog.consult "path/rules.pl"`	loads or reloads a rule file
`!prolog.query Goal.`	takes a snapshot and runs `Goal` over it
`!prolog.queryfile "path/goal.txt"`	the same, with the goal read from a file
`!prolog.eval Goal.`	runs `Goal` with no snapshot
`!prolog.limits [Ms Answers Frames Inferences]`	shows or sets the budget
`!prolog.reset`	stops the worker; rules are lost

The defaults are five seconds, one hundred answers, 128 frames per thread, and ten million inferences; a goal with a disjunction is written in double quotes, since the debugger’s command separator is the semicolon, and a goal with quoting the command line would mangle goes in a file. The following is the repository’s validation log of an x64 run under CDB, abridged to the lines that show the shape of a session and with the long lines of the log folded at the page width, the folds indented; the paths and the warning about a checksum are the log’s.

```
0:000> .load "D:/.../dist/x64/prolog.dll"
0:000> !prolog.runtime "D:/.../swipl-x64/bin/swipl.exe"
[prolog] Runtime set to D:\a\_temp\swipl-x64\bin\swipl.exe
0:000> !prolog.info
[prolog] 0.2.0 / protocol 1; extension=amd64;
  SWI-Prolog 10.1.18; worker running
0:000> !prolog.consult "D:/.../examples/patterns.pl"
[prolog] ok; 0 answer(s). File consulted
0:000> !prolog.eval X is 6*7, X=42, writeln('PD_ARITHMETIC_OK').
PD_ARITHMETIC_OK
bindings(['X'=42],[])
[prolog] ok; 1 answer(s). Search exhausted for the captured
  evidence
0:001> !prolog.query wd:current_snapshot(S),
  findall(T,wd:thread(S,T),Ts), length(Ts,N), N>=3,
  writeln('PD_THREADS_OK').
*** WARNING: Unable to verify checksum for prologdbg_target.exe
[prolog] Snapshot 4
PD_THREADS_OK
bindings(['S'=4,'T'=_155800,'Ts'=[0,1,2,3,4,5],'N'=6],[])
[prolog] ok; 1 answer(s). Search exhausted for the captured
  evidence
0:001> !prolog.query wd:current_snapshot(S),
  once(patterns:shared_function(S,A,B,prologdbg_target,
                                'WaitForRelease')),
  writeln('PD_STACKS_OK').
[prolog] Snapshot 7
PD_STACKS_OK
bindings(['S'=7,'A'=4,'B'=5],[])
[prolog] ok; 1 answer(s). Search exhausted for the captured
  evidence
0:001> !prolog.limits 100 100 128 10000000
[prolog] 100 ms; 100 answers; 128 frames/thread;
  10000000 inferences
0:001> !prolog.eval sleep(2).
[prolog] timeout; 0 answer(s). Time budget exhausted; results
  are incomplete
```

Each query is a new snapshot, numbered, and the status line after the answers is the one that matters for evidence: `Search exhausted for the captured evidence` is the strongest statement the extension makes, exhaustion over what was captured, and the timeout line says that whatever was printed before it is incomplete. The worker's SWI-Prolog is whatever is installed, 10.1.18 in this log; the rules are ordinary Prolog, and nothing in this book's recognizers depends on a version past 9.0.4.

31.7.1 The Two Designs Compared

The two designs answer the same four questions differently, and each answer has a cost. The comparison is the point of keeping both in the book.

Where the engine runs. The design of Section 31.6 links `libswipl` into the DLL. The cost of a call is then that of a function call, a term is handed over without serialization, and one binary is deployed. The price is isolation: a Prolog program that loops, exhausts the stacks or crashes does so inside the

debugger, and takes the session with it; memory grows in the debugger's process; and the DLL is bound to the version of the runtime it was built against. The extension as built pays the other price. Every memory read and every symbol lookup is a round trip through a pipe, which is why the module caches exact repeated reads for the length of a query and why the reads are capped at 64 KiB each and 16 MiB a query; the facts are serialized as text, capped at 8 MiB; two processes are deployed, and a foreign library in the worker that writes to the process's own standard output can corrupt the protocol. What it buys is that a runaway query costs a worker restart and the rules consulted since, never the debugger; that a job object can bound the worker's memory; that the runtime is the user's installation, of any version, so that the extension and the rules are decoupled from the DLL; and that the worker and the protocol can be tested without a debugger, which the repository does with a simulated debugger peer and the real runtime, the 99 portable checks of its validation report, on any platform. There is a third consideration the design of Section 31.6 did not see. The design proposed to run a long analysis on a Prolog thread of its own so as not to block the debugger's thread. The debugger engine's client objects have thread affinity: calls into the engine must come from the thread that owns the client, which is the command thread the extension is called on. A Prolog thread inside the DLL could therefore compute, but could not read memory or resolve a symbol; the extension as built makes the same restriction explicit, relaying the worker's requests on the command thread and allowing them from the worker's main thread only, and it answers the blocking problem with budgets and an interrupt check between engine calls rather than with a thread.

How the dump reaches Prolog. The design exposes the dump on demand: a foreign `frame/5` fetches the stack of a thread when a query asks for it and a cache keeps what was fetched. Nothing is translated that is not needed, which matters for a complete dump with thousands of threads, and the knowledge base grows with the questions. The difficulty is the one the design itself noted, invalidation. On a live target the cache is stale as soon as the target runs; and even on a dump, a cache filled across several queries holds the stack of one thread fetched at one moment and of another fetched later, with a per-thread completeness flag to maintain, and the meaning of a negated goal depends on what has been fetched so far. The extension as built takes one snapshot per query. Every fact of a query then dates from one stop of the target; the facts are an explicit, finite list that the module validates; a negated goal means one thing, that no such fact is in this snapshot; and nothing is retained, so nothing goes stale. The cost is proportional to the snapshot: the threads times the frame limit, with the symbol of every captured address resolved, which the documentation warns is slow before the symbol cache is warm, repeated on every query of a live session, and bounded by the limits, so that a stack deeper than 128 frames arrives as a prefix marked `frame_limit`, which the Stack Overflow recognizers of Chapter 19 must read before they count runs. The lazy part of the design survives in two places: memory and symbol lookups are fetched on request, with the cache the design proposed, for the length of a query.

What the schema is. The design implemented the schema of Part II directly, `frame/5` with the module name lowercased and the function as the symbol server returned it. The extension's schema is the engine's: an address is a term in its own right, the symbol of an address is a separate fact with a status, the names are as the debugger returned them, case included, and a thread has the engine's number and its Windows identifier both. The advantage is fidelity, a frame without a symbol is a

frame with an address and an `unavailable` status rather than the `none` and `'Unknown'` of Section 15.4, and no information is lost in a convention; the price is that this book's recognizers do not run over it as they stand. The bridge below pays that price in thirty lines, and the transcript shows the recognizers of Chapter 18 running over a snapshot unchanged.

Whether Prolog drives the debugger. The design had `dbg_execute/2`, a foreign predicate that runs a debugger command and returns its output for the DCGs of Chapter 12 to parse, and with it the automation of the analyst's own procedure, a sequence of commands conditioned on earlier answers. The extension as built has no such predicate, by decision: a symbol name is passed to the engine's lookup interface and not to the command interpreter, there is no predicate that writes to the target or controls its execution, and the fact channel carries facts only, so that a rule file, which is a trusted program, is the only code that runs, and a goal typed at the command line cannot run a debugger command. What that forgoes is everything the engine shows only through a command: the critical section list, the heap summary, the handle table, the thread times, the exception record, which is why version 0.2.0 captures none of the relations of Part II beyond threads, frames, symbols and modules, and why the recognizers of Chapters 19 to 22 do not yet apply inside it. Breakpoint actions written in Prolog are listed in the repository as planned.

What both must respect. The evidence discipline of Part II is not a choice between the designs; both must keep it, and the extension as built keeps it in the interface: coverage facts that make no claim they cannot support, three-valued memory results, residual goals returned with the bindings, and a status line that says that an answer limit does not assert that more solutions exist. Sections 15.5 and 17.8 are the theory of those facts, and the bridge below maps the one onto the other.

31.7.2 The Schema of Part II over a Snapshot

The bridge defines the relations of the schema that a snapshot can supply, `thread/2`, `frame/5` and `module/3`, over the facts of `wd`, and declares the rest dynamic and empty, so that a recognizer that consults them fails rather than raising an existence error and so that their coverage is unknown. It imports `current_snapshot/1` only, since `wd` also exports `thread/2` and `module/4`, which would collide with the definitions in `user`, and refers to the other predicates by module. `stack_coverage/2` states the coverage of `frame/5` for one thread in the terms of Section 15.5; it is never `complete`, because the engine certifies no unwind.

```
% wd_bridge.pl -- the schema of Part II over the
% snapshot of a PrologWinDbg query. Loaded into
% user beside patterns_dump, whose recognizers
% call user:frame/5 and the other relations.
:- use_module(wd, [current_snapshot/1]).
```

```
% The relations the extension does not capture
% in version 0.2.0: declared, and empty, so that
% a recognizer fails on them rather than raising
% an existence error, and so that their coverage
% is unknown (Section 15.5), never complete.
:- dynamic process/2, module_info/3,
           thread_time/3, thread_state/2,
           exception_record/3, exception_info/3,
           critical_section/4, waits_for/3,
           handle_count/3, heap/4, last_error/2,
           hooked/3.

% thread(TID, PID): every thread the engine
% enumerated, by its Windows thread identifier,
% in the process the snapshot was taken from.
thread(TID, PID) :-
    current_snapshot(S),
    wd:target(S, _, PID, _),
    wd:thread_id(S, _, TID).

% frame(TID, N, Module, Function, Offset): a
% captured frame with its symbol, the module name
% lowercased as Section 15.4 requires; a frame
% whose address has no symbol is recorded as
% frame(TID, N, none, 'Unknown', 0), as there.
frame(TID, N, Module, Function, Offset) :-
    current_snapshot(S),
    wd:thread_id(S, T, TID),
    wd:stack_frame(S, T, N, IP),
    (   wd:symbol(S, IP, M0, Function, Offset)
    ->  downcase_atom(M0, Module)
    ;   Module = none, Function = 'Unknown',
        Offset = 0
    ).

% module(Name, Base, Size): a loaded module.
module(Name, Base, Size) :-
    current_snapshot(S),
    wd:module(S, M0, Base, Size),
    downcase_atom(M0, Name).

% stack_coverage(TID, Status): the coverage of
% frame/5 for one thread, in the terms of Section
% 15.5. The engine certifies no unwind, so the
% status is never complete: a captured prefix is
% incomplete, with the reason the engine gives,
% and a thread the snapshot does not hold is
% unknown.
stack_coverage(TID, Status) :-
    current_snapshot(S),
    (   wd:thread_id(S, T, TID),
        wd:coverage(S, stack(T), C)
    ->  (   C = captured_prefix(N, Why)
        ->  Status = incomplete([prefix(N, Why)])
        ;   Status = incomplete([C])
        )
    ;   Status = unknown
    ).
```

The snapshot below is synthetic, shaped as the DLL sends one: thread 102 of `dump1.pl` as the engine would capture it, engine thread 2 with the Windows identifier 102, its seven frames as addresses, the symbol of each address with the module name as a debugger would return it, three of the modules, and the coverage statements. `install_snapshot/2` is the module's entry point for the protocol, not exported, and it is called by module here as the worker calls it; it validates every fact against the vocabulary before asserting any, which the last query of the transcript shows.

```
% sample_snapshot(Facts): a snapshot shaped as the
% extension sends it, for thread 102 of dump1.pl
% as the engine would capture it: engine thread
% 2, the seven frames, the symbols of their
% addresses, and the coverage statuses.
sample_snapshot(
    [ target(1, dump, 4120, 8),
      architecture(1, amd64),
      coverage(1, threads, enumerated(1)),
      thread(1, 2),
      thread_id(1, 2, 102),
      coverage(1, stack(2),
          captured_prefix(7, unverified_end)),
      stack_frame(1, 2, 0, 0x7ffb_8e41_1a3e),
      stack_frame(1, 2, 1, 0x7ffb_8e41_209a),
      stack_frame(1, 2, 2, 0x7ffb_8e45_0214),
      stack_frame(1, 2, 3, 0x7ff6_1d11_2120),
      stack_frame(1, 2, 4, 0x7ffb_ae51_8042),
      stack_frame(1, 2, 5, 0x7ffb_b1c1_7614),
      stack_frame(1, 2, 6, 0x7ffb_b2a5_2a21),
      symbol(1, 0x7ffb_8e41_1a3e, 'ReportEngine',
             'Table::Find', 0x3e),
      symbol(1, 0x7ffb_8e41_209a, 'ReportEngine',
             'Table::Insert', 0x9a),
      symbol(1, 0x7ffb_8e45_0214, 'ReportEngine',
             'Report::Build', 0x214),
      symbol(1, 0x7ff6_1d11_2120, 'AppServer',
             'Worker::Run', 0x120),
      symbol(1, 0x7ffb_ae51_8042, ucrtbase,
             'thread_start<...>', 0x42),
      symbol(1, 0x7ffb_b1c1_7614, kernel32,
             'BaseThreadInitThunk', 0x14),
      symbol(1, 0x7ffb_b2a5_2a21, ntdll,
             'RtlUserThreadStart', 0x21),
      coverage(1, modules, enumerated(3)),
      module(1, 'AppServer', 0x7ff6_1d10_0000,
             0x3c000),
      module(1, 'ReportEngine', 0x7ffb_8e40_0000,
             0x1a2000),
      module(1, ntdll, 0x7ffb_b2a0_0000, 0x1f8000)
    ]).

?- use_module(patterns_dump).
true.

?- set_prolog_flag(toplevel_print_anon, false).
true.
```

```
?- sample_snapshot(_Fs),
   wd:install_snapshot(1, _Fs).
true.

?- wd:current_snapshot(S),
   wd:target(S, Kind, PID, _),
   wd:architecture(S, Arch).
S = 1,
Kind = dump,
PID = 4120,
Arch = amd64.

?- thread(T, P).
T = 102,
P = 4120.

?- frame(102, N, M, F, O).
N = 0,
M = reportengine,
F = 'Table::Find',
O = 62 ;
N = 1,
M = reportengine,
F = 'Table::Insert',
O = 154 ;
N = 2,
M = reportengine,
F = 'Report::Build',
O = 532 ;
N = 3,
M = appserver,
F = 'Worker::Run',
O = 288 ;
N = 4,
M = ucrtbase,
F = 'thread_start<...>',
O = 66 ;
N = 5,
M = kernel32,
F = 'BaseThreadInitThunk',
O = 20 ;
N = 6,
M = ntdll,
F = 'RtlUserThreadStart',
O = 33.

?- stack_trace_signature(102, _Sig),
   forall(member(M-F, _Sig), ( print(M-F), nl )).
reportengine-'Table::Find'
reportengine-'Table::Insert'
reportengine-'Report::Build'
appserver-'Worker::Run'
ucrtbase-'thread_start<...>'
kernel32-'BaseThreadInitThunk'
ntdll-'RtlUserThreadStart'
true.
```

```
?- module(Name, Base, Size).
Name = appserver,
Base = 140695026270208,
Size = 245760 ;
Name = reportengine,
Base = 140718400077824,
Size = 1712128 ;
Name = ntdll,
Base = 140719010349056,
Size = 2064384.

?- stack_coverage(102, S1),
   stack_coverage(103, S2).
S1 = incomplete([prefix(7, unverified_end)]),
S2 = unknown.

?- \+ frame(103, _, _, _, _).
true.

?- thread_time(102, _, _).
false.

?- catch(wd:install_snapshot(2, [(:- halt)]),
         error(E, _), true).
E = domain_error(snapshot_fact, (:-halt)).
```

The frames and the signature are those of Chapter 18 for thread 102, obtained from a snapshot instead of from `dump1.pl`, with the module names lowercased by the bridge and the offsets printed in decimal by the toplevel. The three queries after the modules are the evidence discipline at work. The coverage of thread 102 is incomplete with the engine's reason, since no unwind is certified; the coverage of thread 103, which the snapshot does not hold, is unknown, and the negated goal that follows succeeds for that thread and is not evidence of anything, exactly as Section 15.5 said of a thread `!teb` was not run on. `thread_time/3` fails because the extension does not capture it, which is the correct answer for a knowledge base that has no such relation; **Spiking Thread** (page 335), which needs it, finds nothing, and a report over a snapshot says so under `coverage/2` rather than reporting the absence of spiking threads. The last query is the fact channel refusing a directive.

Memory is the one relation the design and the extension read the same way, on request. The rule below is the one the extension's distribution ships, verbatim, and the three clauses after it stand in for what the DLL would answer three read requests with: a complete read, a short read at the edge of a readable page, with the engine's result code in the reason, and a failed one. `memory_provider/4` is the hook `wd` declares for the purpose; the worker binds it to the pipe, and a test binds it to facts, as here.

```
% inspect_magic(S, Address, Verdict): the example
% rule of the extension's distribution, verbatim:
% a three-way verdict on eight bytes, and never a
% bare failure.
inspect_magic(S,Address,Verdict) :-
    wd:memory(S,Address,8,Result),
    ( Result = ok([80,82,79,76,79,71,68,66])
      -> Verdict=matched(prologdb,Address)
    ; Result = ok(Bytes)
      -> Verdict=mismatch(Bytes)
    ; Verdict=inconclusive(Result) ).

% What the DLL would answer three READ requests
% with: a full read, a short read at a page edge,
% and a failed one. The hook is the module's own
% extension point; the worker binds it to the
% pipe.
:- multifile wd:memory_provider/4.
wd:memory_provider(1, 0x7ff6_1d13_a000, 8,
    ok([80,82,79,76,79,71,68,66])).
wd:memory_provider(1, 0x7ff6_1d13_aff8, 8,
    partial([80,82,79,76], 8,
            "HRESULT 0x0; requested 8, read 4")).
wd:memory_provider(1, 0x10, 8,
    unavailable("HRESULT 0x8000FFFF")).

?- inspect_magic(1, 0x7ff6_1d13_a000, V).
V = matched(prologdb, 140695026507776).

?- inspect_magic(1, 0x7ff6_1d13_aff8, _V),
   _V = inconclusive(partial(Bs, N, _Reason)),
   sub_string(_Reason, 0, 11, _, Code).
Bs = [80, 82, 79, 76],
N = 8,
Code = "HRESULT 0x0".

?- inspect_magic(1, 0x10, inconclusive(V)).
V = unavailable("HRESULT 0x8000FFFF").

?- inspect_magic(1, 0x20, V).
V = inconclusive(unavailable(no_memory_provider)).

?- catch(wd:bytes(1, 0x7ff6_1d13_aff8, 8, _),
         error(incomplete_memory(A, L, _R), _),
         true),
   functor(_R, Shape, _).
A = 140695026511864,
L = 8,
Shape = partial.
```

A short read is inconclusive, not a mismatch: the four bytes read agree with the signature, and the four not read decide nothing, whatever the engine’s result code at the head of the reason says, which is the distinction between absence of evidence and evidence of absence that Chapter 17 drew for facts, here drawn for bytes. An address no provider answers is inconclusive with that reason, and `bytes/4`, which promises a complete read, raises rather than returning less. The verdicts are about eight bytes at one address at one stop, and claim nothing beyond that.

One more snapshot, of two threads with one frame each, is the regression case for the commands that close this section. The mode of `stack_trace/2` is `+TID`, as Section 18.1 said: with the identifier unbound, its `findall/3` merges the frames of every thread into one list, and `stack_trace_signature/2` returns one signature for the two threads and no thread; a query over all threads enumerates `thread/2` first, as `frame_regularity/3` does inside itself.

```
% two_thread_snapshot(Facts): two threads with one
% captured frame each, for the regression case.
two_thread_snapshot(
    [ target(2, dump, 4120, 8),
      architecture(2, amd64),
      coverage(2, threads, enumerated(2)),
      thread(2, 2), thread_id(2, 2, 102),
      thread(2, 3), thread_id(2, 3, 105),
      coverage(2, stack(2),
          captured_prefix(1, frame_limit)),
      coverage(2, stack(3),
          captured_prefix(1, frame_limit)),
      stack_frame(2, 2, 0, 0x7ffb_8e41_1a3e),
      stack_frame(2, 3, 0, 0x7ff6_1d11_3b10),
      symbol(2, 0x7ffb_8e41_1a3e, 'ReportEngine',
             'Table::Find', 0x3e),
      symbol(2, 0x7ff6_1d11_3b10, 'AppServer',
             'Idle::Wait', 0x10),
      coverage(2, modules, enumerated(0))
    ]).

?- two_thread_snapshot(_Fs),
   wd:install_snapshot(2, _Fs).
true.

?- stack_trace_signature(T, _Sig),
   length(_Sig, N).
N = 2.

?- thread(T, _), stack_trace_signature(T, Sig).
T = 102,
Sig = [reportengine-'Table::Find'] ;
T = 105,
Sig = [appserver-'Idle::Wait'].
```

The first query returns two keys and leaves `T` unbound, which is a list of frames from two threads and the signature of neither; the second returns one signature per thread. To run this book's recognizers inside the debugger, the three files are consulted in the worker, the bridge's `use_module(wd, ...)` pointed at the module in the extension's distribution, and a query of Chapter 18 typed where a command would go, with the enumeration in front of a recognizer whose mode requires the thread:

```
!prolog.consult "C:/book/patterns_dump.pl"
!prolog.consult "C:/book/wd_bridge.pl"
!prolog.query thread(T, _), stack_trace_signature(T, Sig).
!prolog.query frame_regularity(T, Run, N).
```

Each query is a fresh snapshot of the stopped target, and the recognizers that rest on threads, frames and modules, the stack traces, signatures, runs and regularities of Chapter 18 and those of Chapter 22 that read `module/3` alone, run over it as they ran over `dump1.pl`; those that rest on relations the extension does not yet capture fail, and a report should say under `coverage/2` which ones. The design of Section 31.6 would have run them all, at the cost of everything the previous subsection listed; the extension as built runs fewer, and says exactly which.

31.8 The Machine Query Interface

Between embedding, which puts Prolog inside another program, and a service, which puts it behind a socket for anyone, lies the case of a Python program that wants a Prolog engine of its own. SWI-Prolog's **Machine Query Interface** (MQI), `library(mqi)`, serves it. The library starts a server inside the Prolog process that accepts connections on a local socket, authenticates them with a password it generates, and runs the queries it receives, returning answers as JSON. On the Python side the package `swiplserver`, installed with `pip`, starts a `swipl` process in this mode, connects to it, and exposes a `PrologMQI` class whose threads run queries as strings and return the answers as lists of dictionaries from variable names to values, so that `deadlock(X, Y)` comes back as `[{"X": 103, "Y": 104}]` and a goal with no solutions as `False`. Each `PrologThread` object is a Prolog thread with its own query state, and queries may be run to completion or one answer at a time.

