The Logtalk Handbook

Release v3.92.0

Paulo Moura

May 28, 2025

CONTENTS

1 User Manual **1**

1.1 Declarative object-oriented programming . . . 1
1.2 Main features . . . 2
1.2.1 Integration of logic and object-oriented programming . . . 2
1.2.2 Integration of event-driven and object-oriented programming . . . 2
1.2.3 Support for component-based programming . . . 3
1.2.4 Support for both prototype and class-based systems . . . 3
1.2.5 Support for multiple object hierarchies . . . 3
1.2.6 Separation between interface and implementation . . . 3
1.2.7 Private, protected and public inheritance . . . 3
1.2.8 Private, protected and public object predicates . . . 4
1.2.9 Parametric objects . . . 4
1.2.10 High level multi-threading programming support . . . 4
1.2.11 Smooth learning curve . . . 4
1.2.12 Compatibility with most Prolog systems and the ISO standard . . . 4
1.2.13 Performance . . . 4
1.2.14 Logtalk scope . . . 5
1.3 Nomenclature . . . 6
1.3.1 Prolog nomenclature . . . 6
1.3.2 Smalltalk nomenclature . . . 8
1.3.3 C++ nomenclature . . . 9
1.3.4 Java nomenclature . . . 11
1.3.5 Python nomenclature . . . 12
1.4 Messages . . . 14
1.4.1 Operators used in message-sending . . . 14
1.4.2 Sending a message to an object . . . 14
1.4.3 Delegating a message to an object . . . 15
1.4.4 Sending a message to *self* . . . 15
1.4.5 Broadcasting . . . 15
1.4.6 Calling imported and inherited predicates . . . 15
1.4.7 Message sending and event generation . . . 16
1.4.8 Sending a message from a module . . . 17
1.4.9 Message sending performance . . . 17
1.5 Objects . . . 17
1.5.1 Objects, prototypes, classes, and instances . . . 17
1.5.2 Defining a new object . . . 18
1.5.3 Parametric objects . . . 21
1.5.4 Finding defined objects . . . 23
1.5.5 Creating a new object in runtime . . . 23
1.5.6 Abolishing an existing object . . . 24

1.5.7 Object directives . . . 24
1.5.8 Object relationships . . . 26
1.5.9 Object properties . . . 27
1.5.10 Built-in objects . . . 29
1.5.11 Multi-threading applications . . . 30
1.6 Protocols . . . 30
1.6.1 Defining a new protocol . . . 30
1.6.2 Finding defined protocols . . . 31
1.6.3 Creating a new protocol in runtime . . . 31
1.6.4 Abolishing an existing protocol . . . 31
1.6.5 Protocol directives . . . 32
1.6.6 Protocol relationships . . . 32
1.6.7 Protocol properties . . . 33
1.6.8 Implementing protocols . . . 34
1.6.9 Built-in protocols . . . 35
1.6.10 Multi-threading applications . . . 35
1.7 Categories . . . 35
1.7.1 Defining a new category . . . 36
1.7.2 Hot patching . . . 38
1.7.3 Finding defined categories . . . 39
1.7.4 Creating a new category in runtime . . . 39
1.7.5 Abolishing an existing category . . . 40
1.7.6 Category directives . . . 40
1.7.7 Category relationships . . . 41
1.7.8 Category properties . . . 42
1.7.9 Importing categories . . . 44
1.7.10 Calling category predicates . . . 44
1.7.11 Parametric categories . . . 45
1.7.12 Built-in categories . . . 46
1.7.13 Multi-threading applications . . . 46
1.8 Predicates . . . 46
1.8.1 Reserved predicate names . . . 46
1.8.2 Declaring predicates . . . 46
1.8.3 Defining predicates . . . 58
1.8.4 Definite clause grammar rules . . . 64
1.8.5 Built-in methods . . . 67
1.8.6 Predicate properties . . . 71
1.8.7 Finding declared predicates . . . 73
1.8.8 Calling Prolog predicates . . . 73
1.8.9 Defining Prolog multifile predicates . . . 78
1.8.10 Asserting and retracting Prolog predicates . . . 78
1.9 Inheritance . . . 79
1.9.1 Protocol inheritance . . . 80
1.9.2 Implementation inheritance . . . 80
1.9.3 Public, protected, and private inheritance . . . 84
1.9.4 Multiple inheritance . . . 85
1.9.5 Composition versus multiple inheritance . . . 85
1.10 Event-driven programming . . . 85
1.10.1 Definitions . . . 86
1.10.2 Event generation . . . 87
1.10.3 Communicating events to monitors . . . 87
1.10.4 Performance concerns . . . 87
1.10.5 Monitor semantics . . . 87
1.10.6 Activation order of monitors . . . 88

1.10.7 Event handling . . . 88
1.11 Multi-threading programming . . . 90
1.11.1 Enabling multi-threading support . . . 90
1.11.2 Enabling objects to make multi-threading calls . . . 91
1.11.3 Multi-threading built-in predicates . . . 91
1.11.4 One-way asynchronous calls . . . 93
1.11.5 Asynchronous calls and synchronized predicates . . . 94
1.11.6 Synchronizing threads through notifications . . . 95
1.11.7 Threaded engines . . . 95
1.11.8 Multi-threading performance . . . 97
1.12 Error handling . . . 97
1.12.1 Raising Exceptions . . . 97
1.12.2 Type-checking . . . 98
1.12.3 Expected terms . . . 99
1.12.4 Compiler warnings and errors . . . 99
1.12.5 Runtime errors . . . 101
1.13 Reflection . . . 102
1.13.1 Structural reflection . . . 102
1.13.2 Behavioral reflection . . . 103
1.14 Writing and running applications . . . 103
1.14.1 Starting Logtalk . . . 103
1.14.2 Running parallel Logtalk processes . . . 104
1.14.3 Source files . . . 105
1.14.4 Multi-pass compiler . . . 105
1.14.5 Compiling and loading your applications . . . 106
1.14.6 Compiler errors, warnings, and comments . . . 107
1.14.7 Loader files . . . 107
1.14.8 Libraries of source files . . . 108
1.14.9 Settings files . . . 109
1.14.10 Compiler linter . . . 110
1.14.11 Compiler flags . . . 110
1.14.12 Reloading source files . . . 118
1.14.13 Batch processing . . . 118
1.14.14 Optimizing performance . . . 118
1.14.15 Portable applications . . . 119
1.14.16 Conditional compilation . . . 119
1.14.17 Avoiding common errors . . . 119
1.14.18 Coding style guidelines . . . 120
1.15 Printing messages and asking questions . . . 120
1.15.1 Printing messages . . . 121
1.15.2 Message tokenization . . . 122
1.15.3 Meta-messages . . . 123
1.15.4 Defining message prefixes and output streams . . . 123
1.15.5 Defining message prefixes and output files . . . 123
1.15.6 Intercepting messages . . . 124
1.15.7 Asking questions . . . 125
1.15.8 Intercepting questions . . . 126
1.15.9 Multi-threading applications . . . 126
1.16 Term and goal expansion . . . 127
1.16.1 Defining expansions . . . 127
1.16.2 Expanding grammar rules . . . 129
1.16.3 Bypassing expansions . . . 129
1.16.4 Hook objects . . . 129
1.16.5 Virtual source file terms and loading context . . . 131

1.16.6 Default compiler expansion workflow . . . 132
1.16.7 User defined expansion workflows . . . 132
1.16.8 Using Prolog defined expansions . . . 132
1.16.9 Debugging expansions . . . 133
1.17 Documenting . . . 134
1.17.1 Entity documenting directives . . . 135
1.17.2 Predicate documenting directives . . . 136
1.17.3 Describing predicates . . . 137
1.17.4 Documenting predicate exceptions . . . 138
1.17.5 Processing and viewing documenting files . . . 139
1.17.6 Inline formatting in comments text . . . 140
1.17.7 Diagrams . . . 141
1.18 Debugging . . . 141
1.18.1 Compiling source files in debug mode . . . 141
1.18.2 Procedure box model . . . 142
1.18.3 Activating the debugger . . . 143
1.18.4 Defining breakpoints . . . 143
1.18.5 Defining log points . . . 147
1.18.6 Tracing program execution . . . 148
1.18.7 Debugging using breakpoints . . . 149
1.18.8 Debugging commands . . . 149
1.18.9 Customizing term writing . . . 151
1.18.10 Context-switching calls . . . 152
1.18.11 Debugging messages . . . 153
1.18.12 Using the term-expansion mechanism for debugging . . . 155
1.18.13 Ports profiling . . . 155
1.18.14 Debug and trace events . . . 155
1.18.15 Source-level debugger . . . 156
1.19 Performance . . . 156
1.19.1 Source code compilation modes . . . 157
1.19.2 Source code compilation order . . . 157
1.19.3 Local predicate calls . . . 157
1.19.4 Calls to imported or inherited predicates . . . 157
1.19.5 Calls to module predicates . . . 157
1.19.6 Messages . . . 157
1.19.7 Automatic expansion of built-in meta-predicates . . . 158
1.19.8 Inlining . . . 158
1.19.9 Generated code simplification and optimizations . . . 158
1.19.10 Size of the generated code . . . 159
1.19.11 Circular references . . . 159
1.19.12 Debug mode overhead . . . 159
1.19.13 Other considerations . . . 159
1.20 Installing Logtalk . . . 160
1.20.1 Hardware and software requirements . . . 160
1.20.2 Logtalk installers . . . 161
1.20.3 Source distribution . . . 161
1.20.4 Distribution overview . . . 161
1.21 Prolog integration and migration . . . 164
1.21.1 Source files with both Prolog code and Logtalk code . . . 164
1.21.2 Encapsulating plain Prolog code in objects . . . 165
1.21.3 Converting Prolog modules into objects . . . 166
1.21.4 Compiling Prolog modules as objects . . . 167
1.21.5 Dealing with proprietary Prolog directives and predicates . . . 169
1.21.6 Calling Prolog module predicates . . . 170

1.21.7 Loading converted Prolog applications . . . 170

2 Reference Manual **173**
2.1 Grammar . . . 173
2.1.1 Entities . . . 173
2.1.2 Object definition . . . 173
2.1.3 Category definition . . . 174
2.1.4 Protocol definition . . . 174
2.1.5 Entity relations . . . 175
2.1.6 Entity identifiers . . . 178
2.1.7 Source files . . . 179
2.1.8 Source file names . . . 179
2.1.9 Terms . . . 180
2.1.10 Directives . . . 180
2.1.11 Clauses and goals . . . 188
2.1.12 Lambda expressions . . . 189
2.1.13 Entity properties . . . 190
2.1.14 Predicate properties . . . 193
2.1.15 Compiler flags . . . 194
2.2 Control constructs . . . 194
2.2.1 Message sending . . . 194
2.2.2 Message delegation . . . 197
2.2.3 Calling imported and inherited predicates . . . 198
2.2.4 Calling predicates in *this* . . . 199
2.2.5 Calling external predicates . . . 200
2.2.6 Context switching calls . . . 202
2.3 Directives . . . 204
2.3.1 Source file directives . . . 204
2.3.2 Conditional compilation directives . . . 210
2.3.3 Entity directives . . . 214
2.3.4 Predicate directives . . . 227
2.4 Built-in predicates . . . 245
2.4.1 Enumerating objects, categories and protocols . . . 245
2.4.2 Enumerating objects, categories and protocols properties . . . 248
2.4.3 Creating new objects, categories and protocols . . . 251
2.4.4 Abolishing objects, categories and protocols . . . 255
2.4.5 Objects, categories, and protocols relations . . . 258
2.4.6 Event handling . . . 268
2.4.7 Multi-threading . . . 271
2.4.8 Multi-threading engines . . . 282
2.4.9 Compiling and loading source files . . . 291
2.4.10 Flags . . . 306
2.4.11 Linter . . . 309
2.5 Built-in methods . . . 310
2.5.1 Logic and control . . . 310
2.5.2 Execution context . . . 313
2.5.3 Reflection . . . 318
2.5.4 Database . . . 322
2.5.5 Meta-calls . . . 331
2.5.6 Error handling . . . 336
2.5.7 All solutions . . . 353
2.5.8 Event handling . . . 359
2.5.9 Message forwarding . . . 360
2.5.10 Definite clause grammar rules . . . 361

2.5.11 Term and goal expansion . . . 367
2.5.12 Coinduction hooks . . . 371
2.5.13 Message printing . . . 373
2.5.14 Question asking . . . 379