The version notes are these. The MQI ships with SWI-Prolog from version 8.5.x onwards, and the author verified that `library(mqi)` loads in the edition of this book, 9.0.4, and reports protocol version 1.0; earlier releases do not have it, and the `swiplserver` package requires a `swipl` that does. The package is developed alongside SWI-Prolog and documented in the SWI-Prolog manual's section on the MQI; the exact Python interface is the package's own documentation to state, and this book does not repeat it beyond the shape above.

```
?- use_module(library(mqi)).
true.

?- mqi_version(Major, Minor).
Major = 1,
Minor = 0.
```

Started by hand, with `mqi_start/1` and the options `port(P)` and `password(S)`, the server also serves as a general query endpoint for any language that can open a socket and speak the protocol the manual documents; `mqi_stop/1` stops it. For a diagnostic workflow the natural division is the one this book has assumed throughout: the artifacts and the recognizers live in Prolog, and the orchestration, the fetching of dumps and the rendering of reports, lives where the rest of the tooling does.

31.9 Prolog as a Verifier for LLM-Based Agents

A language-model agent that reads a debugger log is good at proposing: it has seen many stack traces and it will say, quickly and in prose, that threads 103 and 104 are deadlocked, that thread 102 is in a tight loop, and that the crash on thread 109 is a NULL pointer dereference. It is not reliable at checking, and its prose does not say which facts it used. A Prolog knowledge base of the dump is the complement. The agent proposes a **hypothesis** in the form of a goal, the knowledge base attempts to prove it, and the answer, with evidence derived from the proof as in Chapter 29, is returned to the agent or to the analyst. This section builds the verifier and then says what its answers mean.

The recognizers are those of Chapters 18 to 22 as the module `patterns_dump` of Chapter 23 collects them, loaded into a toplevel with the dump. The verifier therefore checks a hypothesis against the same definitions and thresholds as those chapters: `spiking_thread(T)` has the two answers Chapter 18 gives it, and `deadlock(critical_section, C)` the cycle of Chapter 20. The proof collector `prove/3` is a reduced form of `solve/2` of Chapter 29, producing the facts a proof used rather than the tree, with two additions that reading a module's clauses requires. The first is the **module context**. A clause read from `patterns_dump` has body goals that are resolved in that module, and `clause/2` in this edition returns such a body without a qualifier; so the interpreter carries the module in its first argument, switches it on a qualified goal `M:G`, and finds the module whose clauses define a goal with the property `imported_from(Module)` of `predicate_property/2`. The second is the set of constructs it interprets: besides the control constructs and the two collecting goals of Chapter 29, `once/1` and `bagof/3`, which the module's clauses use; `bagof/3` is interpreted for goals without `^/2`, which is how the module uses it. Built-ins and library predicates run natively, as in Chapter 29, with `module_property/2` telling a library module from the analyst's own.

A fact is recorded as a goal that `user` can call, as the contract of Section 17.3 requires: unqualified when `user` sees the predicate, which is the case for the relations of the dump and for the exported thresholds, and qualified with its module otherwise. And a fact is recorded only when its predicate is defined by facts alone. The base case `runs([], [])` of a recursive helper of the module is a clause without a body, and so is the first clause of the module's naming table `deadlock_name/2`, but neither is evidence of anything in the dump; the property `number_of_rules(0)` tells a relation of facts from a predicate that happens to have one.

```
:- [dump1].
:- use_module(patterns_dump).
:- use_module(library(http/json)).
:- use_module(library(sandbox)).

% prove(+Module, +Goal, -Facts): Goal, called in
% Module, has a proof that used the facts Facts.
prove(_, true, []).
prove(M, (A, B), Facts) :-
    prove(M, A, FA),
    prove(M, B, FB),
    append(FA, FB, Facts).
prove(M, (If -> Then ; Else), Facts) :-
    (   prove(M, If, FI)
```

```
    ->  prove(M, Then, FT),
        append(FI, FT, Facts)
    ;   prove(M, Else, Facts)
    ).
prove(M, (A ; B), Facts) :-
    A \= (_ -> _),
    (   prove(M, A, Facts)
    ;   prove(M, B, Facts)
    ).
prove(M, \+ A, []) :-
    \+ prove(M, A, _).
prove(M, once(A), Facts) :-
    once(prove(M, A, Facts)).
prove(M, findall(T, G, L), Facts) :-
    findall(T-F, prove(M, G, F), Pairs),
    pairs_keys_values(Pairs, L, Lists),
    append(Lists, Facts).
prove(M, bagof(T, G, L), Facts) :-
    bagof(T-F, prove(M, G, F), Pairs),
    pairs_keys_values(Pairs, L, Lists),
    append(Lists, Facts).
prove(M, aggregate_all(Spec, G, R), Facts) :-
    findall(Spec-F, prove(M, G, F), Pairs),
    pairs_keys_values(Pairs, Specs, Lists),
    aggregate_all(Spec, member(Spec, Specs), R),
    append(Lists, Facts).
prove(_, M:G, Facts) :-
    prove(M, G, Facts).
prove(M, G, Facts) :-
    \+ construct(G),
    (   system_goal(M:G)
    ->  call(M:G),
        Facts = []
    ;   definition_module(M:G, DefM),
        clause(DefM:G, Body),
        (   Body == true
        ->  fact_leaves(DefM, G, Facts)
        ;   prove(DefM, Body, Facts)
        )
    ).

% construct(@Goal): a goal that prove/3 handles by
% a clause of its own.
construct(true).
construct((_, _)).
construct((_ ; _)).
construct(\+ _).
construct(once(_)).
construct(findall(_, _, _)).
construct(bagof(_, _, _)).
construct(aggregate_all(_, _, _)).
construct(_:_).
```

```
% system_goal(+Goal): a built-in or a predicate of
% a library module, executed and not interpreted.
system_goal(G) :-
    predicate_property(G, built_in).
system_goal(G) :-
    predicate_property(G, imported_from(Lib)),
    module_property(Lib, class(library)).

% definition_module(+Goal, -Module): Module holds
% the clauses of the predicate of Goal.
definition_module(M:G, DefM) :-
    (   predicate_property(M:G,
                           imported_from(DefM))
    ->  true
    ;   DefM = M
    ).

% fact_leaves(+Module, +Fact, -Facts): Facts is
% [Fact] as a goal that user can call when Fact
% belongs to a relation defined by facts alone,
% and [] for the base case of a recursive
% predicate.
fact_leaves(M, Fact, Facts) :-
    (   \+ predicate_property(M:Fact,
                              number_of_rules(0))
    ->  Facts = []
    ;   predicate_property(user:Fact, visible)
    ->  Facts = [Fact]
    ;   Facts = [M:Fact]
    ).
```

The verifier admits a **query vocabulary** and nothing else. A hypothesis is a goal built from the predicates `patterns_dump` exports, the recognizers of Part III with their helpers and the thresholds, combined by a supported subset of the control language: conjunction, disjunction, negation, `once/1`, `findall/3` and `aggregate_all/3` over admitted goals, and the arithmetic comparisons, which let a hypothesis test a binding. Everything else is rejected, and the verdict names the leftmost subgoal outside the vocabulary: a goal for a predicate the knowledge base does not define, a built-in such as `length/2` whose answers the verifier would have to enumerate without end, `=/2`, a module-qualified goal, and a wrapper such as `forall/2` whose proof semantics `prove/3` does not implement. The vocabulary is what makes a `proved` verdict trustworthy as evidence: every admitted goal is one the collector understands, so the facts come with the verdict and not an empty list. The `pattern/3` and `recognizer/2` facts of Chapter 17 can supply the vocabulary to the agent.

```
% supported(@Goal, -Subgoals): Goal is a control
% construct of the vocabulary over Subgoals.
supported((A, B), [A, B]).
supported((A ; B), [A, B]).
supported(\+ A, [A]).
supported(once(A), [A]).
supported(findall(_, G, _), [G]).
supported(aggregate_all(_, G, _), [G]).
```

```
% comparison(@Goal): an arithmetic comparison.
comparison(_ < _).
comparison(_ > _).
comparison(_ =< _).
comparison(_ >= _).
comparison(_ =:= _).
comparison(_ =\= _).

% admitted(@Goal): Goal is in the vocabulary: a
% supported construct over admitted goals, a
% comparison, or an export of patterns_dump.
admitted(Goal) :-
    callable(Goal),
    Goal \= _:_,
    (   supported(Goal, Subgoals)
    ->  maplist(admitted, Subgoals)
    ;   comparison(Goal)
    ->  true
    ;   predicate_property(Goal,
                imported_from(patterns_dump))
    ).

% unsupported(@Goal, -Leaf): Goal is not admitted
% and Leaf is its leftmost subgoal that is not.
unsupported(Goal, Leaf) :-
    (   callable(Goal),
        supported(Goal, Subgoals),
        member(Sub, Subgoals),
        \+ admitted(Sub)
    ->  unsupported(Sub, Leaf)
    ;   Leaf = Goal
    ).
```

Three checks precede the proof, and each answers a different question. The hypothesis arrives as text, because that is what an agent produces; its length is checked first, since an agent can produce a goal of any size, and `term_string/3` then parses it and reports the variable names, which are the agent's names for the unknowns it wants filled in. The vocabulary check asks whether the collector understands the goal. `safe_goal/1` from `library(sandbox)` asks whether running it is safe: it expands the call graph of the goal statically and raises a permission error if the goal reaches anything the library classifies as unsafe, among them the shell, the file system, and the loading of code. That is a **safety check** and only that. It does not establish that the collector records the facts a goal uses, which the vocabulary does, and it does not establish that the goal terminates, which the limits do; a goal such as `once(deadlock(critical_section, C))` or `length(L, N)` passes the sandbox, and a verifier that relied on the sandbox alone would return the first proved without its facts and would not return from the second. With the vocabulary in place the sandbox is a second line, kept because it costs little: if a recognizer in the module were ever changed to call something unsafe, the vocabulary check would not notice and the sandbox would. When a hypothesis is rejected as unsupported, its variables are first bound to `'$VAR'(Name)` terms, the device of `numbervars/3` in Chapter 8, so that the offending subgoal prints with the agent's variable names, in the verdict and in the JSON of the response, rather than with names the system invents.

The limits are the third check, and they run with the proof rather than before it. The answers are collected with `findnsols/4`, which is `findall/3` with a bound on the number of solutions; asking for one more than the limit tells the verifier whether the limit was reached. The collection runs under `call_with_inference_limit/3` of Chapter 29, which stops the whole search when its budget of inferences is spent and reports `inference_limit_exceeded` in place of success. A limit reached is a distinct verdict, `incomplete(Reason, Answers)`, with the answers collected before the answer limit stopped the collection, or none when the inference limit stopped it. It is neither `proved`, since the search did not finish, nor `not_proved`, since nothing was refuted. `call_with_time_limit/2` of `library(time)` adds a wall-clock bound in the same place for a deployment whose budget is time rather than inferences; this section bounds inferences because a count is reproducible and a transcript can show it. The defaults, two hundred characters, ten answers, and a million inferences, are options of `check_hypothesis/3`, read with `option/3` of `library(option)`, and the two-argument form uses them.

```
% check_hypothesis(+Text, -Verdict): the verdict
% on the hypothesis Text under the default limits.
check_hypothesis(Text, Verdict) :-
    check_hypothesis(Text, [], Verdict).

% check_hypothesis(+Text, +Options, -Verdict):
% Verdict is proved(Answers), not_proved,
% incomplete(Reason, Answers), or rejected(Why);
% an answer is Bindings-Facts. Options: chars(N),
% answers(N), inferences(N).
check_hypothesis(Text, Options, Verdict) :-
    catch(verify(Text, Options, Verdict),
          error(Why, _),
          Verdict = rejected(Why)).

verify(Text, Options, Verdict) :-
    option(chars(MaxChars), Options, 200),
    string_length(Text, Length),
    (   Length > MaxChars
    ->  Verdict = rejected(input_limit(MaxChars))
    ;   term_string(Goal, Text,
                    [variable_names(Names)]),
        must_be(callable, Goal),
        verify(Goal, Names, Options, Verdict)
    ).

% verify(+Goal, +Names, +Options, -Verdict): the
% verdict on the parsed goal; Names are the
% agent's variable names.
verify(Goal, Names, Options, Verdict) :-
    (   admitted(Goal)
    ->  safe_goal(Goal),
        collect(Goal, Names, Options, Verdict)
    ;   unsupported(Goal, Leaf),
        maplist(name_variable, Names),
        Verdict = rejected(unsupported(Leaf))
    ).
```

```
% name_variable(+Binding): bind the variable of a
% Name = Var pair so that it prints as Name.
name_variable(Name = Var) :-
    Var = '$VAR'(Name).

% collect(+Goal, +Names, +Options, -Verdict): the
% answers of Goal with their facts, within the
% limits on answers and inferences.
collect(Goal, Names, Options, Verdict) :-
    option(answers(MaxA), Options, 10),
    option(inferences(MaxI), Options, 1_000_000),
    Chunk is MaxA + 1,
    call_with_inference_limit(
        once(findnsols(Chunk, Names-Facts,
                       answer(Goal, Facts),
                       Found)),
        MaxI, Result),
    (   Result == inference_limit_exceeded
    ->  Reason = inference_limit(MaxI),
        Verdict = incomplete(Reason, [])
    ;   length(Found, N),
        N > MaxA
    ->  length(Answers, MaxA),
        append(Answers, _, Found),
        Reason = answer_limit(MaxA),
        Verdict = incomplete(Reason, Answers)
    ;   Found == []
    ->  Verdict = not_proved
    ;   Verdict = proved(Found)
    ).

% answer(+Goal, -Facts): one proof of Goal, called
% in user, with its facts, each once.
answer(Goal, Facts) :-
    prove(user, Goal, Facts0),
    list_to_set(Facts0, Facts).

% print_verdict(+Text): the verdict, one line per
% binding and per fact.
print_verdict(Text) :-
    check_hypothesis(Text, Verdict),
    format("~s: ", [Text]),
    (   Verdict = proved(Answers)
    ->  length(Answers, N),
        format("proved, ~d answer(s)~n", [N]),
        print_answers(Answers)
    ;   Verdict = incomplete(Reason, Answers)
    ->  length(Answers, N),
        format("incomplete (~q), ~d answer(s)~n",
               [Reason, N]),
        print_answers(Answers)
    ;   format("~q~n", [Verdict])
    ).

print_answers(Answers) :-
    forall(member(Names-Facts, Answers),
           print_answer(Names, Facts)).
```

```
print_answer(Names, Facts) :-
    forall(member(N = X, Names),
           format("  ~a = ~q~n", [N, X])),
    forall(member(F, Facts),
           format("  ~q~n", [F])).

?- set_prolog_flag(toplevel_print_anon, false).
true.

?- print_verdict("deadlock(critical_section, C)").
deadlock(critical_section, C): proved, 1 answer(s)
  C = [103,104]
  waits_for(103,critical_section,140695026508096)
  critical_section(140695026508096,104,1,1)
  waits_for(104,critical_section,140695026507968)
  critical_section(140695026507968,103,2,1)
true.

?- print_verdict("spiking_thread(107)").
spiking_thread(107): proved, 1 answer(s)
  thread_time(107,3200,900)
  thread_time(100,120,40)
  thread_time(101,15,5)
  thread_time(102,8450,1230)
  thread_time(103,30,10)
  thread_time(104,25,10)
  thread_time(105,20,5)
  thread_time(106,10,5)
  thread_time(108,0,0)
  thread_time(109,40,20)
  threshold(spiking_share,25)
true.

?- print_verdict("spiking_thread(105)").
spiking_thread(105): not_proved
true.

?- print_verdict("blocked_thread(102)").
blocked_thread(102): not_proved
true.

?- check_hypothesis("spiking_thread(T)",
                    proved(_As)),
   pairs_keys(_As, Bindings).
Bindings = [['T'=102], ['T'=107]].

?- print_verdict("handle_leak(T)").
handle_leak(T): proved, 1 answer(s)
  T = event
  handle_count(4120,event,12040)
  threshold(handle_type_count,5000)
true.
```

The deadlock is proved with the four facts Chapter 20 lists as its evidence: each thread's wait and the ownership record of the section it waits for. The **Spiking Thread** (page 335) is proved with the times of every thread, since the share of thread 107 was computed against the total, and with the threshold; thread 105 is not one, thread 102 is not a **Blocked Thread (Software)** (page 336), and the

agent that asked which threads spike is told 102 and 107, as Chapter 18 does. The wrappers of the vocabulary preserve the evidence. A collector that ran `once/1` natively, as `solve/1` of Chapter 29 runs `forall/2`, would return the wrapped deadlock proved with no facts, which is a `proved` verdict that an agent cannot check; the first query below is the boundary case, and the ones after it show that the collecting goals preserve the evidence too.

```
?- _T1 = "deadlock(critical_section, C)",
   _T2 = "once(deadlock(critical_section, C))",
   check_hypothesis(_T1, proved([_-_Direct])),
   check_hypothesis(_T2, proved([_-_Once])),
   _Direct == _Once,
   length(_Direct, N).
N = 4.

?- _T = "findall(C, deadlock(any, C), L)",
   check_hypothesis(_T, proved([Bs-_Facts])),
   length(_Facts, N).
Bs = ['C'=_, 'L'=[[103, 104]]],
N = 4.

?- _T = "aggregate_all(count, waits(_, _, _), N)",
   check_hypothesis(_T, proved([Bs-_Facts])),
   length(_Facts, K).
Bs = ['N'=3],
K = 3.

?- check_hypothesis("\\+ deadlock(mutant, C)", V).
V = proved([['C'=_]-[]]).
```

The `findall/3` hypothesis binds `L` and leaves `C` unbound, as the goal itself would, and its four facts are those of the one cycle; the count of waits comes with the three wait facts that were counted. The negated hypothesis is proved with no facts, which is correct and is the limit noted in Chapter 29: the absence of a mutant cycle rests on facts that are not there, and a proof by negation has no leaves to record.

```
?- check_hypothesis("deadlock(103", V).
V = rejected(syntax_error(operator_expected)).

?- check_hypothesis("42", V).
V = rejected(type_error(callable, 42)).

?- check_hypothesis("length(L, N)", V).
V = rejected(unsupported(length(L, N))).

?- check_hypothesis("between(1, inf, X)", V).
V = rejected(unsupported(between(1, inf, X))).

?- check_hypothesis("X = f(a)", V).
V = rejected(unsupported(X=f(a))).

?- _T = "spiking_thread(T), shell('rm -rf /')",
   check_hypothesis(_T, V).
V = rejected(unsupported(shell('rm -rf /'))).
```

```
?- _T = "forall(spiking_thread(T), T > 100)",
   check_hypothesis(_T, rejected(unsupported(G))).
G = forall(spiking_thread(T), T>100).

?- check_hypothesis("hidden_exception(109)", V).
V = rejected(unsupported(hidden_exception(109))).

?- check_hypothesis("deadlock(bogus, C)",
       rejected(type_error(oneof(_), Culprit))).
Culprit = bogus.
```

The first two rejections are the parser's and the type check's. The rest are the vocabulary's, and each names the subgoal that put the hypothesis outside it, with the agent's variable names kept: the unbounded generators `length/2` and `between/3` are refused before they run, `=/2` is refused because it is not a recognizer, the shell command is refused by the vocabulary before the sandbox sees it, and `forall/2` is refused because `prove/3` has no clause for it. The hypothesis about a **Hidden Exception (User Space)** (page 374) is refused because no recognizer of that name is exported: the verifier can check only what the knowledge base defines, and an agent that wants to ask about a pattern must use the vocabulary. The last rejection is an error raised by a recognizer itself, `must_be/2` in `edge_of/4` of the module, which the verifier reports as it reports its own.