3 Tutorial **383**
3.1 List predicates . . . 383
3.1.1 Defining a list object . . . 383
3.1.2 Defining a list protocol . . . 384
3.1.3 Summary . . . 386
3.2 Dynamic object attributes . . . 386
3.2.1 Defining a category . . . 386
3.2.2 Importing the category . . . 387
3.2.3 Summary . . . 388
3.3 A reflective class-based system . . . 388
3.3.1 Defining the base classes . . . 388
3.3.2 Summary . . . 389
3.4 Profiling programs . . . 389
3.4.1 Messages as events . . . 390
3.4.2 Profilers as monitors . . . 390
3.4.3 Summary . . . 392

4 FAQ **393**
4.1 General . . . 393
4.1.1 Why are all versions of Logtalk numbered 2.x or 3.x? . . . 393
4.1.2 Why do I need a Prolog compiler to use Logtalk? . . . 393
4.1.3 Is the Logtalk implementation based on Prolog modules? . . . 393
4.1.4 Does the Logtalk implementation use term-expansion? . . . 394
4.2 Compatibility . . . 394
4.2.1 What are the backend Prolog compiler requirements to run Logtalk? . . . 394
4.2.2 Can I use constraint-based packages with Logtalk? . . . 394
4.2.3 Can I use Logtalk objects and Prolog modules at the same time? . . . 394
4.3 Installation . . . 394
4.3.1 The integration scripts/shortcuts are not working! . . . 394
4.3.2 I get errors when starting up Logtalk after upgrading to the latest version! . . . 395
4.4 Portability . . . 395
4.4.1 Are my Logtalk applications portable across Prolog compilers? . . . 395
4.4.2 Are my Logtalk applications portable across operating systems? . . . 395
4.5 Programming . . . 395
4.5.1 Should I use prototypes or classes in my application? . . . 396
4.5.2 Can I use both classes and prototypes in the same application? . . . 396
4.5.3 Can I mix classes and prototypes in the same hierarchy? . . . 396
4.5.4 Can I use a protocol or a category with both prototypes and classes? . . . 396
4.5.5 What support is provided in Logtalk for defining and using components? . . . 396
4.5.6 What support is provided in Logtalk for reflective programming? . . . 396
4.6 Troubleshooting . . . 396
4.6.1 Using compiler options on calls to the Logtalk compiling and loading predicates does not work! . . . 397
4.6.2 Gecko-based browsers (e.g., Firefox) show non-rendered HTML entities when browsing XML documenting files! . . . 397
4.6.3 Compiling a source file results in errors or warnings but the Logtalk compiler reports a successful compilation with zero errors and zero warnings! . . . 397
4.7 Usability . . . 397
4.7.1 Is there a shortcut for compiling and loading source files? . . . 397

4.7.2 Is there an equivalent directive to the ensure_loaded/1 Prolog directive? . . . 398
4.7.3 Are there shortcuts for the make functionality? . . . 398
4.8 Deployment . . . 398
4.8.1 Can I create standalone applications with Logtalk? . . . 398
4.9 Performance . . . 398
4.9.1 Is Logtalk implemented as a meta-interpreter? . . . 398
4.9.2 What kind of code Logtalk generates when compiling objects? Dynamic code? Static code? . . . 399
4.9.3 How about message-sending performance? Does Logtalk use static binding or dynamic binding? . . . 399
4.9.4 Which Prolog-dependent factors are most crucial for good Logtalk performance? . . . 399
4.9.5 How does Logtalk performance compare with plain Prolog and with Prolog modules? 399
4.10 Licensing . . . 399
4.10.1 What's the Logtalk distribution license? . . . 400
4.10.2 Can Logtalk be used in commercial applications? . . . 400
4.10.3 What's the final license for a combination of Logtalk with a Prolog compiler? . . . 400
4.11 Support . . . 400
4.11.1 Are there professional consulting, training and supporting services? . . . 400

5 Developer Tools 401
5.1 Overview . . . 401
5.1.1 Loading the developer tools . . . 402
5.1.2 Tools documentation . . . 402
5.1.3 Tools common flags . . . 402
5.1.4 Tools requirements . . . 402
5.2 asdf . . . 405
5.3 assertions . . . 405
5.3.1 API documentation . . . 405
5.3.2 Loading . . . 405
5.3.3 Testing . . . 405
5.3.4 Adding assertions to your source code . . . 406
5.3.5 Automatically adding file and line context information to assertions . . . 406
5.3.6 Suppressing assertion calls from source code . . . 406
5.3.7 Redirecting assertion failure messages . . . 406
5.3.8 Converting assertion failures into errors . . . 407
5.4 code_metrics . . . 407
5.4.1 API documentation . . . 407
5.4.2 Loading . . . 408
5.4.3 Testing . . . 408
5.4.4 Available metrics . . . 408
5.4.5 Coupling metrics . . . 408
5.4.6 Halstead metric . . . 409
5.4.7 UPN metric . . . 410
5.4.8 Cyclomatic complexity metric . . . 410
5.4.9 Usage . . . 411
5.4.10 Excluding code from analysis . . . 411
5.4.11 Defining new metrics . . . 411
5.4.12 Third-party tools . . . 411
5.4.13 Applying metrics to Prolog modules . . . 411
5.4.14 Applying metrics to plain Prolog code . . . 412
5.5 dead_code_scanner . . . 412
5.5.1 API documentation . . . 413
5.5.2 Loading . . . 413
5.5.3 Testing . . . 413

5.5.4 Usage . . . 413
5.5.5 Excluding code from analysis . . . 414
5.5.6 Integration with the make tool . . . 414
5.5.7 Caveats . . . 414
5.5.8 Scanning Prolog modules . . . 414
5.5.9 Scanning plain Prolog files . . . 414
5.6 debug_messages . . . 415
5.6.1 API documentation . . . 415
5.6.2 Loading . . . 415
5.6.3 Testing . . . 415
5.6.4 Usage . . . 416
5.7 debugger . . . 417
5.7.1 API documentation . . . 417
5.7.2 Loading . . . 417
5.7.3 Testing . . . 417
5.7.4 Usage . . . 417
5.7.5 Alternative debugger tools . . . 418
5.7.6 Known issues . . . 418
5.8 diagrams . . . 418
5.8.1 Requirements . . . 419
5.8.2 API documentation . . . 420
5.8.3 Loading . . . 420
5.8.4 Testing . . . 420
5.8.5 Supported diagrams . . . 420
5.8.6 Graph elements . . . 421
5.8.7 Supported graph languages . . . 422
5.8.8 Customization . . . 423
5.8.9 Linking diagrams . . . 427
5.8.10 Creating diagrams for Prolog module applications . . . 429
5.8.11 Creating diagrams for plain Prolog files . . . 429
5.8.12 Other notes . . . 430
5.9 doclet . . . 430
5.9.1 API documentation . . . 431
5.9.2 Loading . . . 431
5.9.3 Automating running doclets . . . 431
5.9.4 Integration with the make tool . . . 431
5.10 help . . . 431
5.10.1 Requirements . . . 431
5.10.2 API documentation . . . 431
5.10.3 Loading . . . 432
5.10.4 Testing . . . 432
5.10.5 Supported operating-systems . . . 432
5.10.6 Usage . . . 432
5.10.7 Experimental features . . . 432
5.10.8 Known issues . . . 433
5.11 issue_creator . . . 433
5.11.1 Requirements . . . 433
5.11.2 Loading . . . 434
5.11.3 Usage . . . 434
5.11.4 Known issues . . . 435
5.12 lgtdoc . . . 435
5.12.1 Requirements . . . 435
5.12.2 API documentation . . . 436
5.12.3 Loading . . . 436

5.12.4 Testing . . . 436
5.12.5 Documenting source code . . . 436
5.12.6 Generating documentation . . . 437
5.12.7 Documentation linter checks . . . 437
5.13 lgtunit . . . 438
5.13.1 Main files . . . 438
5.13.2 API documentation . . . 438
5.13.3 Loading . . . 438
5.13.4 Testing . . . 439
5.13.5 Writing and running tests . . . 439
5.13.6 Automating running tests . . . 441
5.13.7 Parametric test objects . . . 442
5.13.8 Test dialects . . . 443
5.13.9 User-defined test dialects . . . 445
5.13.10 QuickCheck . . . 446
5.13.11 Skipping tests . . . 450
5.13.12 Selecting tests . . . 451
5.13.13 Checking test goal results . . . 451
5.13.14 Testing local predicates . . . 453
5.13.15 Testing non-deterministic predicates . . . 453
5.13.16 Testing generators . . . 453
5.13.17 Testing input/output predicates . . . 454
5.13.18 Suppressing tested predicates output . . . 455
5.13.19 Tests with timeout limits . . . 456
5.13.20 Setup and cleanup goals . . . 456
5.13.21 Test annotations . . . 456
5.13.22 Test execution times and memory usage . . . 457
5.13.23 Working with test data files . . . 457
5.13.24 Flaky tests . . . 458
5.13.25 Mocking . . . 458
5.13.26 Debugging messages in tests . . . 459
5.13.27 Debugging failed tests . . . 460
5.13.28 Code coverage . . . 460
5.13.29 Utility predicates . . . 462
5.13.30 Exporting test results in xUnit XML format . . . 463
5.13.31 Exporting test results in the TAP output format . . . 464
5.13.32 Generating Allure reports . . . 464
5.13.33 Exporting code coverage results in XML format . . . 466
5.13.34 Automatically creating bug reports at issue trackers . . . 467
5.13.35 Minimizing test results output . . . 467
5.13.36 Help with warnings . . . 467
5.13.37 Known issues . . . 467
5.14 linter . . . 467
5.14.1 Main linter checks . . . 468
5.14.2 Help on linter warnings . . . 469
5.14.3 Extending the linter . . . 469
5.14.4 Linting Prolog modules . . . 469
5.14.5 Linting plain Prolog files . . . 469
5.15 make . . . 470
5.15.1 API documentation . . . 470
5.15.2 Help with warnings . . . 470
5.16 packs . . . 470
5.16.1 Requirements . . . 471
5.16.2 API documentation . . . 472

5.16.3 Loading . . . 472
5.16.4 Testing . . . 472
5.16.5 Usage . . . 472
5.16.6 Registries and packs storage . . . 472
5.16.7 Virtual environments . . . 473
5.16.8 Registry specification . . . 474
5.16.9 Registry handling . . . 476
5.16.10 Registry development . . . 477
5.16.11 Pack specification . . . 477
5.16.12 Encrypted packs . . . 478
5.16.13 Signed packs . . . 479
5.16.14 Pack URLs and Single Sign-On . . . 479
5.16.15 Multiple pack versions . . . 480
5.16.16 Pack dependencies . . . 480
5.16.17 Pack portability . . . 481
5.16.18 Pack development . . . 481
5.16.19 Pack handling . . . 482
5.16.20 Pack documentation . . . 484
5.16.21 Pinning registries and packs . . . 485
5.16.22 Testing packs . . . 485
5.16.23 Security considerations . . . 486
5.16.24 Best practices . . . 486
5.16.25 Installing Prolog packs . . . 486
5.16.26 Help with warnings . . . 487
5.16.27 Known issues . . . 487
5.17 `ports_profiler` . . . 487
5.17.1 API documentation . . . 488
5.17.2 Loading . . . 488
5.17.3 Testing . . . 488
5.17.4 Compiling source files for port profiling . . . 488
5.17.5 Generating profiling data . . . 488
5.17.6 Printing profiling data reports . . . 489
5.17.7 Interpreting profiling data . . . 490
5.17.8 Profiling Prolog modules . . . 491
5.17.9 Profiling plain Prolog code . . . 491
5.17.10 Known issues . . . 492
5.18 `profiler` . . . 492
5.18.1 Loading . . . 492
5.18.2 Testing . . . 492
5.18.3 Supported backend Prolog compilers . . . 492
5.18.4 Compiling source code for profiling . . . 493
5.19 `tutor` . . . 493
5.19.1 API documentation . . . 493
5.19.2 Loading . . . 493
5.19.3 Usage . . . 494
5.20 `wrapper` . . . 494
5.20.1 API documentation . . . 495
5.20.2 Loading . . . 495
5.20.3 Workflows . . . 495
5.20.4 Customization . . . 495
5.20.5 Current limitations . . . 496