The limits are shown by lowering them, since no admitted hypothesis exceeds the defaults on a dump of ten threads. The answer limit returns the answers it collected and says that there were more; the inference limit returns none. The third and fourth queries run the collector on goals the vocabulary rejects, to show that the limits stop an unbounded generator on their own: `length/2` and `between/3` with an infinite bound have answers without end, and the collection ends with the answer limit, or, when that limit is out of the way, with the inference limit. The last query lowers the length limit below the length of a short hypothesis, which is rejected before it is parsed.

```
?- check_hypothesis("no_component_symbols(M)",
                    [answers(2)],
                    incomplete(Reason, _Answers)),
   pairs_keys(_Answers, Bindings).
Reason = answer_limit(2),
Bindings = [['M'=dbclient], ['M'=cachelib]].

?- check_hypothesis("deadlock(any, C)",
                    [inferences(100)], V).
V = incomplete(inference_limit(100), []).

?- collect(length(L, N), ['L'=L, 'N'=N],
           [answers(3)], incomplete(R, _As)),
   length(_As, K).
R = answer_limit(3),
K = 3.

?- collect(between(1, inf, X), ['X'=X],
           [answers(1000), inferences(500)], V).
V = incomplete(inference_limit(500), []).
```

```
?- check_hypothesis("spiking_thread(T)",
                    [chars(10)], V).
V = rejected(input_limit(10)).
```

The verdict is converted to JSON for the service of Section 31.4 by `verdict_dict/3`. Prolog terms are not JSON values: a binding to an integer or an atom happens to be one, a binding to a compound term is not, and a conversion that inserts bindings as they are fails on the second kind with a type error from the writer. Every binding and every fact is therefore encoded the same way, as a string of quoted Prolog text that `term_string/2` reads back as the term, with `numbervars/3` giving a letter to any variable inside a term and `_` to one that occurs once, so that the text has no system-generated names. A variable the proof left unbound is encoded as `null`, which is explicit and is also a loss: the encoding does not record that two unbound variables were the same variable. The response carries every answer the verifier collected, under the answer limit, as an array; `answers_shown` is the length of that array and `complete` says whether the collection finished, so that a consumer can tell ten answers of ten from the first ten of more, and `reason` is present when it did not finish or the hypothesis was rejected.

```
% verdict_dict(+Text, +Verdict, -Dict): JSON-ready
% form of a verdict.
verdict_dict(Text, Verdict, Dict) :-
    (   Verdict = proved(Answers)
    ->  answers_dict(Text, "proved", true,
                     Answers, Dict)
    ;   Verdict = incomplete(Reason, Answers)
    ->  answers_dict(Text, "incomplete", false,
                     Answers, Dict0),
        term_text(Reason, S),
        Dict = Dict0.put(reason, S)
    ;   Verdict == not_proved
    ->  answers_dict(Text, "not_proved", true,
                     [], Dict)
    ;   Verdict = rejected(Why),
        term_text(Why, S),
        Dict = _{hypothesis:Text,
                 verdict:"rejected", reason:S}
    ).

answers_dict(Text, Kind, Complete, Answers, D) :-
    maplist(answer_dict, Answers, Dicts),
    length(Answers, N),
    D = _{hypothesis:Text, verdict:Kind,
          complete:Complete, answers_shown:N,
          answers:Dicts}.

answer_dict(Names-Facts, D) :-
    maplist(binding_pair, Names, Pairs),
    dict_pairs(BDict, _, Pairs),
    maplist(term_text, Facts, Strings),
    D = _{bindings:BDict, evidence:Strings}.
```

```
% binding_pair(+Binding, -Pair): the agent's
% variable name with its value as text, or null
% when the proof left the variable unbound.
binding_pair(Name = Term, Name-Value) :-
    (   var(Term)
    ->  Value = null
    ;   term_text(Term, Value)
    ).

% term_text(+Term, -String): Term as quoted Prolog
% text that reads back as Term; a variable inside
% it is written as a letter, or as _ if it occurs
% once.
term_text(Term, String) :-
    copy_term(Term, Copy),
    numbervars(Copy, 0, _, [singletons(true)]),
    term_string(Copy, String, [numbervars(true)]).

?- check_hypothesis("handle_leak(T)", _V),
   verdict_dict("handle_leak(T)", _V, _D),
   json_write_dict(current_output, _D,
                   [width(40), tab(80)]).
{
  "answers": [
    {
      "bindings": {"T":"event"},
      "evidence": [
        "handle_count(4120,event,12040)",
        "threshold(handle_type_count,5000)"
      ]
    }
  ],
  "answers_shown":1,
  "complete":true,
  "hypothesis":"handle_leak(T)",
  "verdict":"proved"
}
true.

?- _T = "no_component_symbols(M)",
   check_hypothesis(_T, [answers(2)], _V),
   verdict_dict(_T, _V, _D),
   json_write_dict(current_output, _D,
                   [width(40), tab(80)]).
{
  "answers": [
    {
      "bindings": {"M":"dbclient"},
      "evidence": [
        "module_info(dbclient,symbols,export)"
      ]
    },
    {
      "bindings": {"M":"cachelib"},
      "evidence": [
        "module_info(cachelib,symbols,none)"
      ]
    }
  ],
```

```
  "answers_shown":2,
  "complete":false,
  "hypothesis":"no_component_symbols(M)",
  "reason":"answer_limit(2)",
  "verdict":"incomplete"
}
true.
```

The `tab(80)` option sets the distance between tab stops beyond any indentation the output reaches, so that the writer indents with spaces; by default it uses a tab character for every eight columns, which a printed transcript cannot show. Three boundary cases are checked below. A binding to a compound term, the cycle `[103,104]`, is a string in the dictionary and the whole response serializes, where a dictionary holding the list itself would have serialized too and one holding a term such as `f(a)` would have raised a type error from the JSON writer; the hypothesis `X = f(a)`, which would produce such a binding, is rejected before any binding is made, and its rejection serializes; and an unbound binding is `null`.

```
?- _T = "deadlock(critical_section, C)",
   check_hypothesis(_T, _V),
   verdict_dict(_T, _V, _D),
   get_dict(answers, _D, [_A]),
   get_dict(bindings, _A, B),
   with_output_to(string(_),
       json_write_dict(current_output, _D)).
B = _{'C':"[103,104]"}.

?- check_hypothesis("X = f(a)", _V),
   verdict_dict("X = f(a)", _V, _D),
   json_write_dict(current_output, _D,
                   [width(40)]).
{
  "hypothesis":"X = f(a)",
  "reason":"unsupported(X=f(a))",
  "verdict":"rejected"
}
true.

?- _T = "\\+ deadlock(mutant, C)",
   check_hypothesis(_T, _V),
   verdict_dict(_T, _V, _D),
   get_dict(answers, _D, [_A]),
   get_dict(bindings, _A, B).
B = _{'C':null}.
```

A verdict of `proved` establishes that the goal follows from the facts of the knowledge base under the definitions of the recognizers, and nothing else. Three qualifications are built into that sentence. The facts are the dump as the debugger reported it and the parser rendered it, with the limits Chapter 15 set out: a stack trace is a reconstruction, a symbol may be coincidental, a truncated dump is silent about what it lacks. The definitions are the book’s formalizations, which Chapters 18 to 28 said at every step are narrower than the catalog’s descriptions; a proof of `deadlock(critical_section, [103, 104])` is a proof that two threads each wait on a critical section the other owns, which is what the predicate says, and the analyst’s judgment that this is the problem, rather than a symptom of it,

is not in the proof. And a verdict of `not_proved` is the failure of a search, not the truth of a negation: under the closed-world assumption of Chapter 17 it means that the facts loaded do not support the hypothesis, which is consistent both with the hypothesis being false and with the facts being incomplete. A verdict of `incomplete` says less still: that the budget ran out, and that the answers it carries, if any, are the first of an unknown number.

What the verifier does establish is worth having. It establishes that the agent's claim is at least consistent with the artifact, which prose cannot; it attaches to the claim the facts that support it, so that an analyst can check them in the debugger in a minute; it refuses claims that the knowledge base has no means to assess, instead of guessing; and it does all this within stated limits. The limits are what make the last point precise. Every hypothesis in the vocabulary either receives a verdict within the inference budget or receives the verdict that the budget was spent, so that the agent can ask about every hypothesis it produces rather than the ones the analyst has time for; on this dump, with the default limits, the verdicts above take a few milliseconds each, and on a dump with thousands of threads the budget is raised and the time grows with it. A workflow in which the agent proposes, Prolog checks, and the analyst reads the evidence puts each of the three where it is strongest. The proofs are the analyst's to read, and the diagnosis remains the analyst's to make.

Appendix A. SWI-Prolog Quick Reference

The built-in and library predicates used in this book, grouped by subject, with the chapter that introduces each. Modes follow the SWI-Prolog manual: `+` an input that must be instantiated, `-` an output, `?` either, `:` a goal or other meta-argument. Library predicates are autoloaded in SWI-Prolog; the library is named for the reader who prefers an explicit `use_module/1`.

A.1 Toplevel and Programs

Predicate	Meaning (chapter)
`consult(+File), [File]`	load a program file (1)
`make`	reload modified files (1)
`halt, halt(+Status)`	exit, with an exit status (1, 13)
`listing(:Spec)`	print the clauses of a predicate (1)
`help(+What), apropos(+Word)`	manual lookup (1)
`include(+File)`	insert a file's clauses textually (19)
`module(+Name, +Exports)`	directive beginning a module (13)
`use_module(+File), use_module(+File, +Imports)`	import a module (13)
`initialization(:Goal, +When)`	run a goal at load or as `main` (13)
`set_prolog_flag(+Flag, +Value), current_prolog_flag(?Flag, ?Value)`	flags such as `double_quotes`, `occurs_check`, `toplevel_print_anon`, `stack_limit` (2, 3, 5, 30)
`dynamic(+Spec), discontiguous(+Spec), multifile(+Spec)`	predicate declarations (11, 27)
`table(+Spec)`	directive enabling tabled execution (11)

A.2 Terms and Types

Predicate	Meaning (chapter)
`var(@T), nonvar(@T)`	unbound or bound (2)
`atom(@T), number(@T), integer(@T), float(@T)`	type tests (2)
`atomic(@T), compound(@T), callable(@T), is_list(@T), string(@T), ground(@T)`	type tests (2)
`functor(?T, ?Name, ?Arity)`	name and arity of a term (8)
`arg(?N, +T, ?Arg)`	the N-th argument (8)
`T =.. L`	term to list and back (8)
`copy_term(+In, -Out)`	copy with fresh variables (3)
`term_variables(+T, -Vars)`	the variables of a term (3)
`compare(?Order, @A, @B), @<, @>, @=<, @>=`	standard order of terms (2)

Predicate	Meaning (chapter)
`write_canonical(+T), print(+T), write_term(+T, +Opts)`	printing terms (2, 8)
`portray_clause(+T)`	print a term as a clause (8, 15)
`numbervars(+T, +Start, -End)`	bind variables to `'$VAR'(N)` (8)

A.3 Unification and Comparison

Predicate	Meaning (chapter)
`A = B, A \= B`	unify, not unifiable (3)
`A == B, A \== B`	identical, not identical (3)
`unify_with_occurs_check(?A, ?B)`	sound unification (3)
`cyclic_term(@T), acyclic_term(@T)`	cyclicity (3)
`?=(@A, @B)`	identity or non-unifiability is decidable now (3)
`subsumes_term(@General, @Specific)`	one-sided matching (3)
`dif(?A, ?B)`	constraint: A and B are different (14)

A.4 Control

Predicate	Meaning (chapter)
`,, ;, ->, *->, \+`	conjunction, disjunction, if-then-else, soft cut, negation as failure (4, 7)
`!`	cut (7)
`true, fail, false`	always succeed, always fail (4)
`once(:G), ignore(:G)`	first solution only; succeed regardless (7)
`forall(:Cond, :Action)`	no instance of Cond without Action (7)
`repeat`	infinite choice point (7)
`between(+L, +H, ?X), succ(?A, ?B), plus(?A, ?B, ?C)`	integer generation and arithmetic relations (6)
`catch(:G, ?Catcher, :Recovery), throw(+Ball)`	exceptions (7)
`setup_call_cleanup(:S, :G, :C)`	guaranteed cleanup (7)
`must_be(+Type, @Term)`	type check that throws (7, 24, 25)
`call_with_inference_limit(:G, +Limit, -Result), findnsols(+N, ?T, :G, -L)`	bounded execution and bounded collection (31)
`call(:G), call(:G, +A1, ...)`	call a goal, with extra arguments (10)
`freeze(?V, :G), frozen(@V, -G), when(+Cond, :G)`	coroutining (14)

A.5 Arithmetic

Predicate	Meaning (chapter)
`X is Expr`	evaluate (6)
`=:=`, `=\=`, `<`, `>`, `=<`, `>=`	arithmetic comparison (6)
`+`, `-`, `*`, `/`, `//`, `mod`, `rem`, `div`, `abs`, `sign`, `min`, `max`, `gcd`, `msb`	functions (6)
`**`, `^`, `sqrt`, `float`, `integer`, `truncate`, `round`, `ceiling`, `floor`	functions (6)
`/\`, `\/`, `xor`, `\`, `<<`, `>>`	bitwise functions (6)
`#=`, `#\=`, `#<`, `#>`, `#=<`, `#>=`	CLP(FD) constraints (14)
`in/2`, `ins/2`, `label/1`, `labeling/2`, `all_different/1`, `all_distinct/1`, `sum/3`, `tuples_in/2`, `#<==>`, `#==>`	CLP(FD) (14)

A.6 Atoms, Strings, and Text

Predicate	Meaning (chapter)
`atom_codes/2`, `atom_chars/2`, `char_code/2`, `number_codes/2`	conversions (8)
`atom_length/2`, `atom_concat/3`, `sub_atom/5`, `atom_number/2`, `atom_string/2`, `upcase_atom/2`, `downcase_atom/2`	atoms (8, 15)
`atomic_list_concat/2`, `atomic_list_concat/3`	join, and split with a separator (8)
`string_concat/3`, `sub_string/5`, `string_length/2`, `string_code/3`, `string_chars/2`, `string_codes/2`	strings (8, 12)
`split_string/4`	split on separators, stripping pad characters (8)
`string_lower/2`, `string_upper/2`, `number_string/2`, `term_string/2`, `term_to_atom/2`	strings (8)
`read_term_from_atom/3`	parse a term from text (8)
`format/1,2,3`	formatted output; `format(atom(A), F, Args)` to text (6, 8)
`with_output_to(+Sink, :G)`	capture output (8)
`re_match/2`, `re_matchsub/3`, `re_foldl/6`	regular expressions, `library(pcre)` (13)

A.7 Lists

Predicate	Meaning (chapter)
`member/2`, `memberchk/2`, `append/3`, `length/2`, `last/2`, `reverse/2`	basic list relations (5, 7)
`nth0/3`, `nth1/3`, `nextto/3`	positional access (5, 18)

Predicate	Meaning (chapter)
`msort/2`, `sort/2`, `sort/4`, `predsort/3`, `keysort/2`	sorting (5, 9)
`sum_list/2`, `max_list/2`, `min_list/2`, `numlist/3`	numeric lists (5)
`select/3`, `subtract/3`, `intersection/3`, `union/3`, `list_to_set/2`, `delete/3`, `exclude/3`, `include/3`, `partition/4`	selection and filtering (5, 10)
`maplist/2..5`, `foldl/4..6`	mapping and folding (10)
`pairs_keys_values/3`, `pairs_keys/2`, `pairs_values/2`, `transpose_pairs/2`, `map_list_to_pairs/3`	`library(pairs)` (9, 17)
`ord_union/3`, `ord_subtract/3`, `ord_intersection/3`, `ord_subset/2`, `ord_disjoint/2`	`library(ordsets)` (24, 26)

A.8 Solutions and Aggregation

Predicate	Meaning (chapter)
`findall/3`, `findall/4`	collect all solutions as copies (9)
`bagof/3`, `setof/3`, `^`	collect with free-variable grouping; sorted set (9)
`aggregate_all/3`, `aggregate/3`	`count`, `sum`, `max`, `min`, `bag`, `set`, `max(X, W)` (9)
`distinct/1,2`, `limit/2`, `offset/2`, `order_by/2`, `group_by/4`, `call_nth/2`	`library(solution_sequences)` (9)

A.9 Data Structures

Predicate	Meaning (chapter)
`empty_assoc/1`, `put_assoc/4`, `get_assoc/3`, `list_to_assoc/2`, `assoc_to_list/2`, `assoc_to_keys/2`, `assoc_to_values/2`	AVL trees, `library(assoc)` (10)
`_{k: v}`, `get_dict/3`, `put_dict/4`, `del_dict/4`, `dict_pairs/3`	dicts (10)
`vertices_edges_to_ugraph/3`, `reachable/3`, `top_sort/2`, `transitive_closure/2`, `transpose_ugraph/2`	`library(ugraphs)` (10)
`empty_nb_set/1`, `add_nb_set/3`, `nb_set_to_list/2`	`library(nb_set)` (30)

A.10 Database and State

Predicate	Meaning (chapter)
`assertz/1`, `asserta/1`, `retract/1`, `retractall/1`	modify the dynamic database (11)
`clause/2`	retrieve clauses (11, 29)
`predicate_property/2`, `current_predicate/1`, `module_property/2`	inspection (11, 29)

Predicate	Meaning (chapter)
`b_setval/2`, `b_getval/2`, `nb_setval/2`, `nb_getval/2`, `flag/3`	global variables and counters (11)
`abolish_all_tables/0`	clear tables (11)

A.11 Grammars

Predicate	Meaning (chapter)
`-->`	grammar rule (12)
`phrase/2`, `phrase/3`	parse a list, with a remainder (12)
`blanks//0`, `blank//0`, `whites//0`, `digits//1`, `integer//1`, `number//1`, `xinteger//1`, `string//1`, `string_without//2`, `nonblanks//1`, `remainder//1`, `eos//0`, `eol//0`	`library(dcg/basics)` (12)
`sequence//2`, `sequence//3`, `sequence//5`	`library(dcg/high_order)` (12)
`phrase_from_file/2`	parse a file as a lazy list (12)

A.12 Files and Interchange

Predicate	Meaning (chapter)
`open/3`, `open/4`, `close/1`, `read_term/2`, `read_term/3`, `read_line_to_string/2`, `read_line_to_codes/2`	streams (13)
`csv_read_file/2,3`, `csv_write_file/2`	`library(csv)` (13, 16)
`json_read_dict/2`, `json_write_dict/2`, `atom_json_dict/3`	`library(http/json)` (13, 31)
`main/0`, `argv_options/3`	`library(main)` scripts (13, 31)
`qcompile/1`	compile a file to `.qlf` (30)

A.13 Debugging, Testing, and Measurement

Predicate	Meaning (chapter)
`trace/0`, `notrace/0`, `spy/1`, `nospy/1`, `nodebug/0`	the tracer (4)
`debug/3`, `debug/1`, `nodebug/1`, `assertion/1`	`library(debug)` (13)
`begin_tests/1`, `end_tests/1`, `test/2`, `run_tests/0`	`library(plunit)` (13, 17)
`time/1`, `statistics/2`, `profile/1`	measurement (13, 30)
`jiti_list/1`	show just-in-time indexes (13, 16, 30)
`print_message/2`, `catch_with_backtrace/3`	messages and backtraces (7, 13)
`print_term/2`	pretty printing, `library(pprint)` (18)

Appendix B. The Shared Artifacts

The three files listed here are consulted throughout the book. The first two are the shared artifacts; the third is the module of canonical recognizers that Chapters 23, 29, and 31 load. Both are synthetic: `dump1.pl` is modelled on what a Windows debugger reports for a 64-bit user-mode process, and `trace1.pl` on an application log, but every value in them is invented, and the thread and module names were chosen for legibility rather than fidelity. The files are reproduced in full so that every transcript in the book can be reproduced from the text alone. They are also in the book's repository, github.com/DumpAnalysis/PrologForSoftwareDiagnostics, with every program block of every chapter as a file and the harness that replays the transcripts. The first two are plain Prolog source files and load with `consult/1`; they load together without conflict, because they define disjoint predicates. The third is a module, loaded with `use_module/1` after `dump1.pl`, whose clauses are those of Chapters 18–22; it calls the dump relations in the `user` module and does not consult the dump itself. A fourth file is in the package and not listed: `wd.pl`, the Prolog module of the PrologWinDbg extension that Section 31.7 runs against a synthetic snapshot, reproduced verbatim from the extension's repository at the commit that section cites, so that it is the repository's text and not the book's.

B.1 dump1.pl

```
% dump1.pl -- a synthetic user-space process memory dump
% rendered as Prolog facts.  Used throughout the book.
%
% The data is modelled on what a Windows debugger shows
% for a 64-bit process, but every value is invented.

:- discontiguous frame/5.

% process(PID, Name)
process(4120, 'AppServer.exe').

% module(Name, Base, Size)
module(ntdll, 0x7ffb_b2a0_0000, 0x1f8000).
module(kernelbase, 0x7ffb_b0b0_0000, 0x2f1000).
module(kernel32, 0x7ffb_b1c0_0000, 0xbd000).
module(ucrtbase, 0x7ffb_ae50_0000, 0x100000).
module(msvcp140, 0x7ffb_9c20_0000, 0x9f000).
module(appserver, 0x7ff6_1d10_0000, 0x3c000).
module(reportengine, 0x7ffb_8e40_0000, 0x1a2000).
module(dbclient, 0x7ffb_8d10_0000, 0x6e000).
module(cachelib, 0x7ffb_7a00_0000, 0x4a000).
module(thirdparty_hook,0x7ffb_6f30_0000, 0x12000).

% module_info(Name, Key, Value)
module_info(appserver, version, '3.2.1.0').
module_info(reportengine, version, '3.2.1.0').
module_info(dbclient, version, '3.1.0.7').
module_info(cachelib, version, '2.0.0.0').
module_info(thirdparty_hook, version, '1.0.0.1').
module_info(ntdll, symbols, pdb).
module_info(kernelbase, symbols, pdb).
```

```
module_info(kernel32, symbols, pdb).
module_info(ucrtbase, symbols, pdb).
module_info(msvcp140, symbols, pdb).
module_info(appserver, symbols, pdb).
module_info(reportengine, symbols, pdb).
module_info(dbclient, symbols, export).
module_info(cachelib, symbols, none).
module_info(thirdparty_hook, symbols, none).
module_info(thirdparty_hook, company, 'Unknown').
module_info(appserver, company, 'Example Corp').
module_info(reportengine, company, 'Example Corp').
module_info(dbclient, company, 'Example Corp').
module_info(cachelib, company, 'CacheWorks').

% thread(TID, PID)
thread(100, 4120).
thread(101, 4120).
thread(102, 4120).
thread(103, 4120).
thread(104, 4120).
thread(105, 4120).
thread(106, 4120).
thread(107, 4120).
thread(108, 4120).
thread(109, 4120).

% thread_time(TID, UserMs, KernelMs)
thread_time(100, 120, 40).
thread_time(101, 15, 5).
thread_time(102, 8450, 1230).
thread_time(103, 30, 10).
thread_time(104, 25, 10).
thread_time(105, 20, 5).
thread_time(106, 10, 5).
thread_time(107, 3200, 900).
thread_time(108, 0, 0).
thread_time(109, 40, 20).

% thread_state(TID, State)
thread_state(100, waiting(user_request)).
thread_state(101, waiting(user_request)).
thread_state(102, running).
thread_state(103, waiting(user_request)).
thread_state(104, waiting(user_request)).
thread_state(105, waiting(user_request)).
thread_state(106, waiting(user_request)).
thread_state(107, running).
thread_state(108, suspended).
thread_state(109, waiting(exception)).
```

```
% frame(TID, N, Module, Function, Offset)
% thread 100: main thread, waiting in the message loop
frame(100, 0, ntdll, 'NtWaitForMultipleObjects', 0x14).
frame(100, 1, kernelbase, 'WaitForMultipleObjectsEx', 0xe9).
frame(100, 2, appserver, 'MainLoop', 0x1b2).
frame(100, 3, appserver, 'wmain', 0x55).
frame(100, 4, appserver, 'invoke_main', 0x22).
frame(100, 5, kernel32, 'BaseThreadInitThunk', 0x14).
frame(100, 6, ntdll, 'RtlUserThreadStart', 0x21).

% thread 101: worker blocked on a critical section
frame(101, 0, ntdll, 'NtWaitForAlertByThreadId', 0x14).
frame(101, 1, ntdll, 'RtlpWaitOnCriticalSection',0x1cf).
frame(101, 2, ntdll, 'RtlpEnterCriticalSectionContended',
      0x1cd).
frame(101, 3, ntdll, 'RtlEnterCriticalSection', 0x40).
frame(101, 4, cachelib, 'Unknown', 0x2a10).
frame(101, 5, appserver, 'Session::Lookup', 0x7c).
frame(101, 6, appserver, 'Worker::Run', 0x120).
frame(101, 7, ucrtbase, 'thread_start<...>', 0x42).
frame(101, 8, kernel32, 'BaseThreadInitThunk', 0x14).
frame(101, 9, ntdll, 'RtlUserThreadStart', 0x21).

% thread 102: spiking thread, busy in a tight loop
frame(102, 0, reportengine, 'Table::Find', 0x3e).
frame(102, 1, reportengine, 'Table::Insert', 0x9a).
frame(102, 2, reportengine, 'Report::Build', 0x214).
frame(102, 3, appserver, 'Worker::Run', 0x120).
frame(102, 4, ucrtbase, 'thread_start<...>', 0x42).
frame(102, 5, kernel32, 'BaseThreadInitThunk', 0x14).
frame(102, 6, ntdll, 'RtlUserThreadStart', 0x21).

% thread 103: owner of CS A, waiting for CS B (deadlock half)
frame(103, 0, ntdll, 'NtWaitForAlertByThreadId', 0x14).
frame(103, 1, ntdll, 'RtlpWaitOnCriticalSection',0x1cf).
frame(103, 2, ntdll, 'RtlpEnterCriticalSectionContended',
      0x1cd).
frame(103, 3, ntdll, 'RtlEnterCriticalSection', 0x40).
frame(103, 4, dbclient, 'Connection::Send', 0x58).
frame(103, 5, appserver, 'Session::Commit', 0x9e).
frame(103, 6, appserver, 'Worker::Run', 0x120).
frame(103, 7, ucrtbase, 'thread_start<...>', 0x42).
frame(103, 8, kernel32, 'BaseThreadInitThunk', 0x14).
frame(103, 9, ntdll, 'RtlUserThreadStart', 0x21).

% thread 104: owner of CS B, waiting for CS A (deadlock half)
frame(104, 0, ntdll, 'NtWaitForAlertByThreadId', 0x14).
frame(104, 1, ntdll, 'RtlpWaitOnCriticalSection',0x1cf).
frame(104, 2, ntdll, 'RtlpEnterCriticalSectionContended',
      0x1cd).
frame(104, 3, ntdll, 'RtlEnterCriticalSection', 0x40).
frame(104, 4, appserver, 'Session::Lookup', 0x7c).
frame(104, 5, dbclient, 'Connection::OnReply', 0x1f).
frame(104, 6, dbclient, 'Reader::Loop', 0xa4).
frame(104, 7, ucrtbase, 'thread_start<...>', 0x42).
frame(104, 8, kernel32, 'BaseThreadInitThunk', 0x14).
frame(104, 9, ntdll, 'RtlUserThreadStart', 0x21).
```

```
% thread 105: thread pool, idle
frame(105, 0, ntdll, 'NtWaitForWorkViaWorkerFactory', 0x14).
frame(105, 1, ntdll, 'TppWorkerThread', 0x2df).
frame(105, 2, kernel32, 'BaseThreadInitThunk', 0x14).
frame(105, 3, ntdll, 'RtlUserThreadStart', 0x21).

% thread 106: thread pool, idle
frame(106, 0, ntdll, 'NtWaitForWorkViaWorkerFactory', 0x14).
frame(106, 1, ntdll, 'TppWorkerThread', 0x2df).
frame(106, 2, kernel32, 'BaseThreadInitThunk', 0x14).
frame(106, 3, ntdll, 'RtlUserThreadStart', 0x21).

% thread 107: deep recursion (stack overflow candidate)
frame(107, 0, reportengine, 'Node::Visit', 0x12).
frame(107, 1, reportengine, 'Node::Visit', 0x6b).
frame(107, 2, reportengine, 'Node::Visit', 0x6b).
frame(107, 3, reportengine, 'Node::Visit', 0x6b).
frame(107, 4, reportengine, 'Node::Visit', 0x6b).
frame(107, 5, reportengine, 'Node::Visit', 0x6b).
frame(107, 6, reportengine, 'Node::Visit', 0x6b).
frame(107, 7, reportengine, 'Node::Visit', 0x6b).
frame(107, 8, reportengine, 'Tree::Walk', 0x30).
frame(107, 9, reportengine, 'Report::Build', 0x1f0).
frame(107, 10, appserver, 'Worker::Run', 0x120).
frame(107, 11, ucrtbase, 'thread_start<...>', 0x42).
frame(107, 12, kernel32, 'BaseThreadInitThunk', 0x14).
frame(107, 13, ntdll, 'RtlUserThreadStart', 0x21).

% thread 108: suspended, no meaningful frames
frame(108, 0, ntdll, 'NtDelayExecution', 0x14).
frame(108, 1, kernelbase, 'SleepEx', 0x9e).
frame(108, 2, appserver, 'Housekeeping::Run', 0x3a).
frame(108, 3, kernel32, 'BaseThreadInitThunk', 0x14).
frame(108, 4, ntdll, 'RtlUserThreadStart', 0x21).

% thread 109: the faulting thread
frame(109, 0, dbclient, 'Buffer::Copy', 0x1a).
frame(109, 1, dbclient, 'Connection::Receive', 0x8c).
frame(109, 2, appserver, 'Session::Fetch', 0x44).
frame(109, 3, appserver, 'Worker::Run', 0x120).
frame(109, 4, ucrtbase, 'thread_start<...>', 0x42).
frame(109, 5, kernel32, 'BaseThreadInitThunk', 0x14).
frame(109, 6, ntdll, 'RtlUserThreadStart', 0x21).

% exception_record(TID, Code, Address)
exception_record(109, 0xc0000005, 0x7ffb_8d10_31aa).

% exception_info(TID, Key, Value)
exception_info(109, access, read).
exception_info(109, target, 0x0000_0000_0000_0018).

% critical_section(Addr, OwnerTID, WaiterCount, RecursionCount)
critical_section(0x7ff6_1d13_a0c0, 103, 2, 1).
critical_section(0x7ff6_1d13_a140, 104, 1, 1).
critical_section(0x7ff6_1d13_a1c0, 0, 0, 0).
```

```
% waits_for(TID, Kind, Addr)
waits_for(101, critical_section, 0x7ff6_1d13_a0c0).
waits_for(103, critical_section, 0x7ff6_1d13_a140).
waits_for(104, critical_section, 0x7ff6_1d13_a0c0).

% handle_count(PID, Type, Count)
handle_count(4120, event, 12040).
handle_count(4120, file, 38).
handle_count(4120, thread, 10).
handle_count(4120, mutant, 4).
handle_count(4120, section, 6).
handle_count(4120, key, 14).

% heap(HeapAddr, ReservedKb, CommittedKb, FreeKb)
heap(0x0000_0210_a000_0000, 2097152, 1835008, 24576).
heap(0x0000_0210_a001_0000, 1024, 256, 128).
heap(0x0000_0210_a200_0000, 65536, 61440, 1024).

% last_error(TID, Code)
last_error(100, 0).
last_error(101, 0).
last_error(102, 0).
last_error(103, 0).
last_error(104, 0).
last_error(109, 0x3e6).   % ERROR_NOACCESS

% hooked(Function, Module, Hooker)
hooked('CreateFileW', kernelbase, thirdparty_hook).
hooked('ReadFile', kernelbase, thirdparty_hook).
```

B.2 trace1.pl

```
% trace1.pl -- a synthetic software trace.
% Used throughout the book. Values are invented.
%
% msg(N, Time, PID, TID, Component, Text)
%   N          message number (1-based)
%   Time       seconds from trace start (float)
%   PID, TID   process and thread identifiers
%   Component  atom naming the source component
%   Text       the message text (a string)

msg(1,  0.000, 4120, 100, appserver,
    "Service starting, version 3.2.1.0").
msg(2,  0.004, 4120, 100, appserver,
    "Loading configuration C:\\App\\app.cfg").
msg(3,  0.011, 4120, 100, cachelib,
    "Cache initialized, capacity=4096").
msg(4,  0.015, 4120, 100, dbclient,
    "Connecting to db01:5432").
msg(5,  0.212, 4120, 100, dbclient,
    "Connected to db01:5432, session=71").
msg(6,  0.214, 4120, 100, appserver,
    "Worker pool started, threads=4").
msg(7,  0.215, 4120, 101, appserver,
    "Worker 101 ready").
msg(8,  0.215, 4120, 102, appserver,
```

```
    "Worker 102 ready").
msg(9,  0.216, 4120, 103, appserver,
    "Worker 103 ready").
msg(10, 0.216, 4120, 104, appserver,
    "Worker 104 ready").
msg(11, 1.002, 4120, 100, appserver,
    "Request 1 received: GET /report/17").
msg(12, 1.003, 4120, 101, appserver,
    "Request 1 dispatched to worker 101").
msg(13, 1.004, 4120, 101, dbclient,
    "Query start: SELECT report 17").
msg(14, 1.048, 4120, 101, dbclient,
    "Query end: 1 row, 44 ms").
msg(15, 1.049, 4120, 101, reportengine,
    "Build start: report 17").
msg(16, 1.310, 4120, 101, reportengine,
    "Build end: report 17, 2611 nodes, 261 ms").
msg(17, 1.311, 4120, 101, appserver,
    "Request 1 completed: 200, 309 ms").
msg(18, 1.500, 4120, 100, appserver,
    "Request 2 received: GET /report/18").
msg(19, 1.501, 4120, 102, appserver,
    "Request 2 dispatched to worker 102").
msg(20, 1.502, 4120, 102, dbclient,
    "Query start: SELECT report 18").
msg(21, 1.550, 4120, 102, dbclient,
    "Query end: 1 row, 48 ms").
msg(22, 1.551, 4120, 102, reportengine,
    "Build start: report 18").
msg(23, 2.000, 4120, 100, appserver,
    "Request 3 received: POST /session/commit").
msg(24, 2.001, 4120, 103, appserver,
    "Request 3 dispatched to worker 103").
msg(25, 2.002, 4120, 103, appserver,
    "Session 9 commit start").
msg(26, 2.003, 4120, 103, dbclient,
    "Send start: session=71, bytes=1024").
msg(27, 2.100, 4120, 100, appserver,
    "Request 4 received: GET /session/9").
msg(28, 2.101, 4120, 104, appserver,
    "Request 4 dispatched to worker 104").
msg(29, 2.102, 4120, 104, dbclient,
    "Reply received: session=71, bytes=512").
msg(30, 2.103, 4120, 104, appserver,
    "Session 9 lookup start").
msg(31, 2.500, 4120, 102, reportengine,
    "Warning: table resize, 8192 -> 16384").
msg(32, 3.000, 4120, 102, reportengine,
    "Warning: table resize, 16384 -> 32768").
msg(33, 3.500, 4120, 102, reportengine,
    "Warning: table resize, 32768 -> 65536").
msg(34, 4.000, 4120, 102, reportengine,
    "Warning: table resize, 65536 -> 131072").
msg(35, 4.500, 4120, 102, reportengine,
    "Warning: table resize, 131072 -> 262144").
msg(36, 5.000, 4120, 102, reportengine,
    "Warning: table resize, 262144 -> 524288").
msg(37, 5.001, 4120, 100, appserver,
    "Request 5 received: GET /report/19").
```

```
msg(38, 5.002, 4120, 100, appserver,
    "No idle worker, request 5 queued").
msg(39, 6.000, 4120, 100, appserver,
    "Request 6 received: GET /health").
msg(40, 6.001, 4120, 100, appserver,
    "No idle worker, request 6 queued").
msg(41, 7.000, 4120, 100, appserver,
    "Request 7 received: GET /health").
msg(42, 7.001, 4120, 100, appserver,
    "No idle worker, request 7 queued").
msg(43, 7.002, 4120, 100, appserver,
    "Error: queue depth 3 exceeds threshold 2").
msg(44, 8.000, 4120, 100, appserver,
    "Request 8 received: GET /health").
msg(45, 8.001, 4120, 100, appserver,
    "Error: queue depth 4 exceeds threshold 2").
msg(46, 9.000, 4120, 100, appserver,
    "Request 9 received: GET /health").
msg(47, 9.001, 4120, 100, appserver,
    "Error: queue depth 5 exceeds threshold 2").
msg(48, 9.002, 4120, 108, appserver,
    "Housekeeping: watchdog, workers busy=4").
msg(49, 9.003, 4120, 108, appserver,
    "Housekeeping: worker 103 wait 7001 ms").
msg(50, 9.003, 4120, 108, appserver,
    "Housekeeping: worker 104 wait 6900 ms").
msg(51, 9.004, 4120, 108, appserver,
    "Housekeeping: worker 102 cpu 100%").
msg(52, 9.005, 4120, 108, appserver,
    "Housekeeping: worker 101 idle").
msg(53, 9.006, 4120, 101, appserver,
    "Request 5 dispatched to worker 101").
msg(54, 9.007, 4120, 101, dbclient,
    "Query start: SELECT report 19").
msg(55, 9.060, 4120, 101, dbclient,
    "Query end: 1 row, 53 ms").
msg(56, 9.061, 4120, 101, reportengine,
    "Build start: report 19").
msg(57, 9.300, 4120, 101, reportengine,
    "Build end: report 19, 2402 nodes, 239 ms").
msg(58, 9.301, 4120, 101, appserver,
    "Request 5 completed: 200, 4300 ms").
msg(59, 9.302, 4120, 101, appserver,
    "Request 6 dispatched to worker 101").
msg(60, 9.303, 4120, 101, appserver,
    "Request 6 completed: 200, 3302 ms").
msg(61, 9.304, 4120, 101, appserver,
    "Request 7 dispatched to worker 101").
msg(62, 9.305, 4120, 101, appserver,
    "Request 7 completed: 200, 2305 ms").
msg(63, 9.306, 4120, 101, appserver,
    "Request 8 dispatched to worker 101").
msg(64, 9.307, 4120, 101, appserver,
    "Request 8 completed: 200, 1307 ms").
msg(65, 9.308, 4120, 101, appserver,
    "Request 9 dispatched to worker 101").
msg(66, 9.309, 4120, 101, appserver,
    "Request 9 completed: 200, 309 ms").
msg(67, 14.000, 4120, 109, appserver,
```

```
    "Request 10 received: GET /session/9/data").
msg(68, 14.001, 4120, 109, dbclient,
    "Receive start: session=71").
msg(69, 14.002, 4120, 109, dbclient,
    "Error: buffer copy failed, code=0x3e6").
msg(70, 14.003, 4120, 109, appserver,
    "Unhandled exception 0xc0000005, thread 109").
msg(71, 14.010, 4120, 100, appserver,
    "Service stopping").

?- aggregate_all(count, frame(_, _, _, _, _), F),
   aggregate_all(count, msg(_, _, _, _, _, _), M).
F = 78,
M = 71.
```