6 Libraries **497**
6.1 Overview . . . 497

6.1.1 Library documentation . . . 497
6.1.2 Loading libraries . . . 497
6.1.3 Testing libraries . . . 498
6.1.4 Credits . . . 498
6.1.5 Other notes . . . 498
6.2 arbitrary . . . 498
6.2.1 API documentation . . . 498
6.2.2 Loading . . . 499
6.2.3 Testing . . . 499
6.2.4 Pre-defined types . . . 499
6.2.5 Usage . . . 499
6.2.6 Defining new generators and shrinkers . . . 499
6.2.7 Scoped generators and shrinkers . . . 502
6.2.8 Reproducing sequences of arbitrary terms . . . 503
6.2.9 Default size of generated terms . . . 504
6.2.10 Known issues . . . 504
6.3 assignvars . . . 504
6.3.1 API documentation . . . 504
6.3.2 Loading . . . 505
6.3.3 Testing . . . 505
6.4 base64 . . . 505
6.4.1 API documentation . . . 505
6.4.2 Loading . . . 505
6.4.3 Testing . . . 505
6.4.4 Encoding . . . 505
6.4.5 Decoding . . . 506
6.5 basic_types . . . 507
6.5.1 API documentation . . . 507
6.5.2 Loading . . . 507
6.5.3 Testing . . . 507
6.6 cbor . . . 507
6.6.1 Representation . . . 507
6.6.2 Encoding . . . 508
6.6.3 Decoding . . . 508
6.6.4 API documentation . . . 509
6.6.5 Loading . . . 509
6.6.6 Testing . . . 509
6.7 core . . . 509
6.7.1 API documentation . . . 509
6.7.2 Loading . . . 509
6.7.3 Testing . . . 509
6.8 coroutining . . . 509
6.8.1 API documentation . . . 510
6.8.2 Loading . . . 510
6.8.3 Testing . . . 510
6.8.4 Usage . . . 510
6.9 csv . . . 510
6.9.1 API documentation . . . 511
6.9.2 Loading . . . 511
6.9.3 Testing . . . 511
6.9.4 Usage . . . 511
6.10 dates . . . 512
6.10.1 API documentation . . . 513
6.10.2 Loading . . . 513

6.11 dependents . . . 513
6.11.1 API documentation . . . 513
6.11.2 Loading . . . 513
6.12 dictionaries . . . 513
6.12.1 API documentation . . . 513
6.12.2 Loading . . . 513
6.12.3 Testing . . . 514
6.12.4 Usage . . . 514
6.12.5 Credits . . . 515
6.13 dif . . . 515
6.13.1 API documentation . . . 515
6.13.2 Loading . . . 515
6.13.3 Testing . . . 515
6.13.4 Usage . . . 515
6.14 edcg . . . 516
6.14.1 API documentation . . . 516
6.14.2 Loading . . . 516
6.14.3 Testing . . . 516
6.14.4 Usage . . . 517
6.14.5 Introduction . . . 517
6.14.6 Syntax . . . 518
6.14.7 Declaration of Predicates . . . 518
6.14.8 Declaration of Accumulators . . . 518
6.14.9 Declaration of Passed Arguments . . . 519
6.14.10 Additional documentation . . . 519
6.15 events . . . 519
6.15.1 API documentation . . . 520
6.15.2 Loading . . . 520
6.16 expand_library_alias_paths . . . 520
6.16.1 API documentation . . . 520
6.16.2 Loading . . . 520
6.16.3 Usage . . . 520
6.17 expecteds . . . 520
6.17.1 API documentation . . . 521
6.17.2 Loading . . . 521
6.17.3 Testing . . . 521
6.17.4 Usage . . . 521
6.17.5 See also . . . 521
6.18 format . . . 522
6.18.1 API documentation . . . 522
6.18.2 Loading . . . 522
6.18.3 Testing . . . 522
6.18.4 Usage . . . 522
6.18.5 Portability . . . 523
6.19 genint . . . 523
6.19.1 API documentation . . . 523
6.19.2 Loading . . . 523
6.19.3 Testing . . . 523
6.19.4 Usage . . . 523
6.20 gensym . . . 524
6.20.1 API documentation . . . 524
6.20.2 Loading . . . 524
6.20.3 Testing . . . 524
6.20.4 Usage . . . 524

6.21 git . . . 524
6.21.1 API documentation . . . 525
6.21.2 Loading . . . 525
6.21.3 Testing . . . 525
6.21.4 Usage . . . 525
6.22 grammars . . . 526
6.22.1 API documentation . . . 526
6.22.2 Loading . . . 526
6.22.3 Testing . . . 526
6.22.4 Usage . . . 526
6.23 heaps . . . 527
6.23.1 API documentation . . . 527
6.23.2 Loading . . . 527
6.23.3 Testing . . . 527
6.23.4 Credits . . . 527
6.24 hierarchies . . . 527
6.24.1 API documentation . . . 527
6.24.2 Loading . . . 527
6.24.3 Testing . . . 528
6.25 hook_flows . . . 528
6.25.1 API documentation . . . 528
6.25.2 Loading . . . 528
6.25.3 Testing . . . 528
6.25.4 Usage . . . 528
6.26 hook_objects . . . 529
6.26.1 API documentation . . . 529
6.26.2 Loading . . . 529
6.26.3 Testing . . . 529
6.26.4 Usage . . . 529
6.27 html . . . 533
6.27.1 API documentation . . . 533
6.27.2 Loading . . . 533
6.27.3 Testing . . . 534
6.27.4 Generating a HTML document . . . 534
6.27.5 Generating a HTML fragment . . . 534
6.27.6 Working with callbacks to generate content . . . 535
6.27.7 Working with custom elements . . . 535
6.28 ids . . . 535
6.28.1 API documentation . . . 536
6.28.2 Loading . . . 536
6.28.3 Testing . . . 536
6.28.4 Usage . . . 536
6.29 intervals . . . 537
6.29.1 API documentation . . . 537
6.29.2 Loading . . . 537
6.29.3 Testing . . . 537
6.30 java . . . 537
6.30.1 API documentation . . . 537
6.30.2 Loading . . . 538
6.30.3 Testing . . . 538
6.30.4 Usage . . . 538
6.30.5 Known issues . . . 538
6.31 json . . . 538
6.31.1 API documentation . . . 539

6.31.2 Loading . . . 539
6.31.3 Testing . . . 539
6.31.4 Representation . . . 539
6.31.5 Encoding . . . 541
6.31.6 Decoding . . . 542
6.31.7 Known issues . . . 542
6.32 json_lines . . . 542
6.32.1 API documentation . . . 542
6.32.2 Loading . . . 542
6.32.3 Testing . . . 542
6.32.4 Representation . . . 543
6.32.5 Encoding . . . 545
6.32.6 Decoding . . . 546
6.32.7 Known issues . . . 546
6.33 listing . . . 546
6.33.1 API documentation . . . 546
6.33.2 Loading . . . 546
6.33.3 Testing . . . 546
6.33.4 Usage . . . 546
6.34 logging . . . 548
6.34.1 API documentation . . . 548
6.34.2 Loading . . . 548
6.35 loops . . . 548
6.35.1 API documentation . . . 548
6.35.2 Loading . . . 548
6.35.3 Testing . . . 548
6.35.4 Usage . . . 548
6.36 meta . . . 549
6.36.1 API documentation . . . 549
6.36.2 Loading . . . 549
6.36.3 Testing . . . 549
6.36.4 Usage . . . 549
6.37 meta_compiler . . . 549
6.37.1 API documentation . . . 549
6.37.2 Loading . . . 549
6.37.3 Testing . . . 550
6.37.4 Usage . . . 550
6.37.5 Known issues . . . 550
6.38 mutations . . . 550
6.38.1 API documentation . . . 551
6.38.2 Loading . . . 551
6.38.3 Testing . . . 551
6.38.4 Usage . . . 551
6.39 nested_dictionaries . . . 552
6.39.1 API documentation . . . 552
6.39.2 Loading . . . 552
6.39.3 Testing . . . 552
6.39.4 Usage . . . 552
6.39.5 Curly term representation . . . 553
6.40 optionals . . . 553
6.40.1 API documentation . . . 553
6.40.2 Loading . . . 553
6.40.3 Testing . . . 554
6.40.4 Usage . . . 554

6.40.5 See also . . . 554
6.41 options . . . 554
6.41.1 API documentation . . . 554
6.41.2 Loading . . . 555
6.41.3 Testing . . . 555
6.41.4 Usage . . . 555
6.42 os . . . 556
6.42.1 API documentation . . . 556
6.42.2 Loading . . . 556
6.42.3 Testing . . . 556
6.42.4 Known issues . . . 556
6.43 queues . . . 557
6.43.1 API documentation . . . 557
6.43.2 Loading . . . 557
6.43.3 Testing . . . 557
6.43.4 Usage . . . 557
6.44 random . . . 558
6.44.1 API documentation . . . 558
6.44.2 Loading . . . 558
6.44.3 Testing . . . 558
6.44.4 Usage . . . 558
6.45 reader . . . 559
6.45.1 API documentation . . . 559
6.45.2 Loading . . . 559
6.45.3 Testing . . . 559
6.46 recorded_database . . . 559
6.46.1 API documentation . . . 560
6.46.2 Loading . . . 560
6.46.3 Testing . . . 560
6.46.4 Usage . . . 560
6.46.5 Known issues . . . 560
6.47 redis . . . 560
6.47.1 API documentation . . . 561
6.47.2 Loading . . . 561
6.47.3 Testing . . . 561
6.47.4 Credits . . . 561
6.47.5 Known issues . . . 561
6.48 sets . . . 561
6.48.1 API documentation . . . 562
6.48.2 Loading . . . 562
6.48.3 Testing . . . 562
6.48.4 Usage . . . 562
6.48.5 Credits . . . 563
6.49 statistics . . . 564
6.49.1 API documentation . . . 564
6.49.2 Loading . . . 564
6.49.3 Testing . . . 564
6.50 term_io . . . 564
6.50.1 API documentation . . . 564
6.50.2 Loading . . . 564
6.50.3 Testing . . . 565
6.51 timeout . . . 565
6.51.1 API documentation . . . 565
6.51.2 Loading . . . 565

6.51.3 Testing . . . 565
6.51.4 Known issues . . . 565
6.52 tsv . . . 565
6.52.1 API documentation . . . 566
6.52.2 Loading . . . 566
6.52.3 Testing . . . 566
6.52.4 Usage . . . 566
6.53 types . . . 567
6.53.1 API documentation . . . 567
6.53.2 Loading . . . 567
6.53.3 Testing . . . 567
6.53.4 Type-checking . . . 567
6.53.5 Defining new types . . . 568
6.53.6 Examples . . . 568
6.54 ulid . . . 568
6.54.1 API documentation . . . 569
6.54.2 Loading . . . 569
6.54.3 Testing . . . 569
6.54.4 Generating ULIDs . . . 569
6.54.5 Type-checking ULIDs . . . 570
6.55 unicode_data . . . 570
6.55.1 Authors . . . 570
6.55.2 License . . . 570
6.55.3 Website . . . 571
6.55.4 Description . . . 571
6.55.5 Requirements . . . 571
6.55.6 Usage . . . 571
6.55.7 Known issues . . . 571
6.55.8 Acknowledgements . . . 572
6.55.9 Files and API Summary . . . 572
6.56 union_find . . . 577
6.56.1 API documentation . . . 577
6.56.2 Loading . . . 577
6.56.3 Testing . . . 578
6.56.4 Usage . . . 578
6.57 uuid . . . 579
6.57.1 API documentation . . . 579
6.57.2 Loading . . . 579
6.57.3 Testing . . . 579
6.57.4 Generating version 1 UUIDs . . . 579
6.57.5 Generating version 4 UUIDs . . . 580
6.57.6 Generating the null UUID . . . 580
6.58 zippers . . . 580
6.58.1 API documentation . . . 580
6.58.2 Loading . . . 581
6.58.3 Testing . . . 581

7 Ports **583**
7.1 fcube . . . 583
7.1.1 API documentation . . . 584
7.1.2 Loading . . . 584
7.1.3 Testing . . . 584
7.2 metagol . . . 584
7.2.1 API documentation . . . 585

7.2.2 Loading . . . 585
7.2.3 Testing . . . 585
7.3 toychr . . . 585
7.3.1 API documentation . . . 586
7.3.2 Loading . . . 586
7.3.3 Testing . . . 586

8 Contributions **587**
8.1 flags . . . 587
8.1.1 API documentation . . . 587
8.1.2 Loading . . . 587
8.1.3 Testing . . . 587
8.2 iso8601 . . . 588
8.2.1 API documentation . . . 588
8.2.2 Loading . . . 588
8.2.3 Testing . . . 588
8.3 pddl_parser . . . 588
8.3.1 API documentation . . . 589
8.3.2 Loading . . . 589
8.3.3 Testing . . . 589
8.4 verdi_neruda . . . 589
8.5 xml_parser . . . 592
8.5.1 API documentation . . . 592
8.5.2 Loading . . . 592
8.5.3 Testing . . . 592
8.5.4 Known issues . . . 592

9 Glossary **593**

Bibliography **603**

Index **607**

CHAPTER

ONE

USER MANUAL

1.1 Declarative object-oriented programming

Logtalk is a *declarative object-oriented logic programming language*. This means that Logtalk shares key concepts with other object-oriented programming languages but abstracts and reinterprets these concepts in the context of declarative logic programming.

The key concepts in *declarative* object-oriented programming are *encapsulation* and *reuse patterns*. Notably, the concept of *mutable state*, which is an *imperative* concept, is not a significant concept in *declarative* object-oriented programming. Declarative object-oriented programming concepts can be materialized in both logic and functional languages. In this section, we focus only on declarative object-oriented logic programming.

The first key generalization of object-oriented programming concepts is the concept of *object* itself. What an object encapsulates depends on the *base programming paradigm* where we apply object-oriented programming concepts. When these concepts are applied to an imperative language, where mutable state and destructive assignment are central, objects naturally encapsulate and abstract mutable state, providing disciplined access and modification. When these concepts are applied to a declarative logic language such as Prolog, objects naturally encapsulate predicates. Therefore, an object can be seen as a *theory*, expressed by a set of related predicates. Theories are usually static, and thus Logtalk objects are static by default. This contrasts with imperative object-oriented languages where usually classes are static and objects are dynamic. This view of an object as a set of predicates also forgoes a distinction between *data* and *procedures* that is central to imperative object-oriented languages but moot in declarative, *homoiconic* logic languages.

The second key generalization concerns the relation between objects and other entities such as protocols (interfaces) and ancestor objects. The idea is that entity relations define *reuse patterns* and the *roles* played by the participating entities. A common reuse pattern is *inheritance*. In this case, an entity inherits, and thus reuses, resources from an ancestor entity. In a reuse pattern, each participating entity plays a specific *role*. The same entity, however, can play multiple roles depending on its relations with other entities. For example, an object can play the role of a class for its instances, the role of a subclass for its superclasses, and the role of an instance for its metaclass. Another common reuse pattern is *protocol implementation*. In this case, an object implementing a protocol reuses its predicate declarations by providing an implementation for those predicates and exposing those predicates to its clients. An essential consequence of this generalization is that protocols, objects, and categories are first-class entities, while e.g. *prototype*, *parent*, *class*, *instance*, *metaclass*, *subclass*, *superclass*, or *ancestor* are just *roles* that an object can play. When sending a message to an object, the corresponding predicate *declaration* and predicate *definition* lookup procedures (reuse patterns) depend on the role or roles that the object plays (see the *Inheritance* section for details). Another consequence of this generalization is that a language can provide multiple reuse patterns instead of selecting a set of patterns and supporting this set as a design choice that excludes other reuse patterns. For example, most imperative object-oriented languages are either class-based or prototype-based. In contrast, Logtalk supports both classes and prototypes by providing the corresponding reuse patterns using objects as first-class entities capable of playing multiple roles.

1.2 Main features

Several years ago, I decided that the best way to learn object-oriented programming was to build my own object-oriented language. Prolog being always my favorite language, I chose to extend it with object-oriented capabilities. Strong motivation also comes from my frustration with Prolog shortcomings for writing large applications. Eventually this work led to the Logtalk programming language as we know it today. The first system to use the name Logtalk appeared in February 1995. At that time, Logtalk was mainly an experiment in computational reflection with a rudimentary runtime and no compiler. Based on feedback by users and on the author's subsequent work, the name was retained and Logtalk was created as a full programming language focusing on using object-oriented concepts for code encapsulation and reuse. Development started in January 1998 with the first public alpha version released in July 1998. The first stable release (2.0) was published in February 1999. Development of the third generation of Logtalk started in 2012 with the first public alpha version in August 2012 and the first stable release (3.0.0) in January 2015.

Logtalk provides the following features:

1.2.1 Integration of logic and object-oriented programming

Logtalk tries to bring together the main advantages of these two programming paradigms. On one hand, the object orientation allows us to work with the same set of entities in the successive phases of application development, giving us a way of organizing and encapsulating the knowledge of each entity within a given domain. On the other hand, logic programming allows us to represent, in a declarative way, the knowledge we have of each entity. Together, these two advantages allow us to minimize the distance between an application and its problem domain, making the writing and maintenance of programming easier and more productive.

From a pragmatic perspective, Logtalk objects provide Prolog with the possibility of defining several namespaces, instead of the traditional Prolog single database, addressing some of the needs of large software projects.

1.2.2 Integration of event-driven and object-oriented programming

Event-driven programming enables the building of reactive systems, where computing which takes place at each moment is a result of the observation of occurring events. This integration complements object-oriented programming, in which each computing is initiated by the explicit sending of a message to an object. The user dynamically defines what events are to be observed and establishes monitors for these events. This is especially useful when representing relationships between objects that imply constraints in the state of participating objects [Rumbaugh87], [Rumbaugh88], [Fornarino_et_al_89], [Razek92]. Other common uses are reflective applications like code debugging or profiling [Maes87]. Predicates can be implicitly called when a spied event occurs, allowing programming solutions which minimize object coupling. In addition, events provide support for behavioral reflection and can be used to implement the concepts of *pointcut* and *advice* found in Aspect-Oriented Programming.

1.2.3 Support for component-based programming

Predicates can be encapsulated inside *categories* which can be imported by any object, without any code duplication and irrespective of object hierarchies. A category is a first-class encapsulation entity, at the same level as objects and protocols, which can be used as a component when building new objects. Thus, objects may be defined through composition of categories, which act as fine-grained units of code reuse. Categories may also extend existing objects. Categories can be used to implement *mixins* and *aspects*. Categories allow for code reuse between unrelated objects, independent of hierarchy relations, in the same vein as protocols allow for interface reuse.

1.2.4 Support for both prototype and class-based systems

Almost any (if not all) object-oriented languages available today are either class-based or prototype-based [Lieberman86], with a strong predominance of class-based languages. Logtalk provides support for both hierarchy types. That is, we can have both prototype and class hierarchies in the same application. Prototypes solve a problem of class-based systems where we sometimes have to define a class that will have only one instance in order to reuse a piece of code. Classes solve a dual problem in prototype based systems where it is not possible to encapsulate some code to be reused by other objects but not by the encapsulating object. Stand-alone objects, that is, objects that do not belong to any hierarchy, are a convenient solution to encapsulate code that will be reused by several unrelated objects.

1.2.5 Support for multiple object hierarchies

Languages like Smalltalk-80 [Goldberg83], Objective-C [Cox86] and Java [Joy_et_al_00] define a single hierarchy rooted in a class usually named `Object`. This makes it easy to ensure that all objects share a common behavior but also tends to result in lengthy hierarchies where it is difficult to express objects which represent exceptions to default behavior. In Logtalk we can have multiple, independent, object hierarchies. Some of them can be prototype-based while others can be class-based. Furthermore, stand-alone objects provide a simple way to encapsulate utility predicates that do not need or fit in an object hierarchy.

1.2.6 Separation between interface and implementation

This is an expected (should we say standard ?) feature of almost any modern programming language. Logtalk provides support for separating interface from implementation in a flexible way: predicate directives can be contained in an object, a category or a protocol (first-order entities in Logtalk) or can be spread in both objects, categories and protocols.

1.2.7 Private, protected and public inheritance

Logtalk supports private, protected and public inheritance in a similar way to C++ [Stroustrup86], enabling us to restrict the scope of inherited, imported or implemented predicates (by default inheritance is public).

1.2.8 Private, protected and public object predicates

Logtalk supports data hiding by implementing private, protected and public object predicates in a way similar to C++ [Stroustrup86]. Private predicates can only be called from the container object. Protected predicates can be called by the container object or by the container descendants. Public predicates can be called from any object.

1.2.9 Parametric objects

Object names can be compound terms (instead of atoms), providing a way to parameterize object predicates. Parametric objects are implemented in a similar way to `L&O` [McCabe92], `OL(P)` [Fromherz93] or `SICStus Objects` [SICStus95] (however, access to parameter values is done via a built-in method instead of making the parameters scope global over the whole object). Parametric objects allows us to treat any predicate clause as defining an *instantiation* of a parametric object. Thus, a parametric object allows us to encapsulate and associate any number of predicates with a compound term.

1.2.10 High level multi-threading programming support

High level multi-threading programming is available when running Logtalk with selected backend Prolog compilers, allowing objects to support both synchronous and asynchronous messages. Logtalk allows programmers to take advantage of modern multi-processor and multi-core computers without bothering with the details of creating and destroying threads, implement thread communication, or synchronizing threads.

1.2.11 Smooth learning curve

Logtalk has a smooth learning curve, by adopting standard Prolog syntax and by enabling an incremental learning and use of most of its features.

1.2.12 Compatibility with most Prolog systems and the ISO standard

The Logtalk system has been designed to be compatible with most Prolog compilers and, in particular, with the ISO Prolog standard [ISO95]. It runs in almost any computer system with a modern Prolog compiler.

1.2.13 Performance

The current Logtalk implementation works as a trans-compiler: Logtalk source files are first compiled to Prolog source files, which are then compiled by the chosen Prolog compiler. Therefore, Logtalk performance necessarily depends on the *backend Prolog compiler*. The Logtalk compiler preserves the programmers choices when writing efficient code that takes advantage of tail recursion and first-argument indexing.

As an object-oriented language, Logtalk can use both *static binding* and *dynamic binding* for matching messages and methods. Furthermore, Logtalk entities (objects, protocols, and categories) are independently compiled, allowing for a very flexible programming development. Entities can be edited, compiled, and loaded at runtime, without necessarily implying recompilation of all related entities.

When dynamic binding is used, the Logtalk runtime engine implements caching of *message lookups* (including messages to *self* and *super* calls), ensuring a performance level close to what could be achieved when using static binding.

For more detailed information on performance, see its dedicated *section*.

1.2.14 Logtalk scope

Logtalk, being a superset of Prolog, shares with it the same preferred areas of application but also extends them with those areas where object-oriented features provide an advantage compared to plain Prolog. Among these areas we have:

Logic and object-oriented programming teaching and researching
Logtalk smooth learning curve, combined with support for both prototype and class-based programming, protocols, components or aspects via category-based composition, and other advanced object-oriented features allow a smooth introduction to object-oriented programming to people with a background in Prolog programming. The distribution of Logtalk source code using an open-source license provides a framework for people to learn and then modify to try out new ideas on object-oriented programming research. In addition, the Logtalk distribution includes plenty of programming examples that can be used in the classroom for teaching logic and object-oriented programming concepts.

Structured knowledge representations and knowledge-based systems
Logtalk objects, coupled with event-driven programming features, enable easy implementation of frame-like systems and similar structured knowledge representations.

Blackboard systems, agent-based systems, and systems with complex object relationships
Logtalk support for event-driven programming can provide a basis for the dynamic and reactive nature of blackboard type applications.

Highly portable applications
Logtalk is compatible with most modern Prolog systems that support official and de facto standards. Used as a way to provide Prolog with namespaces, it avoids the porting problems of most Prolog module systems. Platform, operating system, or compiler specific code can be isolated from the rest of the code by encapsulating it in objects with well-defined interfaces.

Alternative to a Prolog module system
Logtalk can be used as an alternative to a Prolog compiler module system. Most Prolog applications that use modules can be converted into Logtalk applications, improving portability across Prolog systems and taking advantage of the stronger encapsulation and reuse framework provided by Logtalk object-oriented features.

Integration with other programming languages
Logtalk support for most key object-oriented features helps users integrating Prolog with object-oriented languages like C++, Java, or Smalltalk by facilitating a high-level mapping between the two languages.

1.3 Nomenclature

Depending on your logic programming and object-oriented programming background (or lack of it), you may find Logtalk nomenclature either familiar or at odds with the terms used in other languages. In addition, being a superset of Prolog, terms such as *predicate* and *method* are often used interchangeably. Logtalk inherits most of its nomenclature from Prolog and Smalltalk.

Note that the same terms can have different meanings in different languages. A good example is *class*. The support for meta-classes in e.g. Smalltalk translates to a concept of class that is different in key aspects from the concept of class in e.g. Java or C++. Other terms that can have different meanings are *delegation* and *forwarding*. There are also cases where the same concept is found under different names in some languages (e.g., *self* and *this*) but that can also mean different concepts in Logtalk and other languages. Always be aware of these differences and be cautious with assumptions carried from other programming languages.

In this section, we map nomenclatures from Prolog and popular OOP languages such as Smalltalk, C++, Java, and Python to the Logtalk nomenclature. The Logtalk distribution includes several examples of how to implement common concepts found in other languages, complementing the information in this section. This Handbook also features a *Prolog interoperability section* and an extensive *glossary* providing the exact meaning of the names commonly used in Logtalk programming.