B.3 patterns_dump.pl

```
% patterns_dump.pl -- the recognizers of Chapters 18 to 22
% that the report of Chapter 23 composes, collected in one
% module with the clauses of those chapters.
%
% The module does not consult dump1.pl: a file without a
% module declaration is loaded into one module only
% (Chapter 13), and the chapters load the dump into user.
% Every reference to a relation of the dump schema is
% therefore qualified with user:, and the module is loaded
% after the dump with :- use_module(patterns_dump).
%
% The thresholds and the analyst's list of system modules
% are facts of this module, exported, multifile and dynamic,
% so that a chapter can add a clause or change a value.
% The extension relations that Chapters 20 and 21 define
% in-chapter are declared dynamic in user, so that a
% knowledge base without them fails rather than raising an
% existence error.
%
% Every predicate that can appear as a leaf of an evidence
% or suspicion term is exported, so that the leaves are
% callable, unqualified, from any module that imports this
% one (Section 17.3). The last group of exports checks that
% contract: derivable/2 and sufficient/1 re-derive a finding
% from its own leaves, and check_explainers/0 runs every
% explainer of explainer/2 on the loaded knowledge base and
% calls the leaves in the module that calls it. The
% derivation has a closed vocabulary: dump_relation/1 names
% the relations it takes from the evidence list only, and
% evaluable/1 the first-order built-ins it calls; every
% other goal is interpreted, expanded through its clauses,
% or rejected with a domain error.

:- module(patterns_dump,
          [ threshold/2,
            system_module/1,
            % Chapter 18
            stack_trace/2,
            stack_trace_signature/2,
            runs/2,
```

```
            main_thread/1,
            cpu_time/2,
            total_cpu/1,
            cpu_share/2,
            spiking_thread/1,
            spiking_thread/2,
            blocked_thread/1,
            blocked_thread/2,
            frame_regularity/3,
            % Chapters 1 and 23
            suspended_thread/1,
            % Chapter 19
            exception_stack_trace/2,
            null_pointer_data/1,
            null_pointer_data/2,
            stack_depth/2,
            stack_overflow/2,
            % Chapter 20
            waits/3,
            wait_edge/3,
            wait_edge/4,
            edge_of/3,
            edge_of/4,
            wait_chain/2,
            deadlock_witness/3,
            cycle_kinds/2,
            deadlock/2,
            deadlock/3,
            % Chapter 21
            insufficient_memory/2,
            insufficient_memory/3,
            handle_leak/1,
            memory_leak/1,
            % Chapter 22
            main_module/1,
            vendor/2,
            not_my_version/1,
            unknown_component/1,
            no_component_symbols/1,
            hooked_functions/3,
            last_error_collection/1,
            % Chapters 17 and 23: the evidence contract
            explainer/2,
            dump_relation/1,
            evaluable/1,
            derivable/2,
            sufficient/1,
            check_finding/3,
            check_explainers/0
          ]).

:- multifile threshold/2, system_module/1.
:- dynamic threshold/2, system_module/1.
```

```
% The library predicates the module uses are imported
% by name, so that a definition of the same name in
% user, the module's default import module, cannot
% stand in for them (Chapter 13).
:- use_module(library(lists)).
:- use_module(library(apply)).
:- use_module(library(pairs)).
:- use_module(library(aggregate)).
:- use_module(library(error)).

:- dynamic user:mutant/3, user:waits_for_object/3.
:- dynamic user:heap_at/5, user:dump_time/2.

% threshold(Name, Value): the thresholds of Chapters 18 to
% 21, as those chapters state them. frame_run is the run
% length of a Frame Regularity (Chapter 18); overflow_run
% is the longer run that makes a deep stack a suspected
% Stack Overflow (User Mode) (Chapter 19).
threshold(spiking_share, 25).
threshold(frame_run, 4).
threshold(overflow_run, 6).
threshold(null_page, 0x1_0000).
threshold(stack_depth, 12).
threshold(committed_ratio, 0.85).
threshold(large_heap_kb, 1048576).
threshold(committed_total_kb, 1572864).
threshold(handle_type_count, 5000).
threshold(handle_total_count, 10000).
threshold(leak_growth_kb, 65536).

% system_module(Module): a module supplied by the
% operating system or the C runtime. The analyst's
% knowledge, as in Chapter 22.
system_module(ntdll).
system_module(kernelbase).
system_module(kernel32).
system_module(ucrtbase).
system_module(msvcp140).


% ---- Chapter 18: stack traces and threads ----

% stack_trace(TID, Frames): Frames is the list of
% frame/5 terms of thread TID, top frame first.
stack_trace(TID, Frames) :-
    findall(frame(TID, N, M, F, O),
            user:frame(TID, N, M, F, O), Frames0),
    sort(2, @=<, Frames0, Frames).

% entry_function(F): F is an entry function that
% the main thread's stack trace passes through.
entry_function(main).
entry_function(wmain).
entry_function('WinMain').
entry_function('wWinMain').
```

```
% main_thread(TID): TID is the main thread.
main_thread(TID) :-
    user:thread(TID, _),
    once(( user:frame(TID, _, _, F, _),
           entry_function(F)
         )).

% cpu_time(TID, Ms): the CPU time of TID in ms.
cpu_time(TID, Ms) :-
    user:thread_time(TID, User, Kernel),
    Ms is User + Kernel.

% total_cpu(Total): the CPU time of all threads.
total_cpu(Total) :-
    aggregate_all(sum(Ms), cpu_time(_, Ms),
                  Total).

% cpu_share(TID, Share): the CPU time of TID as a
% percentage of the total.
cpu_share(TID, Share) :-
    cpu_time(TID, Ms),
    total_cpu(Total),
    Total > 0,
    Share is 100.0 * Ms / Total.

% spiking_thread(TID): the CPU share of TID is at
% least the threshold.
spiking_thread(TID) :-
    cpu_share(TID, Share),
    threshold(spiking_share, Min),
    Share >= Min.

% spiking_thread(TID, Evidence): as the test
% above, with the evidence of the recognition.
spiking_thread(TID, Evidence) :-
    spiking_thread(TID),
    cpu_time(TID, Ms),
    total_cpu(Total),
    threshold(spiking_share, Min),
    Evidence = evidence('Spiking Thread', TID,
                        [ cpu_time(TID, Ms),
                          total_cpu(Total),
                          threshold(spiking_share,
                                    Min)
                        ]).

% blocked_thread(TID): the dump records an object
% TID waits for.
blocked_thread(TID) :-
    user:waits_for(TID, _, _).

% wait_function(F): F is a frame of the critical
% section wait path in ntdll.
wait_function('RtlpWaitOnCriticalSection').
wait_function('RtlEnterCriticalSection').

% blocked_thread(TID, Evidence): as
% blocked_thread/1, with the object and the wait
% frames as evidence.
```

```
blocked_thread(TID, Evidence) :-
    user:waits_for(TID, Kind, Addr),
    findall(frame(TID, N, M, F, O),
            ( user:frame(TID, N, M, F, O),
              wait_function(F)
            ),
            Frames),
    Facts = [waits_for(TID, Kind, Addr) | Frames],
    Name = 'Blocked Thread (Software)',
    Evidence = evidence(Name, TID, Facts).

% frame_key(Frame, Key): Key is the pair
% Module-Function of the frame/5 term Frame.
frame_key(frame(_, _, M, F, _), M-F).

% stack_trace_signature(TID, Signature): the keys
% of the frames of TID, top frame first.
stack_trace_signature(TID, Signature) :-
    stack_trace(TID, Frames),
    maplist(frame_key, Frames, Signature).

% runs(List, Runs): Runs is the list of X-N pairs,
% one per maximal run of N consecutive elements
% equal to X in List.
runs([], []).
runs([X | Xs], [X-N | Runs]) :-
    run_length(Xs, X, 1, N, Rest),
    runs(Rest, Runs).

run_length(Xs, X, N0, N, Rest) :-
    (   Xs = [Y | Xs1],
        Y == X
    ->  N1 is N0 + 1,
        run_length(Xs1, X, N1, N, Rest)
    ;   N = N0,
        Rest = Xs
    ).

% frame_regularity(TID, Key, Count): the signature
% of TID has a run of Count frames with Key, Count
% at least the threshold.
frame_regularity(TID, Key, Count) :-
    user:thread(TID, _),
    stack_trace_signature(TID, Signature),
    runs(Signature, Runs),
    member(Key-Count, Runs),
    threshold(frame_run, Min),
    Count >= Min.


% ---- Chapters 1 and 23: thread state ----

% suspended_thread(TID): TID is suspended.
suspended_thread(TID) :-
    user:thread_state(TID, suspended).


% ---- Chapter 19: exceptions and faults ----
```

```
% exception_stack_trace(TID, Frames): TID has an
% exception record and Frames is its stack trace.
exception_stack_trace(TID, Frames) :-
    user:exception_record(TID, _, _),
    stack_trace(TID, Frames).

% access_violation(TID): the exception of TID is
% an access violation.
access_violation(TID) :-
    user:exception_record(TID, 0xc0000005, _).

% null_pointer_data(TID): TID read or wrote
% through a null pointer.
null_pointer_data(TID) :-
    access_violation(TID),
    user:exception_info(TID, access, Access),
    Access \== execute,
    user:exception_info(TID, target, Target),
    threshold(null_page, Limit),
    Target < Limit.

% null_pointer_data(TID, Evidence): as above, with
% the facts the recognition rests on: the record,
% its parameters, and the null page limit.
null_pointer_data(TID, Evidence) :-
    null_pointer_data(TID),
    user:exception_record(TID, Code, Addr),
    findall(exception_info(TID, K, V),
            user:exception_info(TID, K, V), Info),
    threshold(null_page, Limit),
    Facts = [exception_record(TID, Code, Addr)
             | Rest],
    append(Info, [threshold(null_page, Limit)],
           Rest),
    Evidence = evidence('NULL Pointer (Data)',
                        TID, Facts).

% stack_depth(TID, Depth): the stack trace of TID
% has Depth frames.
stack_depth(TID, Depth) :-
    stack_trace(TID, Frames),
    length(Frames, Depth).

% stack_overflow(TID, Finding): Finding is the
% evidence that TID overflowed its stack, by the
% status code, or the suspicion of it, by depth
% and a run at least overflow_run long.
stack_overflow(TID, Finding) :-
    user:exception_record(TID, 0xc00000fd, Addr),
    Name = 'Stack Overflow (User Mode)',
    Code = 0xc00000fd,
    Facts = [exception_record(TID, Code, Addr)],
    Finding = evidence(Name, TID, Facts).
stack_overflow(TID, Finding) :-
    user:thread(TID, PID),
    stack_depth(TID, Depth),
    threshold(stack_depth, MinDepth),
    Depth >= MinDepth,
    frame_regularity(TID, Key, Count),
```

```
    threshold(overflow_run, MinRun),
    Count >= MinRun,
    Name = 'Stack Overflow (User Mode)',
    Grounds = [ thread(TID, PID),
                stack_depth(TID, Depth),
                threshold(stack_depth, MinDepth),
                frame_regularity(TID, Key, Count),
                threshold(overflow_run, MinRun)
              ],
    Finding = suspected(Name, TID, Grounds).


% ---- Chapter 20: synchronization ----

% wait_fact(TID, Kind, Object, Fact): Fact is the
% fact, of the shared file or of the extension,
% that records that TID waits for Object of Kind.
wait_fact(T, K, O, waits_for(T, K, O)) :-
    user:waits_for(T, K, O).
wait_fact(T, K, O, waits_for_object(T, K, O)) :-
    user:waits_for_object(T, K, O).

% waits(TID, Kind, Object): TID waits for Object
% of Kind, by the shared facts or the extension.
waits(TID, Kind, Object) :-
    wait_fact(TID, Kind, Object, _).

% wait_edge(Kind, Waiter, Owner, Facts): Waiter
% waits for an object of Kind that Owner holds;
% Facts are the wait fact and the ownership fact
% that say so.
wait_edge(critical_section, W, O,
          [Wait, critical_section(A, O, L, R)]) :-
    wait_fact(W, critical_section, A, Wait),
    user:critical_section(A, O, L, R),
    O =\= 0.
wait_edge(mutant, W, O,
          [Wait, mutant(H, O, N)]) :-
    wait_fact(W, mutant, H, Wait),
    user:mutant(H, O, N),
    O =\= 0.
wait_edge(thread, W, O, [Wait]) :-
    wait_fact(W, thread, O, Wait).

% wait_edge(Kind, Waiter, Owner): the edge without
% its facts.
wait_edge(Kind, W, O) :-
    wait_edge(Kind, W, O, _).
```

```
% edge_of(+Kind, ?W, ?O, -Edge): a wait edge of
% Kind, or of any kind when Kind is the atom any;
% Edge is K-Facts, the kind of the edge and the
% facts that make it.
edge_of(Kind, W, O, K-Facts) :-
    must_be(oneof([critical_section, mutant,
                   thread, any]), Kind),
    (   Kind == any
    ->  wait_edge(K, W, O, Facts)
    ;   K = Kind,
        wait_edge(K, W, O, Facts)
    ).

% edge_of(+Kind, ?W, ?O): the edge without its
% kind and facts.
edge_of(Kind, W, O) :-
    edge_of(Kind, W, O, _).

% wait_chain(+Kind, -Chain): Chain is the path of
% Kind edges from a waiting thread, followed until
% the next thread would repeat one in the chain.
wait_chain(Kind, [T | Rest]) :-
    edge_of(Kind, T, _),
    follow(Kind, T, [T], Rest).

follow(Kind, T, Visited, Chain) :-
    (   edge_of(Kind, T, Next),
        \+ memberchk(Next, Visited)
    ->  Chain = [Next | Rest],
        follow(Kind, Next, [Next | Visited], Rest)
    ;   Chain = []
    ).

% deadlock_witness(+Kind, -Cycle, -Edges): Cycle
% is a list of threads each waiting, through an
% object of Kind, for the next, and the last for
% the first, the first the smallest; Edges holds,
% in the same order, the edge each thread was
% followed along: its kind and the facts that make
% it, as edge_of/4 selected them.
deadlock_witness(Kind, [T | Path],
                 [Edge | Edges]) :-
    edge_of(Kind, T, Next, Edge),
    cycle_from(Kind, T, Next, [T], Path, Edges),
    min_list([T | Path], T).

% cycle_from(Kind, Start, Current, Visited, Path,
% Edges): Path leads from Current back to Start
% along Kind edges without repeating a thread, and
% Edges are the edges followed.
cycle_from(Kind, Start, Current, Visited,
           Path, Edges) :-
    (   Current == Start
    ->  Path = [],
        Edges = []
    ;   \+ memberchk(Current, Visited),
        Path = [Current | Rest],
        Edges = [Edge | Rest1],
        edge_of(Kind, Current, Next, Edge),
```

```
        cycle_from(Kind, Start, Next,
                   [Current | Visited],
                   Rest, Rest1)
    ).

% deadlock(+Kind, -Cycle): Cycle is a cycle of
% Kind edges, from its smallest thread; the
% witness is dropped.
deadlock(Kind, Cycle) :-
    deadlock_witness(Kind, Cycle, _).

% cycle_kinds(Edges, Kinds): Kinds is the set of
% kinds of the edges Edges of a witness.
cycle_kinds(Edges, Kinds) :-
    pairs_keys(Edges, Kinds0),
    sort(Kinds0, Kinds).

% deadlock_name(Kinds, Name): the catalog name of
% a deadlock whose selected edges have the set of
% kinds Kinds: the critical section variant, the
% mixed variant for two kinds or more, and the
% family name for one kind of another object.
deadlock_name([critical_section],
              'Deadlock (Critical Sections)').
deadlock_name([Kind], 'Deadlock') :-
    Kind \== critical_section.
deadlock_name([_, _ | _],
              'Deadlock (Mixed Objects)').

% deadlock(+Kind, -Cycle, -Evidence): as the test
% above; the evidence holds the facts of the edges
% the walk selected, the wait fact and the
% ownership fact of each, and the pattern is named
% from the kinds of those edges. Under a variant
% name the instance is the cycle; under the family
% name it is cycle(Kind, Cycle), so that the kind
% of object is reported.
deadlock(Kind, Cycle, Evidence) :-
    deadlock_witness(Kind, Cycle, Edges),
    cycle_kinds(Edges, Kinds),
    deadlock_name(Kinds, Name),
    (   Kinds = [One],
        One \== critical_section
    ->  Instance = cycle(One, Cycle)
    ;   Instance = Cycle
    ),
    pairs_values(Edges, Lists),
    append(Lists, Facts),
    Evidence = evidence(Name, Instance, Facts).


% ---- Chapter 21: memory, handles, and leaks ----
```

```
% committed_ratio(Heap, Ratio): Ratio is the
% committed part of Heap over its reservation.
committed_ratio(Heap, Ratio) :-
    user:heap(Heap, Reserved, Committed, _),
    Reserved > 0,
    Ratio is Committed / Reserved.

% insufficient_memory(Variant, Instance): the two
% variants of the pattern treated in Chapter 21.
insufficient_memory(committed_memory, heap(H)) :-
    committed_ratio(H, Ratio),
    threshold(committed_ratio, MinRatio),
    Ratio >= MinRatio,
    user:heap(H, Reserved, _, _),
    threshold(large_heap_kb, MinKb),
    Reserved >= MinKb.
insufficient_memory(committed_memory, process) :-
    aggregate_all(sum(C), user:heap(_, _, C, _),
                  Total),
    threshold(committed_total_kb, MaxKb),
    Total >= MaxKb.
insufficient_memory(handle_leak, process(PID)) :-
    user:process(PID, _),
    aggregate_all(sum(C),
                  user:handle_count(PID, _, C),
                  Total),
    threshold(handle_total_count, Max),
    Total > Max.

% variant_name(Variant, Name): the catalog name of
% a variant of Insufficient Memory.
variant_name(committed_memory,
    'Insufficient Memory (Committed Memory)').
variant_name(handle_leak,
    'Insufficient Memory (Handle Leak)').

% insufficient_memory(Variant, Instance, Ev): the
% explainer form for heap instances: the heap and
% the two thresholds the first clause compared it
% with.
insufficient_memory(committed_memory, I, E) :-
    I = heap(H),
    insufficient_memory(committed_memory, I),
    user:heap(H, R, C, F),
    threshold(committed_ratio, Min),
    threshold(large_heap_kb, MinKb),
    variant_name(committed_memory, Name),
    Facts = [ heap(H, R, C, F),
              threshold(committed_ratio, Min),
              threshold(large_heap_kb, MinKb)
            ],
    E = evidence(Name, I, Facts).
```

```
% handle_leak(Type): the count of handles of Type
% exceeds the analyst's limit for one type.
handle_leak(Type) :-
    user:handle_count(_, Type, Count),
    threshold(handle_type_count, Max),
    Count > Max.

% committed_series(Heap, Series): Series is the
% list of Time-CommittedKb pairs of Heap in time
% order, over the heap_at/5 and dump_time/2
% extension of Chapter 21.
committed_series(Heap, Series) :-
    findall(T-C,
            ( user:heap_at(D, Heap, _, C, _),
              user:dump_time(D, T)
            ),
            Pairs),
    keysort(Pairs, Series).

% strictly_increasing(Series): every value is
% greater than the one before it.
strictly_increasing([_-C0 | Pairs]) :-
    increasing_from(Pairs, C0).

% increasing_from(Pairs, Prev): the values of
% Pairs increase, starting above Prev.
increasing_from([], _).
increasing_from([_-C | Pairs], Prev) :-
    Prev < C,
    increasing_from(Pairs, C).

% memory_leak(Heap): the committed size of Heap
% grows at every snapshot, by at least the leak
% threshold over the series.
memory_leak(Heap) :-
    user:heap(Heap, _, _, _),
    committed_series(Heap, Series),
    Series = [_-First, _ | _],
    strictly_increasing(Series),
    last(Series, _-Last),
    threshold(leak_growth_kb, MinGrowth),
    Last - First >= MinGrowth.


% ---- Chapter 22: modules and environment ----

% vendor(Module, Vendor): the company recorded for
% Module; system for a system module without one;
% unrecorded otherwise.
vendor(Module, Vendor) :-
    user:module(Module, _, _),
    (   user:module_info(Module, company, Company)
    ->  Vendor = Company
    ;   system_module(Module)
    ->  Vendor = system
    ;   Vendor = unrecorded
    ).
```

```
% version_parts(Version, Parts): the numbers of a
% dotted version atom, as a list.
version_parts(Version, Parts) :-
    atomic_list_concat(Texts, '.', Version),
    maplist(atom_number, Texts, Parts).

% main_module(Module): the module of the process's
% executable, by its file name.
main_module(Module) :-
    user:process(_, Executable),
    file_name_extension(Base, _, Executable),
    downcase_atom(Base, Module),
    user:module(Module, _, _).

% version_of(Module, Parts): the version of Module
% as a list of numbers; a module has one version.
version_of(Module, Parts) :-
    once(user:module_info(Module, version, Version)),
    version_parts(Version, Parts).

% not_my_version(Module): Module is from the same
% vendor as the executable but has a different
% version.
not_my_version(Module) :-
    main_module(Main),
    vendor(Main, Vendor),
    vendor(Module, Vendor),
    Module \== Main,
    version_of(Main, MainParts),
    version_of(Module, Parts),
    MainParts \== Parts.

% unknown_component(Module): the dump records no
% vendor for Module, or records it as unknown.
unknown_component(Module) :-
    vendor(Module, Vendor),
    memberchk(Vendor, ['Unknown', unrecorded]).

% no_component_symbols(Module): the debugger had
% no symbol file for Module.
no_component_symbols(Module) :-
    user:module_info(Module, symbols, Kind),
    Kind \== pdb.

% hooked_functions(Module, Hooker, Functions): the
% functions of Module that Hooker hooks.
hooked_functions(Module, Hooker, Functions) :-
    bagof(Function,
          user:hooked(Function, Module, Hooker),
          Functions).

% last_error_collection(Collection): the TID-Code
% pairs of every thread with a recorded last
% error, by thread.
last_error_collection(Collection) :-
    findall(TID-Code, user:last_error(TID, Code),
            Pairs),
    keysort(Pairs, Collection).
```

```
% ---- Chapters 17 and 23: the evidence contract ----

% explainer(Goal, Finding): Goal is an explainer of
% this module, with its instance unbound, and
% Finding is the evidence or suspicion term that a
% call of Goal binds.
explainer(spiking_thread(_, Finding), Finding).
explainer(blocked_thread(_, Finding), Finding).
explainer(null_pointer_data(_, Finding), Finding).
explainer(stack_overflow(_, Finding), Finding).
explainer(deadlock(any, _, Finding), Finding).
explainer(insufficient_memory(committed_memory, _,
                              Finding), Finding).

% finding_facts(Finding, Facts): Facts are the
% leaves of an evidence or suspicion term.
finding_facts(evidence(_, _, Facts), Facts).
finding_facts(suspected(_, _, Facts), Facts).

% dump_relation(Name/Arity): the relations of the
% dump schema of Section 15.2 and the extension
% relations of Chapters 20 and 21. A goal of one of
% them follows from the evidence list alone,
% however it is qualified.
dump_relation(process/2).
dump_relation(module/3).
dump_relation(module_info/3).
dump_relation(thread/2).
dump_relation(thread_time/3).
dump_relation(thread_state/2).
dump_relation(frame/5).
dump_relation(exception_record/3).
dump_relation(exception_info/3).
dump_relation(critical_section/4).
dump_relation(waits_for/3).
dump_relation(handle_count/3).
dump_relation(heap/4).
dump_relation(last_error/2).
dump_relation(hooked/3).
dump_relation(mutant/3).
dump_relation(waits_for_object/3).
dump_relation(heap_at/5).
dump_relation(dump_time/2).

% derivable(+Facts, +Goal): Goal follows from the
% list Facts alone. The control constructs and the
% all-solutions and apply predicates the recognizers
% use are interpreted, so that every goal they call
% is derived in turn; a relation of the dump and a
% threshold follow only as members of Facts, whatever
% module qualifies them, and so does any other fact
% of user, the module of the knowledge base; a goal
% of a predicate defined by clauses, in this module
% or in a chapter, follows as a member of Facts or
% through a clause whose body follows; a first-order
% built-in from the list of evaluable/1 is called;
% must_be/2 is decided by checked/2 for a fixed list
% of descriptors, since the library predicate
```

```
% dispatches through an extensible hook; and any
% other goal is outside the vocabulary and raises a
% domain error, rather than being called. A goal or
% list with an attributed variable (Chapter 14) is
% refused, since unifying one runs a hook.
derivable(Facts, Goal) :-
    term_attvars(Facts-Goal, Vars),
    (   Vars == []
    ->  derive(Goal, patterns_dump, Facts)
    ;   Vars = [Var | _],
        copy_term(Var, Plain, _),
        permission_error(derive, attributed_variable,
                         Plain)
    ).

derive(G, _, _) :-
    var(G),
    !,
    instantiation_error(G).
derive(M:G, _, Facts) :-
    !,
    derive(G, M, Facts).
derive(true, _, _) :- !.
derive((A, B), M, Facts) :-
    !,
    derive(A, M, Facts),
    derive(B, M, Facts).
derive((If -> Then ; Else), M, Facts) :-
    !,
    (   derive(If, M, Facts)
    ->  derive(Then, M, Facts)
    ;   derive(Else, M, Facts)
    ).
derive((If *-> Then ; Else), M, Facts) :-
    !,
    (   derive(If, M, Facts)
    *-> derive(Then, M, Facts)
    ;   derive(Else, M, Facts)
    ).
derive((A ; B), M, Facts) :-
    !,
    (   derive(A, M, Facts)
    ;   derive(B, M, Facts)
    ).
derive((If -> Then), M, Facts) :-
    !,
    (   derive(If, M, Facts)
    ->  derive(Then, M, Facts)
    ).
derive(\+ A, M, Facts) :-
    !,
    \+ derive(A, M, Facts).
derive(not(A), M, Facts) :-
    !,
    \+ derive(A, M, Facts).
derive(once(A), M, Facts) :-
    !,
    once(derive(A, M, Facts)).
derive(ignore(A), M, Facts) :-
```

```
    !,
    ignore(derive(A, M, Facts)).
derive(forall(C, A), M, Facts) :-
    !,
    forall(derive(C, M, Facts),
           derive(A, M, Facts)).
derive(catch(G, E, R), M, Facts) :-
    !,
    catch(derive(G, M, Facts), E,
          derive(R, M, Facts)).
derive(findall(T, G, L), M, Facts) :-
    !,
    findall(T, derive(G, M, Facts), L).
derive(findall(T, G, L, Tail), M, Facts) :-
    !,
    findall(T, derive(G, M, Facts), L, Tail).
derive(bagof(T, G, L), M, Facts) :-
    !,
    existential(G, Vs, G1),
    bagof(T, Vs^derive(G1, M, Facts), L).
derive(setof(T, G, L), M, Facts) :-
    !,
    existential(G, Vs, G1),
    setof(T, Vs^derive(G1, M, Facts), L).
derive(aggregate_all(S, G, R), M, Facts) :-
    !,
    aggregate_all(S, derive(G, M, Facts), R).
derive(aggregate_all(S, D, G, R), M, Facts) :-
    !,
    aggregate_all(S, D, derive(G, M, Facts), R).
derive(G, M, Facts) :-
    applied(G, Closure, Args),
    !,
    add_args(Closure, Args, G1),
    derive(G1, M, Facts).
derive(maplist(P, L), M, Facts) :-
    !,
    maplist(derive_with(M, Facts, P), L).
derive(maplist(P, L1, L2), M, Facts) :-
    !,
    maplist(derive_with(M, Facts, P), L1, L2).
derive(maplist(P, L1, L2, L3), M, Facts) :-
    !,
    maplist(derive_with(M, Facts, P), L1, L2, L3).
derive(include(P, L, R), M, Facts) :-
    !,
    include(derive_with(M, Facts, P), L, R).
derive(exclude(P, L, R), M, Facts) :-
    !,
    exclude(derive_with(M, Facts, P), L, R).
derive(foldl(P, L, V0, V), M, Facts) :-
    !,
    foldl(derive_with(M, Facts, P), L, V0, V).
derive(G, _, Facts) :-
    functor(G, Name, Arity),
    (   dump_relation(Name/Arity)
    ;   Name/Arity == threshold/2
    ),
    !,
```

```
    fact_of(G, Facts).
derive(must_be(Type, X), _, _) :-
    !,
    checked(Type, X).
derive(G, _, _) :-
    functor(G, Name, Arity),
    evaluable(Name/Arity, Module),
    !,
    call(Module:G).
derive(G, M, Facts) :-
    program_predicate(M:G, Defined),
    !,
    (   fact_of(G, Facts)
    ;   clause(Defined:G, Body),
        \+ ( Defined == user, Body == true ),
        derive(Body, Defined, Facts)
    ).
derive(G, _, _) :-
    functor(G, Name, Arity),
    domain_error(derivable_goal, Name/Arity).

% fact_of(+Goal, +Facts): Goal is a member of Facts,
% the members' module qualifications ignored.
fact_of(G, Facts) :-
    member(F0, Facts),
    strip_module(F0, _, G).

% existential(+Goal, -Vars, -Plain): Goal without
% its V^ prefixes, which list the variables Vars.
existential(V^G, [V | Vs], G1) :-
    !,
    existential(G, Vs, G1).
existential(G, [], G).

% applied(+Goal, -Closure, -Args): Goal is call/N.
applied(G, Closure, Args) :-
    G =.. [call, Closure | Args].

% add_args(+Closure, +Args, -Goal): Goal is Closure
% with Args appended to its arguments.
add_args(M:C, Args, M:G) :-
    !,
    add_args(C, Args, G).
add_args(C, Args, G) :-
    C =.. L0,
    append(L0, Args, L1),
    G =.. L1.
```

```
% derive_with(+M, +Facts, +Closure, ...): the closure
% applied to the extra arguments is derived.
derive_with(M, Facts, P, A) :-
    add_args(P, [A], G),
    derive(G, M, Facts).
derive_with(M, Facts, P, A, B) :-
    add_args(P, [A, B], G),
    derive(G, M, Facts).
derive_with(M, Facts, P, A, B, C) :-
    add_args(P, [A, B, C], G),
    derive(G, M, Facts).

% checked(+Type, @Term): must_be(Type, Term) decided
% for the descriptors of derivable_type/1 with the
% tests and the error terms of library(error) of
% SWI-Prolog 9.0.4, reproduced here, so that the
% hook has_type/2 of that library, which a loaded
% file may extend with clauses of its own, is never
% consulted. The whole descriptor is validated
% first, element types included; one outside the
% vocabulary raises a domain error, whatever the
% term, which is the one way this differs from the
% library predicate.
checked(Type, X) :-
    descriptor(Type),
    (   of_type(Type, X)
    ->  true
    ;   not_of_type(Type, X)
    ).

% descriptor(+Type): Type is in the vocabulary of
% derivable_type/1, its element type included.
descriptor(Type) :-
    var(Type),
    !,
    instantiation_error(Type).
descriptor(list(Type)) :-
    !,
    descriptor(Type).
descriptor(Type) :-
    derivable_type(Type),
    !.
descriptor(Type) :-
    domain_error(derivable_type, Type).

% derivable_type(@Type): a descriptor checked/2
% decides.
derivable_type(oneof(_)).
derivable_type(atom).
derivable_type(atomic).
derivable_type(integer).
derivable_type(nonneg).
derivable_type(positive_integer).
derivable_type(number).
derivable_type(boolean).
derivable_type(callable).
derivable_type(list).
```

```
derivable_type(list(_)).
derivable_type(between(_, _)).

% of_type(+Type, @X): has_type/2 of library(error)
% for the vocabulary, clause for clause.
of_type(atom, X) :- atom(X).
of_type(atomic, X) :- atomic(X).
of_type(between(L, U), X) :-
    (   integer(L)
    ->  integer(X), between(L, U, X)
    ;   number(X), X >= L, X =< U
    ).
of_type(boolean, X) :- ( X == true ; X == false ), !.
of_type(callable, X) :- callable(X).
of_type(integer, X) :- integer(X).
of_type(nonneg, X) :- integer(X), X >= 0.
of_type(positive_integer, X) :- integer(X), X > 0.
of_type(number, X) :- number(X).
of_type(oneof(L), X) :- ground(X), \+ \+ memberchk(X, L).
of_type(list, X) :- is_list(X).
of_type(list(Type), X) :-
    is_list(X),
    elements_of_type(X, Type).

elements_of_type([], _).
elements_of_type([H | T], Type) :-
    of_type(Type, H),
    !,
    elements_of_type(T, Type).

% not_of_type(+Type, @X): is_not/2 of library(error)
% for the vocabulary: X is known not to be of Type,
% and the error says how.
not_of_type(list, X) :-
    !,
    not_a_list(list, X).
not_of_type(list(Of), X) :-
    !,
    not_a_list(list(Of), X).
not_of_type(Type, X) :-
    (   var(X)
    ->  instantiation_error(X)
    ;   Type = oneof(_),
        \+ ground(X)
    ->  instantiation_error(X)
    ;   type_error(Type, X)
    ).

not_a_list(Type, X) :-
    '$skip_list'(_, X, Rest),
    (   var(Rest)
    ->  instantiation_error(X)
    ;   Rest == []
    ->  Type = list(Of),
        element_not_of_type(X, Of)
    ;   type_error(Type, X)
    ).
```

```
element_not_of_type([H | T], Of) :-
    of_type(Of, H),
    !,
    element_not_of_type(T, Of).
element_not_of_type([H | _], Of) :-
    not_of_type(Of, H).

% program_predicate(+M:Goal, -Defined): the
% predicate of Goal, resolved in module M, has
% clauses, and Defined is the module that defines
% them.
program_predicate(M:G, Defined) :-
    predicate_property(M:G, defined),
    \+ predicate_property(M:G, built_in),
    \+ predicate_property(M:G, foreign),
    (   predicate_property(M:G, imported_from(D))
    ->  Defined = D
    ;   Defined = M
    ),
    \+ ( module_property(Defined, class(Class)),
         Class \== user
       ).

% evaluable(?Name/Arity): the first-order built-ins
% and library predicates that a derivation calls
% rather than derives: unification and comparison,
% arithmetic, type tests, term construction, atoms
% and strings, and lists. None reads the database,
% calls a goal of its own, or dispatches through a
% hook that a loaded file could extend.
evaluable(PI) :-
    evaluable(PI, _).

% evaluable(?Name/Arity, ?Module): as evaluable/1,
% with the module that defines the predicate; the
% derivation calls it there, so that a definition
% of the same name in user, which a meta-call from
% this module would otherwise reach first, is not
% used in its place.
evaluable((=)/2, system).
evaluable((\=)/2, system).
evaluable((==)/2, system).
evaluable((\==)/2, system).
evaluable((@<)/2, system).
evaluable((@>)/2, system).
evaluable((@=<)/2, system).
evaluable((@>=)/2, system).
evaluable(compare/3, system).
evaluable((is)/2, system).
evaluable((<)/2, system).
evaluable((>)/2, system).
evaluable((=<)/2, system).
evaluable((>=)/2, system).
evaluable((=:=)/2, system).
evaluable((=\=)/2, system).
evaluable(succ/2, system).
evaluable(plus/3, system).
evaluable(between/3, system).
evaluable(var/1, system).
```

```
evaluable(nonvar/1, system).
evaluable(atom/1, system).
evaluable(number/1, system).
evaluable(integer/1, system).
evaluable(float/1, system).
evaluable(atomic/1, system).
evaluable(compound/1, system).
evaluable(callable/1, system).
evaluable(is_list/1, system).
evaluable(ground/1, system).
evaluable(functor/3, system).
evaluable(arg/3, system).
evaluable((=..)/2, system).
evaluable(copy_term/2, system).
evaluable(atom_number/2, system).
evaluable(atom_length/2, system).
evaluable(atom_concat/3, system).
evaluable(atomic_list_concat/2, system).
evaluable(atomic_list_concat/3, system).
evaluable(sub_atom/5, system).
evaluable(downcase_atom/2, system).
evaluable(upcase_atom/2, system).
evaluable(file_name_extension/3, system).
evaluable(atom_codes/2, system).
evaluable(atom_chars/2, system).
evaluable(atom_string/2, system).
evaluable(number_codes/2, system).
evaluable(string_concat/3, system).
evaluable(split_string/4, system).
evaluable(length/2, system).
evaluable(member/2, lists).
evaluable(memberchk/2, system).
evaluable(append/2, lists).
evaluable(append/3, lists).
evaluable(reverse/2, lists).
evaluable(nth0/3, lists).
evaluable(nth1/3, lists).
evaluable(last/2, lists).
evaluable(msort/2, system).
evaluable(sort/2, system).
evaluable(sort/4, system).
evaluable(keysort/2, system).
evaluable(pairs_keys_values/3, pairs).
evaluable(pairs_keys/2, pairs).
evaluable(pairs_values/2, pairs).
evaluable(list_to_set/2, lists).
evaluable(sum_list/2, lists).
evaluable(max_list/2, lists).
evaluable(min_list/2, lists).
evaluable(max_member/2, lists).
evaluable(min_member/2, lists).
evaluable(numlist/3, lists).
evaluable(subtract/3, lists).
evaluable(intersection/3, lists).
evaluable(union/3, lists).
evaluable(delete/3, lists).
evaluable(select/3, lists).
evaluable(selectchk/3, lists).
```

```
% sufficient(+Finding): the evidence or suspicion
% term Finding re-derives from its own leaves: the
% explainer that produces findings of its shape,
% called with Finding bound, follows from the
% leaves alone by derivable/2.
sufficient(Finding) :-
    finding_facts(Finding, Facts),
    once(( explainer(Goal, Finding),
           derivable(Facts, Goal)
        )).

% check_finding(+Ctx, +Finding, -Verdict): Verdict
% is ok when Finding keeps the contract of Section
% 17.3 with its leaves called in module Ctx, and
% otherwise names the part it breaks: not_ground;
% fails(Leaf) or raises(Leaf, Error) for a leaf
% that does not succeed in Ctx; or insufficient.
check_finding(Ctx, Finding, Verdict) :-
    finding_facts(Finding, Facts),
    (   \+ ground(Facts)
    ->  Verdict = not_ground
    ;   member(Leaf, Facts),
        leaf_verdict(Ctx, Leaf, Verdict0),
        Verdict0 \== ok
    ->  Verdict = Verdict0
    ;   sufficient(Finding)
    ->  Verdict = ok
    ;   Verdict = insufficient
    ).

% leaf_verdict(+Ctx, +Leaf, -Verdict): Leaf called
% in Ctx succeeds, fails, or raises Error.
leaf_verdict(Ctx, Leaf, Verdict) :-
    catch(( Ctx:Leaf
          ->  Verdict = ok
          ;   Verdict = fails(Leaf)
          ),
          error(Error, _),
          Verdict = raises(Leaf, Error)).

:- module_transparent check_explainers/0.

% check_explainers: every finding of every
% explainer of explainer/2, on the knowledge base
% loaded, keeps the contract, with its leaves
% called in the module that calls this predicate;
% one line per explainer, and one per finding that
% breaks the contract, in which case the check
% fails.
check_explainers :-
    context_module(Ctx),
    patterns_dump:check_explainers_in(Ctx).

% check_explainers_in(+Ctx): as check_explainers/0,
% with the leaves called in Ctx.
check_explainers_in(Ctx) :-
    findall(Goal-Verdict,
            ( explainer(Goal, Finding),
              call(Goal),
```

```
              check_finding(Ctx, Finding, Verdict)
            ),
            Results),
    forall(explainer(Goal, _),
           report_explainer(Goal, Results)),
    forall(member(_-Verdict, Results),
           Verdict == ok).

% report_explainer(+Goal, +Results): one line for
% the explainer Goal with the number of its
% findings that keep the contract, and one line for
% each finding that does not.
report_explainer(Goal, Results) :-
    functor(Goal, Name, Arity),
    findall(Instance-Verdict,
            ( member(Goal-Verdict, Results),
              arg(Arity, Goal, Finding),
              arg(2, Finding, Instance)
            ),
            Own),
    aggregate_all(count, member(_-ok, Own), Ok),
    length(Own, N),
    format("~a/~d: ~d of ~d ok~n",
           [Name, Arity, Ok, N]),
    forall(( member(Instance-Verdict, Own),
             Verdict \== ok
           ),
           format("    ~q: ~q~n", [Instance, Verdict])).

?- use_module(patterns_dump).
true.

?- deadlock(critical_section, C).
C = [103, 104] ;
false.
```