1.3.1 Prolog nomenclature

Being a superset of Prolog, Logtalk inherits its nomenclature. But Logtalk also aims to fix several Prolog shortcomings, thus introducing new concepts and refining existing Prolog concepts. Logtalk object-oriented nature also introduces names and concepts that are not common when discussing logic programming semantics. We mention here the most relevant ones, notably those where semantics or common practice differ. Further details can be found elsewhere in this Handbook.

arbitrary goals as directives
: Although not ISO Prolog Core standard compliant, several Prolog systems accept using arbitrary goal as directives. This is not supported in Logtalk source files. Always use an *initialization/1* directive to wrap those goals. This ensures that any initialization goals, which often have side-effects, are only called if the source file is successfully compiled and loaded.

calling a predicate
: Sending a *message* to an object is similar to *calling a goal* with the difference that the actual predicate that is called is determined not just by the message *term* but also by the object receiving the message and possibly its ancestors. This is also different from calling a Prolog module predicate: a message may result e.g. in calling a predicate *inherited* by the object but calling a module predicate requires the predicate to exist in (or be reexported by) the module.

closed-world assumption semantics
: Logtalk provides clear closed-world assumption semantics: messages or calls for declared but undefined predicates fail. Messages or calls for unknown (i.e., not declared) predicates throw an error. Crucially, this semantics applies to both *static* and *dynamic* predicates. But in Prolog workarounds are required to have a static predicate being known by the runtime without it being also defined (so that calling it would fail instead of throwing a predicate existence error).

compiling and loading source files
: Logtalk provides its own built-in predicates for *compiling and loading* source files. It also provides convenient top-level interpreter *shorthands* for these and other frequent operations. In general, the traditional Prolog built-in predicates and top-level interpreter shorthands cannot be used to load Logtalk source files.

debugging
: In most (if not all) Prolog systems, debugging support is a built-in feature made available using a

set of built-in predicates like `trace/0` and `spy/1`. But in Logtalk the *default debugger* is a regular application, implemented using a public *reflection API*. This means that the debugger must be explicitly loaded (either automatically from a *settings file* at startup or from the top-level). It also means that the debugger can be easily extended or replaced by an alternative application.

directive operators
Some Prolog systems declare directive names as operators (e.g., `dynamic`, `multifile`, `...`). This is not required by the ISO Prolog Core standard. It's a practice that should be avoided as it makes code non-portable.

encapsulation
Logtalk enforces encapsulation of object predicates, generating a permission error when a predicate is not within the scope of the caller. In contrast, most Prolog module systems allow any module predicate to be called by using explicit qualification, even if not exported. Worse, some Prolog systems also allow defining clauses for a module predicate outside the module, without declaring the predicate as multifile, by simply writing clauses with explicit module-qualified heads.

entity loading
When using Prolog modules, `use_module/1-2` (or equivalent) directives both load the module files and declare that the (implicitly or explicitly) imported predicates can be used with implicit module qualification. But Logtalk separates entity (object, protocol, category, or module) predicate *usage* declarations (via *uses/1* and *uses/2* or its own *use_module/1* and *use_module/2* directives) from *loading* goals (using the *logtalk_load/1* and *logtalk_load/2* predicates), called using an explicit and disciplined approach from *loader files*.

flags scope
The *set_logtalk_flag/2* **directive** is always local to the entity or source file that contains it. Only calls to the *set_logtalk_flag/2* **predicate** set the global default value for a flag. This distinction is lacking in Prolog (where directives usually have a global scope) and Prolog modules (where some flags are local to modules in some systems and global in other systems).

meta-predicate call semantics
Logtalk provides consistent *meta-predicate* call semantics: meta-arguments are always called in the meta-predicate calling context. This contrasts with Prolog module meta-predicates where the semantics of implicitly qualified calls are different from explicitly qualified calls.

operators scope
Operators declared inside an entity (object, protocol, or category) are local to the entity. But operators defined in a source file but outside an entity are global for compatibility with existing Prolog code.

predicates scope
In plain Prolog, all predicates are visible. In a Prolog module, a predicate can be exported or local. In Logtalk, a predicate can be *public*, *protected*, *private*, or *local*.

predicate declaration
Logtalk provides a clear distinction between *declaring* a predicate and *defining* a predicate. This is a fundamental requirement for the concept of *protocol* (aka interface) in Logtalk: we must be able to *declare* a predicate without necessarily *defining* it. This clear distinction is missing in Prolog and Prolog modules. Notably, it's a compiler error for a module to try to export a predicate that it does not define.

predicate loading conflicts
Logtalk does not use predicate import/export semantics. Thus, there are never conflicts when loading entities (objects, protocols, or categories) that declare the same public predicates. But attempting to load two Prolog modules that export the same predicate results in a conflict, usually a compilation error (this is especially problematic when the `use_module/1` directive is used; e.g. adding a new exported predicate can break applications that use the module but not the new predicate).

1.3.2 Smalltalk nomenclature

The Logtalk name originates from a combination of the Prolog and Smalltalk names. Smalltalk had a significant influence on the design of Logtalk and thus inherits some of its ideas and nomenclature. The following list relates the most commonly used Smalltalk terms with their Logtalk counterparts.

abstract class
: Similar to Smalltalk, an abstract class is just a class not meant to be instantiated by not understanding a message to create instances.

assignment statement
: Logtalk, as a superset of Prolog, uses *logic variables* and *unification* and thus provides no equivalent to the Smalltalk assignment statement.

block
: Logtalk supports lambda expressions and meta-predicates, which can be used to provide similar functionality to Smalltalk blocks.

class
: In Logtalk, *class* is a just a *role* that an object can play. This is similar to Smalltalk, where classes are also objects.

class method
: Class methods in Logtalk are simply instance methods declared and defined in the class metaclass.

class variable
: Logtalk objects, which can play the roles of class and instance, encapsulate predicates, not state. Class variables, which in Smalltalk are really shared instance variables, can be emulated in a class by defining a predicate locally instead of defining it in the class instances.

inheritance
: While Smalltalk only supports single inheritance, Logtalk supports single inheritance, multiple inheritance, and multiple instantiation.

instance
: While in Smalltalk every object is an *instance* of some class, objects in Logtalk can play different roles, including the role of a prototype where the concepts of instance and class don't apply. Moreover, instances can be either created dynamically or defined statically.

instance method
: Instance methods in Logtalk are simply predicates declared and defined in a class and thus inherited by the class instances.

instance variable
: Logtalk being a *declarative* language, objects encapsulate a set of predicates instead of encapsulating *state*. But different objects may provide different definitions of the same predicates. Mutable internal state as in Smalltalk can be emulated by using dynamic predicates.

message
: Similar to Smalltalk, a *message* is a request for an operation, which is interpreted in Logtalk as a logic query, asking for the construction of a proof that something is true.

message selector
: Logtalk uses the predicate template (i.e., the predicate callable term with all its arguments unbound) as a message selector. The actual type of the message arguments is not considered. Like Smalltalk, Logtalk uses *single dispatch* on the message receiver.

metaclass
: Metaclasses are optional in Logtalk (except for a root class) and can be shared by several classes. When metaclasses are used, infinite regression is simply avoided by making a class an instance of itself.

method
Same as in Smalltalk, a *method* is the actual code (i.e., predicate definition) that is run to answer a message. Logtalk uses the words *method* and *predicate* interchangeably.

method categories
There is no support in Logtalk for partitioning the methods of an object into different categories. The Logtalk concept of *category* (a first-class entity) was, however, partially inspired by Smalltalk method categories.

object
Unlike Smalltalk, where *everything* is an object, Logtalk language constructs include both *terms* (as in Prolog representing e.g. numbers and structures) and three first-class entities: objects, protocols, and categories.

pool variables*
Logtalk, as a superset of Prolog, uses *predicates* with no distinction between *variables* and *methods*. Categories can be used to share a set of predicate definitions between any number of objects.

protocol
In Smalltalk, an object *protocol* is the set of messages it understands. The same concept applies in Logtalk. But Logtalk also supports protocols as first-class entities where a protocol can be implemented by multiple objects and an object can implement multiple protocols.

self
Logtalk uses the same definition of *self* found in Smalltalk: the object that received the message being processed. Note, however, that *self* is not a keyword in Logtalk but is implicit in the *(::)/1* message to *self* control construct.

subclass
Same definition in Logtalk.

super
As in Smalltalk, the idea of *super* is to allow calling an inherited predicate (that is usually being redefined). Note, however, that *super* is not a keyword in Logtalk, which provides instead a *(^ ^)/1* *super* call control construct.

superclass
Same definition in Logtalk. But while in Smalltalk a class can only have a single superclass, Logtalk support for multiple inheritance allows a class to have multiple superclasses.

1.3.3 C++ nomenclature

There are several C++ glossaries available on the Internet. The list that follows relates the most commonly used C++ terms with their Logtalk equivalents.

abstract class
Logtalk uses an *operational* definition of abstract class: any class that does not inherit a method for creating new instances can be considered an abstract class. Moreover, Logtalk supports *interfaces/protocols*, which are often a better way to provide the functionality of C++ abstract classes.

base class
Logtalk uses the term *superclass* with the same meaning.

data member
Logtalk uses *predicates* for representing both behavior and data.

constructor function
There are no special methods for creating new objects in Logtalk. Instead, Logtalk provides a built-in

predicate, *create_object/4*, which can be used as a building block to define more sophisticated object creation predicates.

derived class
Logtalk uses the term *subclass* with the same meaning.

destructor function
There are no special methods for deleting new objects in Logtalk. Instead, Logtalk provides a built-in predicate, *abolish_object/1*, which is often used to define more sophisticated object deletion predicates.

friend function
Not supported in Logtalk. Nevertheless, see the User Manual section on *meta-predicates*.

instance
In Logtalk, an instance can be either created dynamically at runtime or defined statically in a source file in the same way as classes.

member
Logtalk uses the term *predicate*.

member function
Logtalk uses *predicates* for representing both behavior and data.

namespace
Logtalk does not support multiple identifier namespaces. All Logtalk entity identifiers share the same namespace (Logtalk entities are objects, categories, and protocols).

nested class
Logtalk does not support nested classes.

static member
Logtalk does not support a `static` keyword. But the equivalent of static members can be declared in a class metaclass.

template
Logtalk supports *parametric objects*, which allows you to get the similar functionality of templates at runtime.

this
Logtalk uses the built-in context method *self/1* for retrieving the instance that received the message being processed. Logtalk also provides a *this/1* method but for returning the class containing the method being executed. Why the name clashes? Well, the notion of *self* was inherited from Smalltalk, which predates C++.

virtual member function
There is no `virtual` keyword in Logtalk. Any inherited or imported predicate can be redefined (either overridden or specialized). Logtalk can use *static binding* or *dynamic binding* for locating both method declarations and method definitions. Moreover, methods that are declared but not defined simply fail when called (as per the *closed-world assumption*).