Appendix C. Pattern Cross-Reference

The table lists every diagnostic analysis pattern treated in this book, the predicate or predicates that recognize it, and, in parentheses, the chapter in which each is defined. Patterns treated in prose only are marked as such. A pattern that several chapters treat is listed with each chapter's predicate, because each chapter is a program of its own; the Part III and Part IV definitions are the ones the book regards as canonical.

C.1 Memory Analysis Patterns

Pattern	Predicate (chapter)
Active Thread (page 337)	`active_thread/1` (1); `thread_class/2` (7); `active_thread/1` (18)
Blocked Queue (page 396)	prose (9); prose (18); `blocked_queue/3` (20)
Blocked Thread (Software) (page 336)	`blocked_thread/1` (3); `thread_class/2` (7); `blocked_thread/2` (17); `blocked_thread/1` (18); prose (20); `blocked_thread/1` (23); `blocked_thread/1` (29); `blocked_thread/1` (31)
Blocking Module (page 403)	`blocking_module/3` (20)
Busy System	`busy_system/0` (22)
C++ Exception (page 376)	`cpp_exception/1` (19)
Caller-n-Callee (page 343)	`caller_n_callee/3` (18)
Changed Environment (page 436)	`changed_environment/2` (22)
Coincidental Symbolic Information (page 347)	`coincidental_symbolic_information/3` (18); prose (31)
Corrupt Dump (page 457)	prose (23)
Crash Signature (page 380)	`crash_signature/2` (19)
Critical Section Corruption (page 398)	`critical_section_corruption/2` (20)
Data Correlation (Function	`data_correlation/3` (21)

Pattern	Predicate (chapter)
Parameters) (page 424)	
Deadlock (Critical Sections) (page 388)	prose (4); `deadlocked/1` (10); `deadlock_cycle/1` (11); prose (13); `deadlock_timeline/4` (14); `deadlock/2` (17); `deadlock/2` (20); `deadlock/2` (23); `deadlock_corroborated/3` (28); `deadlock/2` (29); `deadlock/2` (31)
Deadlock (Mixed Objects) (page 394)	`deadlock_mixed/2` (20)
Deviant Module (page 438)	`deviant_module/2` (22)
Dispatch Level Spin (page 400)	prose (20)
Divide by Zero (User Mode) (page 371)	`divide_by_zero/1` (19)
Double Free (Process Heap) (page 416)	`double_free/1` (21)
Dynamic Memory Corruption (Process Heap) (page 414)	`dynamic_memory_corruption/1` (21)
Early Crash Dump (page 454)	`early_crash_dump/1` (23)
Exception Module (page 378)	`exception_module/2` (19)
Exception Stack Trace (page 358)	`exception_stack_trace/2` (19); `exception_stack_trace/2` (23); `exception_stack_trace/3` (27)
Execution Residue (User Space) (page 350)	`execution_residue/4` (18); prose (31)
False Positive Dump (page 455)	`false_positive_dump/2` (23)
Fault Context (page 359)	`fault_context/2` (19)
Frame Regularity (page 342)	prose (3); prose (5); `top_run/3` (7); `frame_regularity/2` (17); `frame_regularity/3` (18); `frame_regularity/3` (19); `frame_regularity/3` (23); `frame_regularity/1` (29)
Frozen Process	`frozen_process/1` (22)
Glued Stack Trace (page 349)	`glued_stack_trace/1` (18); prose (19)
Handle Leak (page 409)	`handle_leak/2` (17); `handle_leak/1` (21); `handle_leak/1` (23); `handle_leak/1` (29); `handle_leak/1` (31)

Pattern	Predicate (chapter)
Handled Exception (User Space) (page 375)	`handled_exception/2` (19)
Hidden Exception (User Space) (page 374)	`hidden_exception/2` (19); prose (31)
Hidden Module (page 435)	`hidden_module/2` (22)
High Contention (Critical Sections) (page 397)	`high_contention/2` (20)
Historical Information (page 443)	prose (22)
Hooked Functions (User Space) (page 433)	prose (15); `hooked_functions/2` (17); `hooked_functions/3` (22); `hooked_functions/3` (23)
Hooked Modules (page 434)	`hooked_modules/1` (22)
Inconsistent Dump (page 453)	`inconsistent_dump/1` (23)
Incorrect Stack Trace (page 345)	`incorrect_stack_trace/2` (18)
Insufficient Memory (Committed Memory) (page 406)	`insufficient_memory/2` (21); `insufficient_memory/2` (23)
Insufficient Memory (Handle Leak) (page 410)	`insufficient_memory/2` (21); `insufficient_memory/2` (23)
Invalid Handle (General) (page 419)	`invalid_handle/1` (21)
Invalid Pointer (General) (page 362)	`invalid_pointer/1` (19)
Last Error Collection (page 443)	`last_error_collection/1` (22); `last_error_collection/1` (23)
Lateral Damage (page 446)	prose (22)

Pattern	Predicate (chapter)
Least Common Frame (page 339)	`least_common_frame/2` (18)
Local Buffer Overflow (User Space) (page 365)	`local_buffer_overflow/2` (19)
Main Thread (page 332)	`main_thread/1` (18); `main_thread/1` (23)
Managed Code Exception (page 376)	`managed_code_exception/1` (19)
Manual Dump (Process) (page 451)	prose (17); `manual_dump/1` (23)
Memory Fluctuation (page 413)	`memory_fluctuation/1` (21)
Memory Leak (Process Heap) (page 411)	`memory_leak/1` (21); `memory_leak/1` (23)
Missing Component (General) (page 432)	`missing_component/1` (22)
Module Hint (page 442)	`module_hint/2` (22)
Module Variety (page 427)	`module_variety/1` (9); `module_variety/1` (22)
Multiple Exceptions (User Space) (page 373)	`multiple_exceptions/1` (19)
Nested Exceptions (User Space) (page 372)	`nested_exceptions/1` (19)
No Component Symbols (page 431)	`no_component_symbols/1` (2); `unknown_frame/3` (8); `symbol//3` (12); prose (15); `no_component_symbols/2` (17); `no_component_symbols/1` (22); `no_component_symbols/1` (23)
No Current Thread (page 458)	`no_current_thread/0` (23)
Not My Version (page 429)	prose (15); `not_my_version/1` (22); `not_my_version/1` (23)
NULL Pointer (Code) (page 361)	`null_pointer_code/1` (19); prose (23)

Pattern	Predicate (chapter)
NULL Pointer (Data) (page 360)	prose (6); `null_pointer_data/2` (17); `null_pointer_data/1` (19); `null_pointer_data/1` (23); prose (28)
Origin Module (page 441)	`origin_module/2` (22)
Paged Out Data (page 420)	`paged_out_data/1` (21)
Paratext (page 509)	prose (24)
Passive Thread (page 334)	`passive_thread/1` (1); prose (3); `thread_class/2` (7); `passive_thread/1` (18); prose (19)
Problem Module (page 379)	`problem_module/2` (19)
Procedure Call Chain (page 344)	`procedure_call_chain/2` (18)
Regular Data (page 423)	`regular_data/2` (21)
Same Vendor (page 439)	`same_vendor/3` (22)
Shared Buffer Overwrite (page 421)	prose (21)
Special Process	`special_process/1` (22)
Special Thread (page 333)	`special_thread/2` (18)
Spiking Thread (page 335)	`spiking_thread/1` (6); prose (7); prose (9); `print_cpu_shares/0` (10); `spiking_thread/1` (13); `spiking_thread/2` (17); `spiking_thread/1` (18); `spiking_thread/1` (20); `spiking_thread/1` (23); `spiking_corroborated/2` (28); `spiking_thread/1` (29); `spiking_thread/1` (31)
Stack Overflow (User Mode) (page 366)	prose (18); `stack_overflow/2` (19); `stack_overflow/2` (23); prose (29)
Stack Trace (page 329)	`stack_trace/2` (5); `print_frame/2` (8); `stack_trace/2` (9); `frame_text//1` (12); `stack_trace/2` (13); `stack_trace/2` (18)
Stack Trace Collection (Managed Space)	prose (15)
Stack Trace Collection (Predicate) (page 331)	`stack_trace_collection/2` (18)
Stack Trace Collection	`stack_trace_collection/1` (9); `stack_trace_collection/1` (18); prose (31)

Pattern	Predicate (chapter)
(Unmanaged Space) (page 330)	
Stack Trace Set (page 341)	`stack_trace_set/2` (18)
Stack Trace Signature (page 338)	`stack_trace_signature/2` (18)
Step Dumps (page 460)	`step_dumps/2` (23)
Stored Exception (page 377)	`past_exception/3` (19)
Suspended Thread	`suspended_thread/1` (1); `thread_class/2` (7); `suspended_thread/1` (23)
Swarm of Shared Locks (page 400)	`swarm_of_shared_locks/2` (20)
Thread Age (page 338)	prose (18)
Thread Starvation (Normal Priority) (page 402)	`thread_starvation/2` (20)
Top Module (page 440)	`top_module/2` (22)
Truncated Dump (page 452)	prose (17); `truncated_dump/1` (23)
Truncated Stack Trace (page 346)	`truncated_stack_trace/1` (18)
Ubiquitous Component (page 447)	`ubiquitous_component/1` (22)
Unknown Component (page 430)	`unknown_component/1` (2); prose (15); `unknown_component/1` (22); `unknown_component/1` (23)
Value References (page 421)	`value_references/2` (21)
Virtualized Process (WOW64) (page 437)	`virtualized_process/0` (22)
Wait Chain (Critical Sections) (page 384)	`blocked_by/2` (4); `blocked_through/2` (5); `waits_on/2` (10); `wait_chain/3` (11); `blocked_on/2` (13); `wait_chain/2` (17); `wait_chain/2` (20); `wait_chain/2` (23); `blocked_by/2` (29)
Wait Chain (Mutex Objects) (page 386)	`wait_chain/2` (20)

Pattern	Predicate (chapter)
Wait Chain (Thread Objects) (page 387)	`wait_chain/2` (20)
Waiting Thread Time (page 401)	`waiting_thread_time/2` (20)
Wild Code (page 364)	`wild_code/1` (19)
Wild Pointer (page 363)	`wild_pointer/1` (19)
Zombie Processes	`zombie_processes/1` (22)

C.2 Trace and Log Analysis Patterns

Pattern	Predicate (chapter)
Activity Overlap (page 542)	`activity_overlap/4` (26)
Activity Region (page 540)	`activity_region/3` (26)
Activity Theatre (page 541)	`activity_theatre/3` (26)
Adjoint Thread of Activity (page 538)	`adjoint_thread_of_activity/2` (26)
Anchor Messages (page 577)	`anchor_messages/1` (27)
Background and Foreground Components (page 549)	`background_components/1` (26); `foreground_components/3` (26)
Basic Facts (page 483)	`basic_facts/1` (24)
Blackout (page 535)	`blackout/1` (25)
Characteristic Message Block (page 574)	`characteristic_message_block/3` (27)
Correlated Discontinuity (page 514)	`correlated_discontinuity/2` (25)
Counter Value (page 568)	`depth_model/4` (14); `counter_value/3` (27)
Coupled Activities (page 545)	`coupled_activities/3` (26)
Data Flow (page 559)	`data_flow/3` (26)
Data Reversal (page 615)	`data_reversal/4` (28)
Defamiliarizing Effect (page 599)	prose (27)
Delay Dynamics (page 522)	prose (13); `delay_dynamics/2` (25)
Density Distribution (page 518)	`density_distribution/3` (25)
Dialogue (page 616)	`dialogue/2` (28)
Diegetic Messages (page 590)	`diegetic_messages/2` (27)
Discontinuity (page 513)	prose (13); `delta/3` (16); `discontinuity/2` (25); `discontinuity/2` (30); prose (31)
Dominant Event Sequence (page 557)	`dominant_event_sequence/2` (26); `dominant_event_sequence/2` (28)
Empty Trace (page 500)	`empty_trace/1` (24)