1.3.4 Java nomenclature

There are several Java glossaries available on the Internet. The list that follows relates the most commonly used Java terms with their Logtalk equivalents.

abstract class
: Logtalk uses an *operational* definition of abstract class: any class that does not inherit a method for creating new instances is an abstract class. I.e. there is no `abstract` keyword in Logtalk.

abstract method
: In Logtalk, you may simply declare a method (*predicate*) in a class without defining it, leaving its definition to some descendant subclass.

assertion
: There is no `assertion` keyword in Logtalk. Assertions are supported using Logtalk compilation hooks and developer tools.

class
: Logtalk objects can play the role of classes, instances, or protocols (depending on their relations with other objects).

extends
: There is no `extends` keyword in Logtalk. Class inheritance is indicated using *specialization relations*. Moreover, the *extends relation* is used in Logtalk to indicate protocol, category, or prototype extension.

interface
: Logtalk uses the term *protocol* with a similar meaning. But note that Logtalk objects and categories declared as implementing a protocol are not required to provide definitions for the declared predicates (*closed-world assumption*).

callback method
: Logtalk supports *event-driven programming*, the most common usage context of callback methods. Callback methods can also be implemented using *meta-predicates*.

constructor
: There are no special methods for creating new objects in Logtalk. Instead, Logtalk provides a built-in predicate, *create_object/4*, which is often used to define more sophisticated object creation predicates.

final
: There is no `final` keyword in Logtalk. Predicates can always be redeclared and redefined in subclasses (and instances!).

inner class
: Inner classes are not supported in Logtalk.

instance
: In Logtalk, an instance can be either created dynamically at runtime or defined statically in a source file in the same way as classes.

method
: Logtalk uses the term *predicate* interchangeably with the term *method*.

method call
: Logtalk usually uses the expression *message-sending* for method calls, true to its Smalltalk heritage.

method signature
: Logtalk selects the method/predicate to execute in order to answer a method call based only on the method name and number of arguments. Logtalk (and Prolog) are not typed languages in the same sense as Java.

package
: There is no concept of packages in Logtalk. All Logtalk entities (objects, protocols, categories) share a single namespace. But Logtalk does support a concept of *library* that allows grouping of entities whose source files share a common path prefix.

reflection
: Logtalk features a *white box* API supporting *structural* reflection about *entity contents*, a *black box* API supporting *behavioral* reflection about *object protocols*, and an *events* API for reasoning about messages exchanged at runtime.

static
: There is no `static` keyword in Logtalk. See the entries below on *static method* and *static variable*.

static method
: Static methods may be implemented in Logtalk by using a *metaclass* for the class and defining the static methods in the metaclass. I.e. static methods are simply instance methods of the class metaclass.

static variable
: Static variables are *shared instance variables* and can simply be both declared and defined in a class. The built-in database methods can be used to implement destructive updates if necessary by accessing and updating a single clause of a dynamic predicate stored in the class.

super
: Instead of a `super` keyword, Logtalk provides a super operator and control construct, *(^ ^)/1*, for calling overridden methods.

synchronized
: Logtalk supports *multi-threading programming* in selected Prolog compilers, including a *synchronized/1* predicate directive. Logtalk allows you to synchronize a predicate or a set of predicates using per-predicate or per-predicate-set *mutexes*.

this
: Logtalk uses the built-in context method *self/1* for retrieving the instance that received the message being processed. Logtalk also provides a *this/1* method but for returning the class containing the method being executed. Why the name clashes? Well, the notion of *self* was inherited from Smalltalk, which predates C++.

1.3.5 Python nomenclature

The list that follows relates the commonly used Python concepts with their Logtalk equivalents.

abstract class
: Logtalk uses a different definition of abstract class: a class that does not inherit a method for creating new instances. Notably, the presence of *abstract methods* (i.e., predicates that are declared but not defined) does not make a class abstract.

abstract method
: Logtalk uses the term *predicate* interchangeably with *method*. Predicates can be declared without also being defined in an object (or category).

class
: Logtalk objects can play the role of classes, instances, or protocols (depending on their relations with other objects).

dictionary
: There is no native, built-in associative data type. But the library provides several implementations of a dictionary protocol.

function
The closest equivalent is a predicate defined in `user`, a pseudo-object for predicates not defined in regular objects, and thus callable from anywhere without requiring a scope directive.

function object
Predicate calls (goals) can be passed or returned from other predicates and unified with other terms (e.g., variables).

import path
Logtalk uses the term *library* to refer to a directory of source files and supports defining aliases (symbolic names) to library paths to abstract the actual locations.

lambda
Logtalk natively supports lambda expressions.

list
Lists are compound terms with native syntax support.

list comprehensions
There is no native, built-in support for list comprehensions. But the standard `findall/3` predicate can be used to construct a list by calling a goal that generates the list elements.

loader
Logtalk uses the term *loader* to refer to source files whose main or sole purpose is to load other source files.

loop
There are no native loop control constructs based on a counter. But the library provides implementations of several loop predicates.

metaclass
Logtalk objects play the role of metaclasses when instantiated by objects that play the role of classes.

method
Logtalk uses the terms *method* and *predicate* interchangeably. Predicates can be defined in objects (and categories). The value of *self* is implicit, unlike in Python where it is the first parameter of any method.

method resolution order
Logtalk uses a depth-first algorithm to lookup method (predicate) declarations and definitions. It's possible to use predicate *aliases* to access predicate declarations and definitions other than the first ones found by the lookup algorithm.

object
Objects are first-class entities that can play multiple roles, including prototype, class, instance, and metaclass.

package
Logtalk uses the term *library* to refer to a directory of source files defining objects, categories, and protocols.

set
There is no native, built-in set type. But the library provides set implementations.

string
The interpretation of text between double-quotes depends on the `double_quotes` flag. Depending on this flag, double-quoted text can be interpreted as a list of characters, a list of character codes, or an atom. Some backend Prolog compilers allow double-quoted text to be interpreted as a string in the Python sense.

tuple
Compound terms can be used to represent tuples of any complexity.

variable
: Logtalk works with *logical variables*, which are close to the mathematical concept of variables and distinct from variables in imperative or imperative-based OOP languages where they are symbolic names for memory locations. Logical variables can be *unified* with any term, including other variables.

while loop
: The built-in `forall/2` predicate implements a *generate-and-test* loop.

1.4 Messages

Messages allow us to ask an object to prove a goal and must always match a declared predicate within the scope of the *sender* object. Note that sending a message is fundamentally different from calling a predicate. When calling a predicate, the caller decides implicitly which predicate definition will be executed. When sending a message, it is the receiving object, not the sender, that decides which predicate definition (if any) will be called to answer the message. The predicate definition that is used to answer a message depends on the relations between the object and its imported categories and ancestor objects (if any). See the *Inheritance* section for details on the predicate declaration and predicate definition lookup procedures.

When a message corresponds to a *meta-predicate*, the meta-arguments are always called in the context of the object (or category) sending the message.

Logtalk uses nomenclature similar to other object-oriented programming languages such as Smalltalk. Therefore, the terms *query* and *message* are used interchangeably when referring to a declared predicate that is part of an object interface. Likewise, the terms *predicate* and *method* are used interchangeably when referring to the predicate definition (inside an object or category) that is called to answer a message.

1.4.1 Operators used in message-sending

Logtalk declares the following operators for the message-sending control constructs:

```
:- op(600, xfy, ::).
:- op(600,  fy, ::).
:- op(600,  fy, ^^).
```

It is assumed that these operators remain active (once the Logtalk compiler and runtime files are loaded) until the end of the Prolog session (this is the usual behavior of most Prolog compilers). Note that these operator definitions are compatible with the predefined operators in the Prolog ISO standard.

1.4.2 Sending a message to an object

Sending a message to an object is accomplished by using the *(::)/2* control construct:

```
..., Object::Message, ...
```

The message must match a public predicate declared for the receiving object. The message may also correspond to a protected or private predicate if the *sender* matches the predicate scope container. If the predicate is declared but not defined, the message simply fails (as per the *closed-world assumption*).

1.4.3 Delegating a message to an object

It is also possible to send a message to an object while preserving the original *sender* and *meta-call context* by using the *[]/1* delegation control construct:

```
..., [Object::Message], ....
```

This control construct can only be used within objects and categories (at the top-level interpreter, the *sender* is always the pseudo-object `user` so using this control construct would be equivalent to using the `(::)/2` message-sending control construct).

1.4.4 Sending a message to *self*

While defining a predicate, we sometimes need to send a message to *self*, i.e., to the same object that has received the original message. This is done in Logtalk through the *(::)/1* control construct:

```
..., ::Message, ....
```

The message must match either a public or protected predicate declared for the receiving object or a private predicate within the scope of the *sender* otherwise an error will be thrown. If the message is sent from inside a category or if we are using private inheritance, then the message may also match a private predicate. Again, if the predicate is declared but not defined, the message simply fails (as per the *closed-world assumption*).

1.4.5 Broadcasting

In the Logtalk context, *broadcasting* is interpreted as the sending of several messages to the same object. This can be achieved by using the message-sending control construct described above. However, for convenience, Logtalk implements an extended syntax for message-sending that may improve program readability in some cases. This extended syntax uses the `(,)/2`, `(;)/2`, and `(->)/2` control constructs (plus the `(*->)/2` soft-cut control construct when provided by the backend Prolog compiler). For example, if we wish to send several messages to the same object, we can write:

```
| ?- Object::(Message1, Message2, ...).
```

This is semantically equivalent to:

```
| ?- Object::Message1, Object::Message2, ... .
```

This extended syntax may also be used with the `(::)/1` message-sending control construct.

1.4.6 Calling imported and inherited predicates

When redefining a predicate, sometimes we need to call the inherited definition in the new code. This functionality, introduced by the Smalltalk language through the `super` primitive, is available in Logtalk using the *(^^)/1* control construct:

```
..., ^^Predicate, ....
```

Most of the time we will use this control construct by instantiating the pattern:

```
Predicate :-
    ...,            % do something
    ^^Predicate,    % call inherited definition
    ... .           % do something more
```

This control construct is generalized in Logtalk where it may be used to call any imported or inherited predicate definition. This control construct may be used within objects and categories. When combined with *static binding*, this control construct allows imported and inherited predicates to be called with the same performance as local predicates. As with the message-sending control constructs, the `(^^)/1` call simply fails when the predicate is declared but not defined (as per the *closed-world assumption*).

1.4.7 Message sending and event generation

Assuming the *events* flag is set to `allow` for the object (or category) sending a message using the *(::)/2* control construct, two events are generated, one before and one after the message execution. Messages that are sent using the *(::)/1* (message to *self*) control construct or the *(^ ^)/1* super mechanism described above do not generate any events. The rationale behind this distinction is that messages to *self* and *super* calls are only used internally in the definition of methods or to execute additional messages with the same target object (represented by *self*). In other words, events are only generated when using an object's public interface; they cannot be used to break object encapsulation.

If we need to generate events for a public message sent to *self*, then we just need to write something like:

```
Predicate :-
    ...,
    % get self reference
    self(Self),
    % send a message to self using (::)/2
    Self::Message,
    ... .
```

If we also need the sender of the message to be other than the object containing the predicate definition, we can write:

```
Predicate :-
    ...,
    % send a message to self using (::)/2
    % sender will be the pseudo-object user
    self(Self),
    {Self::Message},
    ... .
```

When events are not used, it is possible to turn off event generation globally or on a per-entity basis by using the `events` compiler flag to optimize message-sending performance (see the *Event-driven programming* section for more details).

1.4.8 Sending a message from a module

Messages can be sent to objects from within Prolog modules. Depending on the backend support for goal-expansion and on the *optimize* flag being turned on, the messages will use static binding when possible. This optimization requires the object to be compiled and loaded before the module. Note that the module can be `user`. This is usually the case when sending the message from the top-level interpreter. Thus, the same conditions apply in this case. Note that loading Prolog modules using Prolog directives or built-in predicates necessarily limits the range of possible optimizations for messages sent from the modules.

> **Warning**
>
> If you want to benchmark the performance of a message-sending goal at the top-level interpreter, be careful to check first if the goal is pre-compiled to use static binding; otherwise you will also be benchmarking the Logtalk compiler itself.

1.4.9 Message sending performance

For a detailed discussion on message-sending performance, see the *Performance* section.

1.5 Objects

The main goal of Logtalk objects is the encapsulation and reuse of predicates. Instead of a single database containing all your code, Logtalk objects provide separated namespaces or databases, allowing the partitioning of code into more manageable parts. Logtalk is a *declarative programming language* and does not aim to bring some sort of new dynamic state change concept to Logic Programming or Prolog.

Logtalk, defines two built-in objects, user and logtalk, which are described at the end of this section.

1.5.1 Objects, prototypes, classes, and instances

There are only three kinds of encapsulation entities in Logtalk: *objects*, *protocols*, and *categories*. Logtalk uses the term *object* in a broad sense. The terms *prototype*, *parent*, *class*, *subclass*, *superclass*, *metaclass*, and *instance* always designate an object. Different names are used to emphasize the *role* played by an object in a particular context. I.e. we use a term other than object when we want to make the relationship with other objects explicit. For example, an object with an *instantiation* relation with another object plays the role of an *instance*, while the instantiated object plays the role of a *class*; an object with a *specialization* relation with another object plays the role of a *subclass*, while the specialized object plays the role of a *superclass*; an object with an *extension* relation with another object plays the role of a *prototype*, the same for the extended object. A *stand-alone* object, i.e. an object with no relations with other objects, is always interpreted as a prototype. In Logtalk, entity relations essentially define *patterns* of code reuse. An entity is compiled according to the roles it plays.

Logtalk allows you to work from standalone objects to any kind of hierarchy, either class-based or prototype-based. You may use single or multiple inheritance, use or forgo metaclasses, implement reflective designs, use parametric objects, and take advantage of protocols and categories (think components).

Prototypes

Prototypes are either self-defined objects or objects defined as extensions to other prototypes with whom they share common properties. Prototypes are ideal for representing one-of-a-kind objects. Prototypes usually represent concrete objects in the application domain. When linking prototypes using *extension* relations, Logtalk uses the term *prototype hierarchies* although most authors prefer to use the term *hierarchy* only with class generalization/specialization relations. In the context of logic programming, prototypes are often the ideal replacement for modules.