Pattern	Predicate (chapter)
Error Distribution (page 555)	`error_distribution/2` (26)
Error Message (page 565)	`error_message/1` (12); `error_message/1` (27)
Error Thread (page 554)	`error_thread/2` (26)
Event Sequence Order (page 603)	`request_order/1` (14); prose (16); `event_sequence_order/2` (28)
Event Sequence Phase (page 604)	`message_phase/2` (28)
False Positive Error (page 589)	`false_positive_error/2` (27)
Fiber Bundle (page 543)	`fiber_bundle/2` (26)
Fiber of Activity (page 543)	`fiber_of_activity/3` (26)
Focus of Tracing (page 494)	`focus_of_tracing/1` (24)
Glued Activity (page 547)	`glued_activity/2` (26)
Guest Component (page 552)	`guest_component/2` (26)
Hidden Error (page 584)	`hidden_error/1` (27)
Identification Messages (page 582)	`identification_messages/1` (27)
Implementation Discourse (page 583)	`implementation_discourse/1` (27)
Impossible Trace (page 614)	`feasible/1` (14); `orderable/2` (28)
Indirect Message (page 581)	`indirect_message/2` (27)
Inter-Correlation	prose (16); prose (28)
Intra-Correlation	`intra_correlation/3` (28)
Karnaugh Map (page 620)	`karnaugh_map/0` (28)
Last Activity (page 532)	`last_activity/3` (25); prose (30)
Linked Messages (page 570)	`linked_messages/3` (27)
Macrofunction (page 606)	`macrofunction/2` (28)
Marked Message (page 591)	`marked_message/3` (27)
Master Trace	`print_merged/3` (14); prose (16); `master_trace/1` (28)
Message Annotations (page 596)	`message_annotation/2` (27)
Message Change (page 587)	`message_change/4` (27)
Message Context (page 580)	`message_context/3` (27)
Message Cover (page 597)	`message_cover/2` (27)
Message Directory (page 503)	`message_directory/1` (24)
Message Interleave (page 556)	`message_interleave/4` (26)
Message Invariant (page 567)	`message_invariant/2` (27)
Message Set (page 485)	`message_set/2` (24)
Message Space (page 490)	`message_space/1` (24)
Milestones (page 531)	`milestones/1` (25)
Minimal Trace (page 617)	`minimal_trace/1` (28)
Missing Message (page 575)	prose (14); `missing_message/4` (27)
Motif (page 607)	`motif/3` (28)
No Activity (page 533)	`no_activity/3` (25)
Opposition Messages (page 572)	`opposition_messages/2` (27)
Palimpsest Messages (page 592)	`palimpsest_messages/4` (27)

Pattern	Predicate (chapter)
Periodic Error (page 523)	`periodic_error/3` (25)
Periodic Message Block (page 525)	`periodic_message_block/4` (25)
Pivot Message (page 579)	`pivot_message/3` (27)
Punctuated Activity (page 529)	`punctuated_activity/2` (25)
Quotient Trace (page 613)	`quotient_trace/1` (28)
Recovered Messages (page 595)	`recovered_messages/2` (27)
Relative Density (page 520)	`relative_density/5` (25)
Resume Activity (page 548)	`resume_activity/3` (26)
Semantic Mapping (page 618)	`semantic/2` (28)
Shared Point (page 571)	prose (16); `shared_point/2` (27)
Sheaf of Activities (page 544)	`sheaf_of_activities/3` (26)
Significant Event (page 566)	`significant_event/2` (27)
Significant Interval (page 528)	`significant_interval/4` (25)
Silent Messages (page 584)	`silent_messages/1` (27)
Singleton Event (page 501)	`singleton_event/1` (24)
Small DA+TA (page 622)	prose (16); `exception_corroborated/3` (28)
Sparse Trace (page 499)	`sparse_trace/1` (24)
Split Trace	`skew/1` (14); `master_trace/1` (28)
State Dump (page 621)	`state_dump/3` (28)
Statement Density and Current (page 515)	`statement_current/3` (25); `statement_density/4` (25)
Thread of Activity (page 537)	`thread_messages/2` (16); `thread_of_activity/2` (26); prose (30)
Time Delta (page 511)	`report_deltas/2` (13); prose (16); `time_delta/3` (25); `gap/3` (31)
Timeout (page 527)	`timeout/2` (25)
Trace Acceleration (page 521)	`trace_acceleration/3` (25)
Trace Dimension (page 484)	`trace_dimension/2` (24)
Trace Extension (page 496)	`trace_extension/1` (24)
Trace Frames (page 493)	`trace_frames/2` (24)
Trace Mask (page 496)	`trace_mask/2` (24)
Trace Partition (page 489)	`component_messages/2` (16); `trace_partition/2` (24)
Trace Summary (page 505)	`trace_summary/0` (24)
Trace Viewpoints (page 506)	`trace_viewpoint/3` (24)
Translated Message (page 593)	`translated_message/3` (27)
Truncated Trace (page 498)	`truncated_trace/2` (24)
Use Case Trail (page 561)	`use_case_trail/2` (26)
Visibility Limit (page 500)	prose (24)
Visitor Trace (page 618)	`visitor_trace/2` (28)
Vocabulary Index (page 487)	`vocabulary/1` (9); `vocabulary_index/1` (24)
Watch Thread (page 553)	`watch_thread/2` (26)
Working Set (page 508)	`working_set/2` (24)

Appendix D. Glossary

The terms are defined as this book uses them. Prolog terms follow the SWI-Prolog manual and the ISO standard; diagnostics terms follow the author's pattern catalogs.

Activity. A set of messages of a trace read as one unit, such as the messages of a thread (a thread of activity) or of a request across threads (an adjoint thread of activity).

Arity. The number of arguments of a compound term or predicate; written after a slash in a predicate indicator, as in `frame/5`.

Artifact. A record of a software system's state or behavior that is analyzed after the fact: a memory dump, a software trace, a log.

Atom. A symbolic constant, such as `ntdll` or `'Worker::Run'`; quoted when it does not begin with a lowercase letter or contains characters other than letters, digits, and underscores.

Backtracking. Returning to the most recent choice point to try the next alternative after a goal fails or after the toplevel asks for another answer.

Body. The goals of a rule, to the right of `:-`.

Choice point. A record of alternatives not yet tried: further clauses of a predicate, or further solutions of a nondeterministic built-in.

Clause. A fact or a rule; the unit of a Prolog program.

Closed-world assumption. The reading of negation as failure under which a goal with no solution is taken to be false; in a knowledge base built from an artifact, it is sound only for what the artifact and its rendering into facts were made to record.

Compound term. A term with a name and one or more arguments, such as `frame(100, 0, ntdll, 'NtWaitForMultipleObjects', 20)`.

Constraint. A relation between variables that is posted and enforced by propagation rather than tested by a goal; in this book, `dif/2` and the constraints of CLP(FD).

Cut. The control construct `!` that commits to the clause and the bindings made so far and discards the choice points created since the clause was entered.

Definite clause grammar (DCG). A grammar notation translated into Prolog clauses with two extra arguments carrying the list to be parsed and its remainder.

Determinism. A goal is deterministic when it succeeds once without leaving a choice point; the toplevel prints its answer with a final period rather than a semicolon.

Diagnostic analysis pattern. A named, recurring structure of abnormal software structure or behavior found in an artifact, described in the pattern catalogs; a memory analysis pattern or a trace and log analysis pattern.

Dynamic predicate. A predicate whose clauses can be added and removed at run time by `assertz/1` and `retract/1`.

Evidence term. The term `evidence(Pattern, Instance, Facts)` with which an explainer form of a recognizer reports a pattern instance together with the ground, callable facts and thresholds that establish it and from which the instance can be re-derived.

Explainer. The form `p(?Instance, -Evidence)` of a pattern recognizer, from which the Boolean test and the enumerator forms are derived.

Fact. A clause without a body; a term asserted to be true.

Frame. One entry of a stack trace: a return address resolved, where symbols allow, to a module, a function, and an offset.

Functor. The name and arity of a compound term; by extension, the name alone.

Goal. A term called for execution; a query or a subgoal of a rule body.

Ground term. A term containing no variables.

Head. The term to the left of `:-` in a rule, or the whole of a fact.

Horn clause. A clause with at most one positive literal; a Prolog clause is a definite Horn clause with exactly one.

Indexing. The selection by the system of candidate clauses by the value of one or more arguments without trying each clause; first-argument indexing is always available, and SWI-Prolog adds just-in-time indexes on other arguments as queries require.

Instance (of a pattern). A concrete occurrence of a pattern in an artifact: a thread, a cycle of threads, a message, an interval.

Knowledge base. The set of loaded clauses, used in this book for the facts that render an artifact together with the rules that query them.

Logical update view. The rule that a goal running over a dynamic predicate sees the clauses as they were when the goal started, regardless of assertions and retractions made while it runs.

Memory dump. A file holding the contents of a process's or a system's memory at an instant, from which a debugger reconstructs threads, stack traces, modules, and other structures.

Message. The unit of a software trace or log: a line or record with a position, usually a timestamp, an origin, and text.

Meta-interpreter. A Prolog program that executes Prolog programs, written over `clause/2`; the vehicle for proof trees and explanations.

Mode. The notation of which arguments of a predicate are inputs and which are outputs, written `+`, `-`, `?`, `@`, and `:` as in the SWI-Prolog manual.

Module. In Prolog, a namespace of predicates with declared exports; in a memory dump, a loaded executable or library with a base address and size.

Most general unifier. The substitution that unifies two terms and specializes them no more than necessary; the result of unification.

Negation as failure. The construct `\+ Goal`, which succeeds if and only if `Goal` has no solution under the current bindings.

Occurs check. The test, omitted by default, that a variable is not bound to a term containing it; without it, unification can build a cyclic term.

Operational definition. The definition a recognizer implements, which is narrower than, and must be stated alongside, the catalog's description of the pattern.

Pattern recognizer. A predicate whose solutions are the instances of a pattern in a knowledge base.

Predicate. A relation named by a functor, defined by its clauses; a procedure in the procedural reading.

Procedure. The set of clauses of a predicate, read as a definition of how to solve goals of that predicate.

Proof tree. The tree of goals and the clauses that resolved them in a successful derivation; built explicitly by the meta-interpreters of Chapter 29.

Provenance. The record of which command, file, or analyst produced a fact; in this book, the `source/3` relation.

Query. A goal submitted to the toplevel, written after `?-`.

Resolution. The inference rule that reduces a goal by unifying it with a clause head and replacing it by the clause body; SLD resolution is the strategy Prolog uses.

Rule. A clause with a head and a body.

Schema. The fixed set of predicates, with their argument types and meanings, by which this book renders an artifact into facts.

Search tree. The tree of alternatives explored by resolution; Prolog traverses it depth-first, left to right.

Software trace. A sequence of messages recorded by a running system, together with their order and timing; a log is a software trace, and the book uses the terms as the catalog does.

Stack trace. The sequence of frames of a thread, from the most recent call to the thread's start, as reconstructed by a debugger from the thread's stack.

String. In SWI-Prolog, a distinct type of text object, written in double quotes by default; not a list of codes.

Substitution. A mapping from variables to terms; applying it binds the variables.

Suspicion term. The term `suspected(Pattern, Instance, Grounds)` with which a recognizer reports that an instance is suspected but not established; its grounds obey the same contract as evidence, and it is never mixed into an evidence list.

Tabling. The execution strategy, enabled by `:- table`, that records the answers of a goal and reuses them, so that left-recursive and cyclic definitions terminate.

Term. The single data type of Prolog: an atom, a number, a string, a variable, or a compound term.

Threshold. A numeric limit on which a recognizer depends, stated as a `threshold/2` fact so that it is visible and changeable.

Toplevel. The interactive loop that reads a query, runs it, and prints its answers.

Unification. The operation that makes two terms identical by binding variables, or fails if no binding can; the single mechanism by which Prolog passes arguments, matches clauses, and constructs data.

Variable. A placeholder, written with an initial capital letter or underscore, that stands for a term not yet known; the anonymous variable _ is distinct at every occurrence.

Wait chain. A sequence of threads each waiting for a synchronization object owned by the next; a cycle in it is a deadlock.

Bibliography

Allievi, A., Russinovich, M., Ionescu, A., and Solomon, D. A. *Windows Internals, Part 2*. Seventh Edition. Microsoft Press, 2021.

Apt, K. R. *From Logic Programming to Prolog*. Prentice Hall, 1997.

Bratko, I. *Prolog Programming for Artificial Intelligence*. Fourth Edition. Addison-Wesley, 2012.

Chen, W. and Warren, D. S. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM* 43(1), 1996, 20–74.

Clark, K. L. Negation as Failure. In Gallaire, H. and Minker, J. (eds.), *Logic and Data Bases*. Plenum Press, 1978, 293–322.

Clocksin, W. F. and Mellish, C. S. *Programming in Prolog: Using the ISO Standard*. Fifth Edition. Springer, 2003.

Colmerauer, A. and Roussel, P. The Birth of Prolog. In Bergin, T. J. and Gibson, R. G. (eds.), *History of Programming Languages II*. ACM Press, 1996, 331–367.

Covington, M. A., Bagnara, R., O'Keefe, R. A., Wielemaker, J., and Price, S. Coding Guidelines for Prolog. *Theory and Practice of Logic Programming* 12(6), 2012, 889–927.

ISO/IEC 13211-1:1995. *Information technology — Programming languages — Prolog — Part 1: General core*. International Organization for Standardization, 1995.

Jaffar, J. and Lassez, J.-L. Constraint Logic Programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*. ACM, 1987, 111–119.

Kowalski, R. A. Predicate Logic as Programming Language. In *Information Processing 74: Proceedings of the IFIP Congress*. North-Holland, 1974, 569–574.

Kowalski, R. A. *Logic for Problem Solving*. North-Holland, 1979.

Lloyd, J. W. *Foundations of Logic Programming*. Second Edition. Springer, 1987.

Microsoft Corporation. *Debugging Tools for Windows*. Documentation, learn.microsoft.com/windows-hardware/drivers/debugger; the `k` command reference and the MINIDUMP_TYPE reference were consulted in October 2026.

O'Keefe, R. A. *The Craft of Prolog*. MIT Press, 1990.

Pereira, F. C. N. and Warren, D. H. D. Definite Clause Grammars for Language Analysis: A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence* 13(3), 1980, 231–278.

Robinson, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* 12(1), 1965, 23–41.

Shapiro, E. Y. *Algorithmic Program Debugging*. MIT Press, 1983.

Sterling, L. and Shapiro, E. *The Art of Prolog: Advanced Programming Techniques*. Second Edition. MIT Press, 1994.

Triska, M. The Finite Domain Constraint Solver of SWI-Prolog. In *Functional and Logic Programming (FLOPS 2012)*, Lecture Notes in Computer Science 7294. Springer, 2012, 307–316.

Triska, M. *The Power of Prolog*. Online book, www.metalevel.at/prolog; consulted in October 2026.

Vostokov, D. *Memory Dump Analysis Anthology*. Volumes 1–15. OpenTask, 2008–2023.

Vostokov, D. *Encyclopedia of Crash Dump Analysis Patterns: Detecting Abnormal Software Structure and Behavior in Computer Memory*. Third Edition. OpenTask, 2020.

Vostokov, D. *Trace, Log, Text, Narrative, Data: An Analysis Pattern Reference for Information Mining, Diagnostics, Anomaly Detection*. Fifth Edition. OpenTask, 2023.

Vostokov, D. *Theoretical Software Diagnostics: Collected Articles*. Fourth Edition. OpenTask, 2024.

Vostokov, D. *Applied Set Theory*. OpenTask, 2026.

Vostokov, D. *Category Theory for Software Diagnostics, Observability and Debugging*. OpenTask, 2026.

Vostokov, D. *Software Diagnostics Library*. DumpAnalysis.org.

Vostokov, D. *PrologForSoftwareDiagnostics: The Code of This Book*. github.com/DumpAnalysis/PrologForSoftwareDiagnostics, 2026.

Vostokov, D. *PrologWinDbg: A WinDbg Extension with a Persistent SWI-Prolog Worker*. Version 0.2.0, github.com/DumpAnalysis/PrologWinDbg, 2026.

Warren, D. S. Memoing for Logic Programs. *Communications of the ACM* 35(3), 1992, 93–111.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. SWI-Prolog. *Theory and Practice of Logic Programming* 12(1–2), 2012, 67–96.

Wielemaker, J. *SWI-Prolog Reference Manual*. The edition for version 9.0.4, as installed with that release and published at www.swi-prolog.org/pldoc; consulted in October 2026. The manual section for each built-in predicate is reached from its name at www.swi-prolog.org/pldoc/doc_for?object=Name/Arity.

Yosifovich, P., Russinovich, M. E., Ionescu, A., and Solomon, D. A. *Windows Internals, Part 1*. Seventh Edition. Microsoft Press, 2017.

Index

!

!/0 (in DCG bodies), 202

!analyze -v, 358

!cs, 383

!handle, 386

!locks, 65, 384

#

#\\=/2, 241

#<==>/2, 243

#=/2, 240

#==>/2, 243

*

**/2, 99

*->/2, 117

,

,/2, 71

.

.ecxr, 359

.exr, 377

.logclose, 264

.logopen, 264

/

// (integer division), 517

///2, 99

//2, 98

//2 (nonterminal indicator), 201

/\\/2, 100

:

:</2 (dict select), 176

;

;/2, 71

?

?=/2, 61

@

@</2, 49

\

\/2, 100

\/1, 100, 272

\+ \\+ (double negation), 118

\+/1, 90, 117

\=/2, 55

\==/2, 55, 239

\c (escape), 207

^

^/2, 99

^/2 (existential quantifier), 149

{

{}/1 (DCG goal), 201

~

~*k, 63

~16r, 104

~d, 103

<

</2, 97

<</2, 100

=

=

=/2, 97

=../2, 140

=/2, 55

=@=/2, 479

=\\=/2, 97

=</2, 97

==/2, 55

>

>/2, 97

->/2, 115

->/2 (in DCG bodies), 202

>=/2, 97

>>/2, 100, 272

>>/N (lambda), 171

A

abolish/1, 186

abolish_all_tables/0, 192, 657

abs/1, 99

absence/2, 322

absolute_file_name/3, 222

access violation, 360

accumulator, 88

Active Thread, 37, 64, 125, 337

Activity Overlap, 542

Activity Region, 540

Activity Theatre, 541

acyclic_term/1, 59

add_nb_set/2, 655

add_nb_set/3, 655

address_module/2, 105, 348

Adjoint Thread of Activity, 538

aggregate/3, 151

aggregate_all/3, 150, 323, 375, 405, 428, 566

- bag, 150
- count, 150, 455
- max, 150
- min, 150
- set, 150
- sum, 150
- witness, 150, 559

agrees/2, 479

all_different/1, 242

all_distinct/1, 242, 603

analysis pattern network, 647

Anchor Messages, 577

anonymous variable, 35, 43

answer subsumption, 192

answer_write_options (flag), 635

append/2, 314, 392

append/3, 84, 345, 561, 574

apropos/1, 33

arg/3, 140

argument, 28, 44

argv_options/3, 224, 663

argv_usage/1, 664

arithmetic comparison, 420

arithmetic expression, 96

arithmetic function, 96

arity, 28, 44

asserta/1, 185

assertion/1, 231

assertz/1, 185, 395, 597

- bulk loading, 651
- copying, 185

assoc_to_keys/2, 174

assoc_to_list/2, 174

assoc_to_values/2, 174

association list, 173, 654

atom, 28, 40, 132

- interning, 41
- quoted, 40

atom//1, 208

atom/1, 40

atom_chars/2, 132

atom_codes/2, 132

atom_concat/3, 134, 269

atom_json_dict/3, 228, 666

atom_length/2, 133

atom_number/2, 133, 429

atom_string/2, 132

atomic/1, 49

atomic_list_concat/2, 134

atomic_list_concat/3, 134, 429

split mode, 134

atomics_to_string/2, 292

attributed variable, 240

autoload (flag), 85

autoloading, 218

B

b_getval/2, 189

b_setval/2, 189

back_quotes (flag), 48

Background and Foreground Components, 549

background components, 549

backtracking, 30, 70

bagof/3, 149, 341, 396, 434, 510, 538

Basic Facts, 483

begin_tests/1, 325

between/3, 101

as a generator, 651

binding, 29, 57

bitwise operations, 364, 423

Blackout, 535

blanks//0, 204

Blocked Queue, 396

Blocked Thread (Software), 64, 125, 312, 336, 459, 626

blocked_by/2, 78

blocked_on/3, 313

blocked_thread/1, 64

Blocking Module, 403

body, 29

boundary message, 548

bounded (flag), 42

bounds consistency, 241

built_in (predicate property), 629

burst, 529

Busy System, 446

C

C++ Exception, 376

call//1, 203

call/1, 164, 536

call/2, 498

call/3, 317

call/N, 164

call_nth/2, 153

call_with_depth_limit/3, 641

call_with_inference_limit/3, 641, 690

call_with_time_limit/2, 690

callable/1, 49

Caller-n-Callee, 343

canonical form, 45

capture, 258

capture policy, 278

capture_policy/2, 278

catch/3, 120

catch_with_backtrace/3, 235

ceiling/1, 100

certainty factor, 645

Changed Environment, 436

char list, 47, 132

char_code/2, 132

character code, 41

Characteristic Message Block, 574

check/0, 233

check_explainers/0, 474

check_finding/3, 475

check_keys/1, 268

choice point, 29, 70, 111, 411, 563

clause, 28

  renaming, 58

clause index, 650

clause indexing, 71

clause order, 73

clause/2, 187, 287, 626

close/1, 211, 225

closed-world assumption, 118, 320, 416

closure, 164

CLP(FD), 240, 614

code list, 47, 132

code_type/2, 488, 568

Coincidental Symbolic Information, 347

collecting goal, 630

compare/3, 49, 414

completeness, 71

completeness assumption, 418

component, 289

component_messages/2, 297

compound term, 44

compound/1, 45

compound_name_arguments/3, 140

compound_name_arity/3, 140

conjunction, 29, 71

consistency check, 398

constraint, 238

consult, 32

consult/1, 32, 352

context

  of a pattern, 303

convlist/3, 570, 622

copy_term/2, 62

coroutining, 238

Correlated Discontinuity, 514

Corrupt Dump, 457

Counter Value, 248, 568

Coupled Activities, 545

coverage/2, 279, 320

covering/2, 276

Crash Signature, 380

critical section, 27, 422

Critical Section Corruption, 398

critical_section/4, 77, 261, 384

csv_read_file/2, 227, 296, 667

csv_read_file/3, 297, 667

csv_read_file_row/3, 667

csv_write_file/2, 227, 296, 667

current_predicate/1, 188, 282, 322

cut, 112

  green, 113

  in interpreted clauses, 632

  opacity, 114

  red, 113

  transparency, 114

cycle, 459

cyclic term, 59

cyclic_term/1, 59

D

Data Correlation (Function Parameters), 424

Data Flow, 559

Data Reversal, 615

DCG

  scanning, 539

Deadlock (Critical Sections), 78, 181, 195, 251, 312, 388, 459, 626

Deadlock (Mixed Objects), 394

deadlock_witness/3, 388

debug mode, 77

debug/1, 230

debug/3, 230

declarative reading, 30

Defamiliarizing Effect, 599

definite clause, 67

definite clause grammar, 199, 606

del_dict/4, 176

Delay Dynamics, 522

delete/3, 87

delta/3, 297

Density Distribution, 518

depth-bounded interpreter, 640

depth-first search, 70

derivable/2, 474

derivation, 68

determinism, 111, 416

  of recognizers, 323

Deviant Module, 438

diagnostic analysis pattern, 27

Dialogue, 616

dict, 175, 654

  functional notation, 176

  tag, 175

dict_pairs/3, 176

Diegetic Messages, 590

dif/2, 56, 238

digit groups, 41

digits//1, 204, 292

directive, 35

discontiguous/1, 36, 285, 352

Discontinuity, 290, 513, 648

disjunction, 71

Dispatch Level Spin, 400

display/1, 46

distinct/1, 152, 323

distinct/2, 152, 346, 422, 434, 525, 551, 560, 570

div/2, 99

Divide by Zero (User Mode), 371

domain, 241

domain_error/2, 121

Dominant Event Sequence, 557, 609

Double Free (Process Heap), 416

double negation, 635

double_free_witness/3, 417

double_quotes (flag), 48

downcase_atom/2, 133, 271, 430

dump_info/2, 276, 449

dump_relation/1, 476

dump1.pl, 63

Dynamic Memory Corruption (Process Heap), 414

dynamic predicate, 184

dynamic/1, 184, 597

E

Early Crash Dump, 454

edit/1, 32

Empty Trace, 500

empty_nb_set/1, 655

end_tests/1, 325

entails/2, 308

eol//0, 204

eos//0, 204, 265, 586

Error Distribution, 555

Error Message, 216, 565

  by an error word, 524

Error Thread, 554

error/2 (term), 121

evaluable/1, 477

evaluation, 96

evaluation error, 97

event sequence, 557

Event Sequence Order, 244, 290, 603

Event Sequence Phase, 604

evidence

  derived from proofs, 637

evidence term, 306, 335, 360, 391, 535

  contract, 306

  execution context, 306

  sufficiency, 308

evidence/2, 637

evidence/3, 306, 405

exception, 120

  ball, 120

Exception Module, 378

Exception Stack Trace, 358, 463, 585

exception_info/3, 261

exception_record/3, 261, 419

exclude/3, 168, 583

Execution Residue (User Space), 350

existence error, 122

existence_error/2, 121

exit status, 34

explainer/2, 474

explanation, 635

export, 218

F

fact, 27, 67

fact_count/2, 263

facts

  as a data structure, 654

fail/0, 72

failure-driven loop, 73, 119, 369

False Positive Dump, 455

False Positive Error, 589

false/0, 72

Fault Context, 359

fd_dom/2, 242

Fiber Bundle, 543

Fiber of Activity, 543

field/3, 263

file_name_extension/3, 271, 430

file_search_path/2, 222

findall/3, 147, 453

  copying, 148

findall/4, 148

findnsols/4, 690

flag/3, 189

float, 41

float/1, 42, 99

floating-point arithmetic, 298

floor/1, 100, 556

floundering, 118

Focus of Tracing, 494

foldl/4, 169, 490, 617

foldl/5, 171

foldl/6, 171, 293

forall/2, 118, 439

foreground components, 551

foreign language interface, 669

foreign predicate, 670

format

  column stops, 104

format directive, 102

format string, 102

format/2, 102, 466

  ~*c, 517

~f, 299

~p, 635

~s, 467

~t~*|, 635

~w (write), 422

~W (write_term), 406

format/3, 137, 211

atom(A), 268

sink, 137

Frame Regularity, 65, 92, 127, 312, 342, 463

frame template, 55

frame/5, 51, 260

frame_address/3, 365

frame_matches/3, 65

frame_param/4, 424

frames_agree/3, 269

free variable, 149

freeze/2, 240, 478

Frozen Process, 446

frozen/2, 240

functor, 44

functor/3, 139, 368, 458

G

garbage_collect/0, 660

gc_thread (flag), 664

gcd/2, 99

get_assoc/3, 174

get_dict/3, 176

global constraint, 242

global stack, 660

global variable, 189

backtrackable, 189

non-backtrackable, 189

Glued Activity, 547

Glued Stack Trace, 349

goal, 29

goal clause, 67

goal order, 73

goal reduction, 68

grammar rule, 199

ground term, 45, 65

ground/1, 45, 318

group_by/4, 153

group_pairs_by_key/2, 488, 613

Guest Component, 552

H

halt/0, 33

halt/1, 225, 666

handle, 410, 419

Handle Leak, 312, 409, 463, 626

handle_count/3, 262, 405

Handled Exception (User Space), 375

hand-off, 546

head, 29

head of a message set, 491

heap, 407

heap/4, 262, 405

heap_at/5, 411

help/1, 33

Hidden Error, 584

Hidden Exception (User Space), 374

Hidden Module, 435

High Contention (Critical Sections), 397

Historical Information, 443

holds/1, 307

Hooked Functions (User Space), 317, 433, 463

Hooked Modules, 434

hooked/3, 262

Horn clause, 67

http_handler/3, 668

http_read_json_dict/2, 668

http_server/1, 668

hypothesis, 645

  as a goal, 686

I

identification, 303

Identification Messages, 582

identity of terms, 55

if-then-else, 115, 359, 386, 602

ignore/1, 114

illustrative block, 663

Implementation Discourse, 583

Impossible Trace, 246, 614

in/2, 241

include/1, 352

include/3, 168, 331, 504, 555

incomplete verdict, 690

Inconsistent Dump, 453

Incorrect Stack Trace, 345

indexing, 234, 648

  first argument, 112, 411, 650

just-in-time, 234, 300, 650

multi-argument, 651

Indirect Message, 581

inference, 648

infinite loop, 75

initialization/2, 224

main, 663

ins/2, 241

instance/2, 463

instantiation error, 97, 626

Insufficient Memory (Committed Memory), 406, 463

Insufficient Memory (Handle Leak), 410, 463

integer, 41

unbounded, 42, 98, 423

integer//1, 204, 292, 501, 522

integer/1, 42, 100

intent

of a pattern, 303

Inter-Correlation, 301, 610

interrupt request level, 400

intersection/3, 87, 589

Intra-Correlation, 610

Invalid Handle (General), 419

Invalid Pointer (General), 362

is/2, 96

is_list/1, 47, 82

iso (flag), 98

iterative deepening, 641

J

jiti_list/1, 234, 300, 650

json_read_dict/2, 228, 666

json_write_dict/2, 228, 666

json_write_dict/3, 666

K

Karnaugh Map, 620

keysort/2, 154, 320, 412, 470, 503, 612

KiUserExceptionDispatch, 372

knowledge base, 28

L

label/1, 241, 604

labeling/2, 242

lambda expression, 171

copying, 171

free variable, 172

Last Activity, 532

Last Error Collection, 443, 463

last/2, 86, 347, 413, 493

last_error/2, 262, 419

last_status/2, 276

last-call optimization, 89

Lateral Damage, 446

lazy list, 210, 658

Least Common Frame, 339

left recursion, 75, 390

left-to-right selection, 68

length/2, 82

library(aggregate), 150

library(apply), 165

library(assoc), 173, 654

library(clpb), 244

library(clpfd), 240

library(csv), 227, 296, 667

library(dcg/basics), 203, 501, 540, 586

library(dcg/high_order), 205

library(debug), 230

library(error), 121

library(fastrw), 659

library(http/http_server), 668

library(http/json), 228, 666

library(http/thread_httpd), 668

library(lists), 85

library(main), 224, 663

library(mqi), 685

library(nb_set), 655

library(ordsets), 482

library(pairs), 154, 462

library(pcre), 229

library(persistency), 190

library(pldoc), 220

library(plunit), 231, 324

library(pprint), 329

library(pure_input), 210, 658

library(rbtrees), 175

library(readutil), 226

library(sandbox), 689

library(solution_sequences), 152

library(ugraphs), 178

library(yall), 165, 171

limit/2, 153

Linked Messages, 570

list, 46, 81

  difference list, 89

head, 81
head and tail, 59, 81
partial list, 47, 82
proper list, 82
tail, 81
list_to_assoc/2, 174
list_to_set/2, 87, 639
list_undefined/0, 233
listing/1, 33, 235, 285
load_files/2, 662
Local Buffer Overflow (User Space), 365
local stack, 660
LockCount, 261
decoding, 272
log, 27, 289
log_complete/1, 418
log_line/3, 265
logical consequence, 68
logical update view, 186

M

Machine Query Interface, 685
Macrofunction, 606
Main Thread, 332, 451
main/0, 224
make/0, 33
Managed Code Exception, 376
Manual Dump (Process), 320, 451
map_list_to_pairs/3, 320
maplist/2, 167, 440
maplist/3, 167, 268, 293, 339, 527, 575
maplist/4, 167
maplist/5, 167
Marked Message, 591
Master Trace, 255, 301, 611
max/2, 99, 543
max_list/2, 87, 343
member/2, 83
memberchk/2, 114, 332, 386, 414, 451, 483
memory dump, 27
complete, 284
kernel, 284
process, 258
Memory Fluctuation, 413
Memory Leak (Process Heap), 411, 460
message, 289, 564
Message Annotations, 596

Message Change, 587

Message Context, 580

Message Cover, 597

Message Directory, 503

Message Interleave, 556

Message Invariant, 567

message kind, 501

Message Set, 482, 485

Message Space, 490

meta_predicate/1, 222

meta-argument, 164

meta-interpreter, 626

meta-predicate, 164

  in meta-interpreters, 632

Milestones, 531

min/2, 99, 543

min_list/2, 87, 389, 460

Minimal Trace, 617

Missing Component (General), 432

Missing Message, 250, 575

missing/3, 280

mod/2, 99

mode, 57

module, 27, 218

module context

  in a meta-interpreter, 686

Module Hint, 442

module qualification, 221, 449

Module Variety, 158, 427

module/2, 218

module/3, 260

module_info/3, 260

module_property/2, 221, 629

  class, 629

most general unifier, 57

Motif, 607

mqi_start/1, 685

msb/1, 99

msg/6, 290

msg/7, 301

msort/2, 86, 268

multifile/1, 449, 585, 611

Multiple Exceptions (User Space), 373

must_be/2, 121, 386, 493, 515

mutant/3, 383

MYCIN, 645

N

nb_getval/2, 189

nb_set_to_list/2, 655

nb_setval/2, 189

negation as failure, 90, 117, 418, 454, 584, 614

in proof trees, 634

safety, 454

Nested Exceptions (User Space), 372

nextto/3, 343, 511, 571

No Activity, 533

No Component Symbols, 53, 145, 216, 274, 317, 431, 463

No Current Thread, 458

nodebug/0, 76

nonblanks//1, 204

nondeterministic, 111

none, 274

nonterminal, 199

nonvar/1, 43

normalization, 271

nospy/1, 77

Not My Version, 429, 463

notrace/0, 76

nth0/3, 86, 312, 350, 580

nth1/3, 86

null page, 358

NULL Pointer (Code), 361

NULL Pointer (Data), 106, 312, 360, 425, 463

number, 28, 41

number//1, 204

number/1, 42

number_codes/2, 133

number_string/2, 133, 523

numbervars/3, 139, 635

numlist/3, 87, 527, 577

O

occurs check, 59

occurs_check (flag), 60

offset/2, 153

once/1, 114, 307, 416, 525, 555, 574

on-demand facts, 672

open/3, 211, 225

operator, 45

priority, 45

Opposition Messages, 572

opt_help/2, 664

opt_type/3, 663

option/3, 690

ord_disjoint/2, 486

ord_intersection/3, 486, 543

ord_subtract/3, 486, 551

ord_union/3, 486

order_by/2, 153, 402, 410, 533

ordinal (in a message), 576

ordinal number, 289

Origin Module, 441

P

Paged Out Data, 420

pair, 154

- key, 154
- value, 154

pairs_keys/2, 154

pairs_keys_values/3, 154

pairs_values/2, 154, 389, 462

Palimpsest Messages, 592

parallel edges, 389

Paratext, 509

parse_record/4, 268

partial order, 320

partial structure, 59

partition/4, 169

partition/6, 169

Passive Thread, 37, 63, 125, 334

pattern

- composition, 460
- enumerator form, 408, 535
- explainer form, 408, 535

pattern description, 303

pattern instance, 27, 303

pattern matching, 58

pattern recognizer, 303

pattern/3, 316, 355, 383

pattern_instances/1, 317

pattern_name/2, 637

pattern-oriented software diagnostics, 27

patterns_dump, 449, 686

Periodic Error, 523

Periodic Message Block, 525

periodicity, 524

permission error, 628

persistence (data structure), 174

phrase/2, 199, 540

phrase/3, 200

phrase_from_file/2, 210, 658

pipeline

  debugger log to facts, 264

Pivot Message, 579

PL_call, 669

PL_initialise, 669

PL_open_query, 670

PL_unify_atom_chars, 670

plus/3, 97

port, 76

  call, 76

  exit, 76

  fail, 76

  redo, 76

portray/1, 422, 635

portray/1 (hook), 138

portray_clause/1, 138, 235, 267, 536, 628

portray_clause/2, 268, 661

ports (call, exit, redo, fail), 643

predicate, 28

predicate_property/2, 188, 221, 301, 450, 629

  built_in, 629

  imported_from, 629

predsort/3, 155, 320, 612

prefer_rationals (flag), 98

prefer_rationals flag, 408

presence (of a component), 549

principal functor, 44

print/1, 138, 293, 635

print_message/2, 122

print_messages/1, 482

print_term/2, 329

Problem Module, 379

procedural reading, 30

procedure, 28

procedure box model, 76

Procedure Call Chain, 344

process/2, 260

profile/1, 662

program, 28, 68

PrologWinDbg, 673

proof

  size, 642

proof tree, 626

proof/2, 632

propagation, 241

protect_static_code (flag), 628

prove/3, 686

provenance, 275, 647

provenance-preserving enrichment, 647

Punctuated Activity, 529

pushback (DCG), 203

put_assoc/4, 174

put_dict/4, 176

Q

qcompile/1, 661

quantification

  universal, 67

query, 27

query vocabulary, 688

quick-load file, 661

Quotient Trace, 613

R

RaiseException, 376

ran/2, 276

rational number, 41

rational/1, 43

rational_syntax (flag), 43

raw_stack/3, 347, 421

re_foldl/6, 229

re_match/2, 229

re_matchsub/3, 229

reachable/3, 179

reaches/2, 74

read_facts/2, 269

read_file_to_string/3, 226

read_line_to_string/2, 226, 265, 659

read_term/2, 226, 659

read_term/3, 269

read_term_from_atom/3, 137

recognition, 303

recognizer

  enumerator, 304

  explainer, 304

  test form, 304

recognizer/2, 316

Recovered Messages, 595

recursion, 82

  base case, 74

  recursive case, 74

  structural, 82

reexport/1, 222

Regular Data, 423

reification, 243

Relative Density, 520

rem/2, 99

remainder//1, 204, 292, 540

repeat/0, 120

reply_json_dict/1, 668

report/0, 467

residual goal, 238

resolvent, 68

Resume Activity, 548

retract/1, 185

retractall/1, 186, 395

reverse/2, 86

round/1, 100, 510

rule, 27, 67

rule/3, 645

run_tests/0, 232, 325

S

safe_goal/1, 689

safety check, 689

Same Vendor, 439

schema

  census, 457

  extension, 405, 449

  memory dump, 259

  software trace, 290

script, 223

search

  constraint, 241

search tree, 70

  failure leaf, 70

  success leaf, 70

select/3, 87

selectchk/3, 476

Semantic Mapping, 618

semideterministic, 111

sequence//2, 205

sequence//3, 205

sequence//5, 205

set_prolog_flag/2, 60

set_prolog_gc_thread/1, 664

setarg/3, 141

setof/3, 149, 299, 440

setup_call_cleanup/3, 122

severity, 318

severity/2, 318, 463

Shared Buffer Overwrite, 421

Shared Point, 302, 571

Sheaf of Activities, 544

shell/1, 665

shell/2, 665

show_profile/1, 662

sign/1, 99

Significant Event, 566

Significant Interval, 528

silence, 533

Silent Messages, 584

Singleton Event, 501

singleton variable, 35

SLD resolution, 68

SLG resolution, 190

Small DA+TA, 296, 622

snapshot (PrologWinDbg), 673

soft cut, 117

software diagnostics, 27

software trace, 27, 289, 482

solve/1, 630

solve/2, 632

solve_count/2, 642

solve_depth/2, 640

sort

  stable, 154

sort/2, 86, 394, 503

sort/4, 86, 610

source/3, 275

Sparse Trace, 499

Special Process, 446

Special Thread, 333

Spiking Thread, 108, 182, 235, 304, 335, 463, 626

Split Trace, 252, 611

split_string/4, 136, 487, 560

spy point, 77

spy/1, 77

sqrt/1, 100

Stack Overflow (User Mode), 65, 366, 463, 645

Stack Trace, 27, 90, 216, 235, 259, 329

Stack Trace Collection (Predicate), 331

Stack Trace Collection (Unmanaged Space), 157, 330

Stack Trace Set, 341

Stack Trace Signature, 338

stack_limit (flag), 75, 660

standard order of terms, 49, 504

State Dump, 621

statement current, 515

statement density, 517

Statement Density and Current, 515

static predicate, 184

statistics/2, 233, 643, 649

  cputime, 650

  garbage_collection, 660

  globalused, 660

  runtime, 650

status code, 355

  severity, 456

status_name/2, 355

steadfastness, 114, 602

Step Dumps, 460

Stored Exception, 377

stored_exception/4, 377

stream, 225

string, 47, 132, 291

string//1, 204

string/1, 47

string_chars/2, 132

string_code/3, 136

string_codes/2, 132, 199

string_concat/3, 134, 523, 592

string_length/2, 133

string_lower/2, 133, 487, 539

string_upper/2, 133

string_without//2, 204, 265, 587

sub_atom/5, 135, 442

  extraction, 135

  prefix, 135

  search, 135

  suffix, 135

sub_string/5, 135, 498, 565

subgoal, 29

substitution, 57

subsumes_term/2, 61

subtract/3, 87, 577

succ/2, 90, 97

sufficient/1, 474

sum/3, 243

sum_list/2, 87

suspected/3, 307, 366

Suspended Thread, 37, 125, 463

suspicion term, 307, 366

suspicion/2, 463

Swarm of Shared Locks, 400

swiplserver, 685

SWI-Prolog, 31

symbol_parts/4, 142

syntax error, 34

syntax_error, 268

system (module), 218

system_module/1, 355, 403

T

table (tabling), 190

  complete, 190

table/1, 190, 390, 502, 557

  as memoization, 656

tabling, 190, 390

  incremental, 657

tail of a message set, 492

tail recursion, 89

template/2, 63

term, 28

term reference, 669

term_string/2, 137

term_string/3, 689

term_to_atom/2, 137

term_variables/2, 62

terminal, 199

termination, 74

test/2, 231, 325

thread, 27

Thread Age, 338

Thread of Activity, 299, 537

Thread Starvation (Normal Priority), 402

thread/2, 260

thread_absence/3, 281

thread_coverage/3, 279

thread_messages/2, 297

thread_state/2, 260

thread_time/3, 260

threshold, 309

threshold/2, 309, 335, 355, 383, 405, 511

throw/1, 120

Time Delta, 236, 290, 511, 664

time/1, 194, 233, 301, 648

Timeout, 527

timestamp, 289

Top Module, 440

top_sort/2, 179

toplevel, 29

toplevel_print_anon, 329, 483

toplevel_print_anon (flag), 90

Trace Acceleration, 521

Trace Dimension, 484

Trace Extension, 496

Trace Frames, 493

trace identifier, 301

Trace Mask, 496

Trace Partition, 291, 489

Trace Summary, 505

Trace Viewpoints, 506

trace/0, 76

tracer, 76

tracing meta-interpreter, 643

trail, 660

transitive closure, 74, 382

transitive_closure/2, 179

Translated Message, 593

transpose_pairs/2, 154

transpose_ugraph/2, 179

true/0, 67

truncate/1, 100

Truncated Dump, 323, 452

Truncated Stack Trace, 346

Truncated Trace, 498

tuples_in/2, 243

type test, 48

type_error/2, 121

U

Ubiquitous Component, 447

ugraph, 178

unifiability, 55

unification, 55

unify_with_occurs_check/2, 60

union/3, 87

univ, 140

unknown

'Unknown', 274

unknown (flag), 122

Unknown Component, 53, 430, 463

Unknown frame, 432

upcase_atom/2, 133

Use Case Trail, 561

use_foreign_library/1, 671

use_module/1, 220, 449

use_module/2, 626

user (module), 218

V

Value References, 421

vanilla meta-interpreter, 630

var/1, 43

variable, 29, 43

  aliasing, 56

  unbound, 43

vendor/2, 427

verdict_dict/3, 695

vertices_edges_to_ugraph/3, 179

Virtualized Process (WOW64), 437

Visibility Limit, 500

visited list, 382, 460

Visitor Trace, 618

Vocabulary Index, 161, 487

W

Wait Chain (Critical Sections), 78, 95, 180, 195, 235, 312, 384, 459, 640

Wait Chain (Mutex Objects), 386

Wait Chain (Thread Objects), 387

wait_fact/4, 383

wait_time/2, 401

WaiterCount, 261

wait-for graph, 382

WaitForSingleObject, 386

Waiting Thread Time, 401

waits/3, 383

waits_for/3, 65, 77, 261, 383

waits_for_object/3, 383

warning, 35

Watch Thread, 553

wd (module), 673

when/2, 240

whites//0, 204

why/1, 635

Wild Code, 364

Wild Pointer, 363

window (time), 515

with_fact/2, 282

with_output_to/2, 138

without_facts/2, 282

witness

of a deadlock, 388

worker process (PrologWinDbg), 673

Working Set, 508

write_canonical/1, 40

write_facts/2, 294

write_term/2, 138

max_depth, 138

portray option, 635

quoted, 138

X

xinteger//1, 204, 265, 501, 587

xor/2, 100

Z

zcompare/3, 243

Zombie Processes, 446