Classes

Classes are used to represent abstractions of common properties of sets of objects. Classes often provide an ideal structuring solution when you want to express hierarchies of abstractions or work with many similar objects. Classes are used indirectly through *instantiation*. Contrary to most object-oriented programming languages, instances can be created both dynamically at runtime or defined in a source file like other objects. Using classes requires defining at least one *metaclass*, as explained below.

1.5.2 Defining a new object

We can define a new object in the same way we write Prolog code: by using a text editor. Logtalk source files may contain one or more objects, categories, or protocols. If you prefer to define each entity in its own source file, it is recommended that the file be named after the object. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter file. For instance, we may define an object named `vehicle` and save it in a `vehicle.lgt` source file, which will be compiled to a `vehicle_lgt.pl` Prolog file (depending on the *backend compiler*, the names of the intermediate Prolog files may include a directory hash and a process identifier to prevent file name clashes when embedding Logtalk applications or running parallel Logtalk processes).

Object names can be atoms or compound terms (when defining parametric objects, see below). Objects, categories, and protocols share the same name space: we cannot have an object with the same name as a protocol or a category.

Object code (directives and predicates) is textually encapsulated by using two Logtalk directives: *object/1-5* and *end_object/0*. The simplest object will be one that is self-contained, not depending on any other Logtalk entity:

```
:- object(Object).
    ...
:- end_object.
```

If an object implements one or more protocols then the opening directive will be:

```
:- object(Object,
    implements([Protocol1, Protocol2, ...])).
    ...
:- end_object.
```

An object can import one or more categories:

```
:- object(Object,
    imports([Category1, Category2, ...])).
```

(continues on next page)

(continued from previous page)

```
    ...
:- end_object.
```

If an object both implements protocols and imports categories, then we will write:

```
:- object(Object,
    implements([Protocol1, Protocol2, ...]),
    imports([Category1, Category2, ...])).
    ...
:- end_object.
```

In object-oriented programming, objects are usually organized in hierarchies that enable interface and code sharing by inheritance. In Logtalk, we can construct prototype-based hierarchies by writing:

```
:- object(Prototype,
    extends(Parent)).
    ...
:- end_object.
```

We can also have class-based hierarchies by defining instantiation and specialization relations between objects. To define an object as a class instance we will write:

```
:- object(Object,
    instantiates(Class)).
    ...
:- end_object.
```

A class may specialize another class, its superclass:

```
:- object(Class,
    specializes(Superclass)).
    ...
:- end_object.
```

If we are defining a reflexive system where every class is also an instance, we will probably be using the following pattern:

```
:- object(Class,
    instantiates(Metaclass),
    specializes(Superclass)).
    ...
:- end_object.
```

In short, an object can be a *stand-alone* object or be part of an object hierarchy. The hierarchy can be prototype-based (defined by extending other objects) or class-based (with instantiation and specialization relations). An object may also implement one or more protocols or import one or more categories.

A *stand-alone* object (i.e., an object with no extension, instantiation, or specialization relations with other objects) always plays the role of a prototype, that is, a self-describing object. If we want to use classes and instances, then we will need to specify at least one instantiation or specialization relation. The best way to do this is to define a set of objects that provide the basis of a reflective system [Cointe87], [Moura94]. For example:

```
% avoid the inevitable unknown entity warnings as in a
% reflective system there will always be references to
% an entity that will be defined after the reference

:- set_logtalk_flag(unknown_entities, silent).


% default root of the inheritance graph
% providing predicates common to all objects

:- object(object,
    instantiates(class)).
    ...
:- end_object.


% default metaclass for all classes providing
% predicates common to all instantiable classes

:- object(class,
    instantiates(class),
    specializes(abstract_class)).
    ...
:- end_object.


% default metaclass for all abstract classes
% providing predicates common to all classes

:- object(abstract_class,
    instantiates(class),
    specializes(object)).
    ...
:- end_object.
```

Note that with these instantiation and specialization relations, `object`, `class`, and `abstract_class` are, at the same time, classes and instances of some class. In addition, each object inherits its own predicates and the predicates of the other two objects without any inheritance loop.

When a full-blown reflective system solution is not needed, the above scheme can be simplified by making an object an instance of itself, i.e. by making a class its own metaclass. For example:

```
:- object(class,
    instantiates(class)).
    ...
:- end_object.
```

We can use, in the same application, both prototype and class-based hierarchies (and freely exchange messages between all objects). We cannot, however, mix the two types of hierarchies by, e.g., specializing an object that extends another object in this current Logtalk version.

Logtalk also supports public, protected, and private inheritance. See the *inheritance* section for details.

1.5.3 Parametric objects

Parametric objects have a compound term as identifier where all the arguments of the compound term are variables. These variables can be bound when sending a message or become bound when a message to the object succeeds, thus acting as *object parameters*. The object predicates can be coded to depend on those parameters, which are logical variables shared by all object predicates. When an object state is set at object creation and never changed, parameters provide a better solution than using the object's database via asserts. Parametric objects can also be used to associate a set of predicates to terms that share a common functor and arity.

Accessing object parameters

Object parameters can be accessed using *parameter variables* or built-in execution context methods. Parameter variables is the recommended solution to access object parameters. Although they introduce a concept of entity global variables, their unique syntax (`_ParameterName_`) avoids conflicts, makes them easily recognizable, and distinguishes them from other named anonymous variables. For example:

```
:- object(foo(_Bar_, _Baz_, ...)).

    ...
    bar(_Bar_).

    baz :-
        baz(_Baz_),
        ... .
```

Note that using parameter variables doesn't change the fact that entity parameters are logical variables. Parameter variables simplify code maintenance by allowing parameters to be added, reordered, or removed without having to specify or update parameter indexes.

Logtalk provides also a *parameter/2* built-in local method to access individual parameters:

```
:- object(foo(_Bar, _Baz, ...)).

    ...
    bar(Bar) :-
        parameter(1, Bar).

    baz :-
        parameter(2, Baz),
        baz(Baz),
        ... .
```

An alternative solution is to use the built-in local method *this/1*, which allows access to all parameters with a single call. For example:

```
:- object(foo(_Bar, _Baz, ...)).

    ...
    baz :-
        this(foo(_, Baz, ...)),
        baz(Baz),
        ... .
```

Both solutions are equally efficient as calls to the methods `this/1` and `parameter/2` are usually compiled inline into a clause head unification. The drawback of this second solution is that we must check all calls of `this/1` if we change the object name. Note that we can't use these method with the message-sending operators (*(::)/2*, *(::)/1*, or *(^^)/1*).

When storing a parametric object in its own source file, the convention is to name the file after the object, with the object arity appended. For instance, when defining an object named `sort(Type)`, we may save it in a `sort_1.lgt` text file. This way it is easy to avoid file name clashes when saving Logtalk entities that have the same functor but different arity.

Parametric object proxies

Compound terms with the same functor and with the same number of arguments as a parametric object identifier may act as *proxies* to a parametric object. Proxies may be stored on the database as Prolog facts and be used to represent different instantiations of a parametric object identifier. For example:

```
:- object(circle(_Id_, _Radius_, _Color_)).

    :- public(area/1).
    ...

:- end_object.

% parametric object proxies:
circle('#1', 1.23, blue).
circle('#2', 3.71, yellow).
circle('#3', 0.39, green).
circle('#4', 5.74, black).
circle('#5', 8.32, cyan).
```

Logtalk provides a convenient notation for accessing proxies represented as Prolog facts when sending a message:

```
..., {Proxy}::Message, ...
```

For example, using the `circle/3` parametric object above, we can compute a list with the areas of all circles using the following goal:

```
| ?- findall(Area, {circle(_, _, _)}::area(Area), Areas).

Areas = [4.75291, 43.2412, 0.477836, 103.508, 217.468].
```

In this context, the proxy argument is proved as a plain Prolog goal. If successful, the message is sent to the corresponding parametric object. Typically, the proof allows retrieval of parameter instantiations. This construct can either be used with a proxy argument that is sufficiently instantiated in order to unify with a single Prolog fact or with a proxy argument that unifies with several facts on backtracking.

1.5.4 Finding defined objects

We can find, by backtracking, all defined objects by calling the *current_object/1* built-in predicate with an unbound argument:

```
| ?- current_object(Object).
Object = logtalk ;
Object = user ;
...
```

This predicate can also be used to test if an object is defined by calling it with a valid object identifier (an atom or a compound term).

1.5.5 Creating a new object in runtime

An object can be dynamically created at runtime by using the *create_object/4* built-in predicate:

```
| ?- create_object(Object, Relations, Directives, Clauses).
```

The first argument should be either a variable or the name of the new object (a Prolog atom or compound term, which must not match any existing entity name). The remaining three arguments correspond to the relations described in the opening object directive and to the object code contents (directives and clauses).

For example, the call:

```
| ?- create_object(
        foo,
        [extends(bar)],
        [public(foo/1)],
        [foo(1), foo(2)]
    ).
```

is equivalent to compiling and loading the object:

```
:- object(foo,
    extends(bar)).

    :- dynamic.

    :- public(foo/1).
    foo(1).
    foo(2).

:- end_object.
```

If we need to create a lot of (dynamic) objects at runtime, then it is best to define a metaclass or a prototype with a predicate that will call this built-in predicate to make new objects. This predicate may provide automatic object name generation, name checking, and accept object initialization options.

1.5.6 Abolishing an existing object

Dynamic objects can be abolished using the *abolish_object/1* built-in predicate:

```
| ?- abolish_object(Object).
```

The argument must be an identifier of a defined dynamic object; otherwise an error will be thrown.

1.5.7 Object directives

Object directives are used to set initialization goals, define object properties, document an object dependencies on other Logtalk entities, and load the contents of files into an object.

Object initialization

We can define a goal to be executed as soon as an object is (compiled and) loaded to memory with the *initialization/1* directive:

```
:- initialization(Goal).
```

The argument can be any valid Logtalk goal. For example, a call to a local predicate:

```
:- object(foo).

    :- initialization(init).
    :- private(init/0).

    init :-
        ... .

    ...

:- end_object.
```

Or a message to another object:

```
:- object(assembler).

    :- initialization(control::start).
    ...

:- end_object.
```

Another common initialization goal is a message to *self* in order to call an inherited or imported predicate. For example, assuming that we have a `monitor` category defining a `reset/0` predicate, we could write:

```
:- object(profiler,
    imports(monitor)).

    :- initialization(::reset).
    ...

:- end_object.
```

Note, however, that descendant objects do not inherit initialization directives. In this context, *self* denotes the object that contains the directive. Also note that object initialization does not necessarily mean setting an object dynamic state.

Dynamic objects

Similar to Prolog predicates, an object can be either static or dynamic. An object created during the execution of a program is always dynamic. An object defined in a file can be either dynamic or static. Dynamic objects are declared by using the *dynamic/0* directive in the object source code:

```
:- dynamic.
```

The directive must precede any predicate directives or clauses. Please be aware that using dynamic code results in a performance hit when compared to static code. We should only use dynamic objects when these need to be abolished during program execution. In addition, note that we can declare and define dynamic predicates within a static object.

Object documentation

An object can be documented with arbitrary user-defined information by using the *info/1* entity directive. See the *Documenting* section for details.

Loading files into an object

The *include/1* directive can be used to load the contents of a file into an object. A typical usage scenario is to load a plain Prolog file into an object, thus providing a simple way to encapsulate its contents. For example, assume a `cities.pl` file defining facts for a `city/4` predicate. We could define a wrapper for this database by writing:

```
:- object(cities).

    :- public(city/4).

    :- include(dbs('cities.pl')).

:- end_object.
```

The `include/1` directive can also be used when creating an object dynamically. For example:

```
| ?- create_object(cities, [], [public(city/4), include(dbs('cities.pl'))], []).
```

Declaring object aliases

The *uses/1* directive can be used to declare object aliases. The typical uses of this directive include shortening long object names, working consistently with specific parameterizations of parametric objects, and simplifying experimenting with different object implementations of the same protocol when using explicit message-sending.

1.5.8 Object relationships

Logtalk provides six sets of built-in predicates that enable us to query the system about the relationships that an object has with other entities.

The *instantiates_class/2-3* built-in predicates can be used to query all instantiation relations:

```
| ?- instantiates_class(Instance, Class).
```

or, if we also want to know the instantiation scope:

```
| ?- instantiates_class(Instance, Class, Scope).
```

Specialization relations can be found by using the *specializes_class/2-3* built-in predicates:

```
| ?- specializes_class(Class, Superclass).
```

or, if we also want to know the specialization scope:

```
| ?- specializes_class(Class, Superclass, Scope).
```

For prototypes, we can query extension relations using the *extends_object/2-3* built-in predicates:

```
| ?- extends_object(Object, Parent).
```

or, if we also want to know the extension scope:

```
| ?- extends_object(Object, Parent, Scope).
```

In order to find which objects import which categories, we can use the *imports_category/2-3* built-in predicates:

```
| ?- imports_category(Object, Category).
```

or, if we also want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

To find which objects implements which protocols, we can use the *implements_protocol/2-3* and *conforms_to_protocol/2-3* built-in predicates:

```
| ?- implements_protocol(Object, Protocol, Scope).
```

or, if we also want to consider inherited protocols:

```
| ?- conforms_to_protocol(Object, Protocol, Scope).
```

Note that, if we use an unbound first argument, we will need to use the *current_object/1* built-in predicate to ensure that the entity returned is an object and not a category.

To find which objects are explicitly complemented by categories, we can use the *complements_object/2* built-in predicate:

```
| ?- complements_object(Category, Object).
```

Note that more than one category may explicitly complement a single object, and a single category can complement several objects.

1.5.9 Object properties

We can find the properties of defined objects by calling the built-in predicate *object_property/2*:

```
| ?- object_property(Object, Property).
```

The following object properties are supported:

static
The object is static

dynamic
The object is dynamic (and thus can be abolished in runtime by calling the *abolish_object/1* built-in predicate)

built_in
The object is a built-in object (and thus always available)

threaded
The object supports/makes multi-threading calls

file(Path)
Absolute path of the source file defining the object (if applicable)

file(Basename, Directory)
Basename and directory of the source file defining the object (if applicable); `Directory` always ends with a `/`

lines(BeginLine, EndLine)
Source file begin and end lines of the object definition (if applicable)

directive(BeginLine, EndLine)
Source file begin and end lines of the object opening directive (if applicable)

context_switching_calls
The object supports context-switching calls (i.e., can be used with the *(<<)/2* debugging control construct)

dynamic_declarations
The object supports dynamic declarations of predicates

events
Messages sent from the object generate events

source_data
Source data available for the object

complements(Permission)
The object supports complementing categories with the specified permission (`allow` or `restrict`)

complements
The object supports complementing categories

public(Resources)
List of public predicates and operators declared by the object

protected(Resources)
List of protected predicates and operators declared by the object

private(Resources)
List of private predicates and operators declared by the object

declares(Predicate, Properties)
List of *properties* for a predicate declared by the object

defines(Predicate, Properties)
List of *properties* for a predicate defined by the object

includes(Predicate, Entity, Properties)
List of *properties* for an object multifile predicate that are defined in the specified entity (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the first multifile predicate clause)

provides(Predicate, Entity, Properties)
List of *properties* for other entity multifile predicates that are defined in the object (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, and `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the first multifile predicate clause)

alias(Entity, Properties)
List of *properties* for an *entity alias* declared by the object (the properties include `object` in case of an object alias, `module` in case of a module alias, `for(Original)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the `uses/1` or `use_module/1` directive)

alias(Predicate, Properties)
List of *properties* for a *predicate alias* declared by the object (the properties include `predicate`, `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the alias directive)

calls(Call, Properties)
List of *properties* for predicate calls made by the object (`Call` is either a predicate indicator or a control construct such as `(::)/1-2` or `(^^)/1` with a predicate indicator as argument; note that `Call` may not be ground in case of a call to a control construct where its argument is only known at runtime; the properties include `caller(Caller)`, `alias(Alias)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, `line_count(Start)` with `Caller`, `Alias`, and `NonTerminal` being predicate indicators and `Start` being the begin line of the predicate clause or directive making the call)

updates(Predicate, Properties)
List of *properties* for dynamic predicate updates (and also access using the `clause/2` predicate) made by the object (`Predicate` is either a predicate indicator or a control construct such as `(::)/1-2` or `(:)/2` with a predicate indicator as argument; note that `Predicate` may not be ground in case of a control construct argument only known at runtime; the properties include `updater(Updater)`, `alias(Alias)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, and `line_count(Start)` with `Updater` being a (possibly multifile) predicate indicator, `Alias` and `NonTerminal` being predicate indicators, and `Start` being the begin line of the predicate clause or directive updating the predicate)

number_of_clauses(Number)
Total number of predicate clauses defined in the object at compilation time (includes both user-defined clauses and auxiliary clauses generated by the compiler or by the *expansion hooks* but does not include clauses for multifile predicates defined for other entities or clauses for the object own multifile predicates contributed by other entities)

number_of_rules(Number)
Total number of predicate rules defined in the object at compilation time (includes both user-defined rules and auxiliary rules generated by the compiler or by the *expansion hooks* but does not include rules for multifile predicates defined for other entities or rules for the object own multifile predicates contributed by other entities)

number_of_user_clauses(Number)
Total number of user-defined predicate clauses defined in the object at compilation time (does not include clauses for multifile predicates defined for other entities or clauses for the object own multifile predicates contributed by other entities)

number_of_user_rules(Number)
: Total number of user-defined predicate rules defined in the object at compilation time (does not include rules for multifile predicates defined for other entities or rules for the object own multifile predicates contributed by other entities)

debugging
: The object is compiled in debug mode

module
: The object resulted from the compilation of a Prolog module

When a predicate is called from an `initialization/1` directive, the argument of the `caller/1` property is `(:-)/1`.

Some properties such as line numbers are only available when the object is defined in a source file compiled with the *source_data* flag turned on. Moreover, line numbers are only supported in *backend Prolog compilers* that provide access to the start line of a read term. When such support is not available, the value `-1` is returned for the start and end lines.

The properties that return the number of clauses (rules) report the clauses (rules) *textually defined in the object* for both multifile and non-multifile predicates. Thus, these numbers exclude clauses (rules) for multifile predicates *contributed* by other entities.

1.5.10 Built-in objects

Logtalk defines some built-in objects that are always available for any application.

The built-in pseudo-object `user`

The built-in user pseudo-object virtually contains all user predicate definitions not encapsulated in a Logtalk entity (or a Prolog module for backends supporting a module system). These predicates are assumed to be implicitly declared public. Messages sent from this pseudo-object, which includes messages sent from the top-level interpreter, generate events when the default value of the *events* flag is set to `allow`. Defining complementing categories for this pseudo-object is not supported.

With some of the *backend Prolog compilers* that support a module system, it is possible to load (the) Logtalk (compiler/runtime) into a module other than the pseudo-module `user`. In this case, the Logtalk pseudo-object `user` virtually contains all user predicate definitions defined in the module where Logtalk was loaded.

The built-in object `logtalk`

The built-in logtalk object provides *message printing* predicates, *question asking* predicates, *debug and trace event* predicates, predicates for accessing the internal database of loaded files and their properties, and also a set of low-level utility predicates normally used when defining hook objects. Consult its API documentation for details.

1.5.11 Multi-threading applications

When writing multi-threading applications, user-defined predicates calling built-in predicates such as `create_object/4` and `abolish_object/1` may need to be declared synchronized in order to avoid race conditions.

1.6 Protocols

Protocols enable the separation between interface and implementation: several objects can implement the same protocol, and an object can implement several protocols. Protocols may contain only predicate declarations. In some languages the term *interface* is used with a similar meaning. Logtalk allows predicate declarations of any scope within protocols, contrary to some languages that only allow public declarations.

Logtalk defines three built-in protocols, monitoring, expanding, and forwarding, which are described at the end of this section.

1.6.1 Defining a new protocol

We can define a new protocol in the same way we write Prolog code: by using a text editor. Logtalk source files may contain one or more objects, categories, or protocols. If you prefer to define each entity in its own source file, it is recommended that the file be named after the protocol. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter file. For example, we may define a protocol named `listp` and save it in a `listp.lgt` source file that will be compiled to a `listp_lgt.pl` Prolog file (depending on the backend compiler, the names of the intermediate Prolog files may include a directory hash and a process identifier to prevent file name clashes when embedding Logtalk applications or running parallel Logtalk processes).

Protocol names must be atoms. Objects, categories, and protocols share the same namespace: we cannot have a protocol with the same name as an object or a category.

Protocol directives are textually encapsulated by using two Logtalk directives: *protocol/1-2* and *end_protocol/0*. The most simple protocol will be one that is self-contained, not depending on any other Logtalk entity:

```
:- protocol(Protocol).
    ...
:- end_protocol.
```

If a protocol extends one or more protocols, then the opening directive will be:

```
:- protocol(Protocol,
    extends([Protocol1, Protocol2, ...])).
    ...
:- end_protocol.
```

In order to maximize protocol reuse, all predicates specified in a protocol should relate to the same functionality. Therefore, the only recommended use of protocol extension is when you need both a minimal protocol and an extended version of the same protocol with additional, convenient predicates.

1.6.2 Finding defined protocols

We can find, by backtracking, all defined protocols by using the *current_protocol/1* built-in predicate with a unbound argument:

```
| ?- current_protocol(Protocol).
```

This predicate can also be used to test if a protocol is defined by calling it with a valid protocol identifier (an atom).

1.6.3 Creating a new protocol in runtime

We can create a new (dynamic) protocol at runtime by calling the Logtalk built-in predicate *create_protocol/3*:

```
| ?- create_protocol(Protocol, Relations, Directives).
```

The first argument should be either a variable or the name of the new protocol (a Prolog atom, which must not match an existing entity name). The remaining two arguments correspond to the relations described in the opening protocol directive and to the protocol directives.

For instance, the call:

```
| ?- create_protocol(ppp, [extends(qqq)], [public([foo/1, bar/1])]).
```

is equivalent to compiling and loading the protocol:

```
:- protocol(ppp,
    extends(qqq)).

    :- dynamic.

    :- public([foo/1, bar/1]).

:- end_protocol.
```

If we need to create a lot of (dynamic) protocols at runtime, then it is best to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

1.6.4 Abolishing an existing protocol

Dynamic protocols can be abolished using the *abolish_protocol/1* built-in predicate:

```
| ?- abolish_protocol(Protocol).
```

The argument must be an identifier of a defined dynamic protocol; otherwise an error will be thrown.

1.6.5 Protocol directives

Protocol directives are used to define protocol properties and documentation.

Dynamic protocols

As usually happens with Prolog code, a protocol can be either static or dynamic. A protocol created during the execution of a program is always dynamic. A protocol defined in a file can be either dynamic or static. Dynamic protocols are declared by using the *dynamic/0* directive in the protocol source code:

```
:- dynamic.
```

The directive must precede any predicate directives. Please be aware that using dynamic code results in a performance hit when compared to static code. We should only use dynamic protocols when these need to be abolished during program execution.

Protocol documentation

A protocol can be documented with arbitrary user-defined information by using the *info/1* entity directive. See the *Documenting* section for details.

Loading files into a protocol

The *include/1* directive can be used to load the contents of a file into a protocol. See the *Objects* section for an example of using this directive.

1.6.6 Protocol relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the relationships that a protocol has with other entities.

The *extends_protocol/2-3* built-in predicates return all pairs of protocols so that the first one extends the second:

```
| ?- extends_protocol(Protocol1, Protocol2).
```

or, if we also want to know the extension scope:

```
| ?- extends_protocol(Protocol1, Protocol2, Scope).
```

To find which objects or categories implement which protocols, we can call the *implements_protocol/2-3* built-in predicates:

```
| ?- implements_protocol(ObjectOrCategory, Protocol).
```

or, if we also want to know the implementation scope:

```
| ?- implements_protocol(ObjectOrCategory, Protocol, Scope).
```

Note that, if we use a non-instantiated variable for the first argument, we will need to use the *current_object/1* or *current_category/1* built-in predicates to identify the kind of entity returned.

1.6.7 Protocol properties

We can find the properties of defined protocols by calling the *protocol_property/2* built-in predicate:

```
| ?- protocol_property(Protocol, Property).
```

A protocol may have the property `static`, `dynamic`, or `built_in`. Dynamic protocols can be abolished in runtime by calling the *abolish_protocol/1* built-in predicate. Depending on the *backend Prolog compiler*, a protocol may have additional properties related to the source file where it is defined.

The following protocol properties are supported:

`static`
: The protocol is static

`dynamic`
: The protocol is dynamic (and thus can be abolished in runtime by calling the *abolish_category/1* built-in predicate)

`built_in`
: The protocol is a built-in protocol (and thus always available)

`source_data`
: Source data available for the protocol

`file(Path)`
: Absolute path of the source file defining the protocol (if applicable)

`file(Basename, Directory)`
: Basename and directory of the source file defining the protocol (if applicable); `Directory` always ends with a `/`

`lines(BeginLine, EndLine)`
: Source file begin and end lines of the protocol definition (if applicable)

`directive(BeginLine, EndLine)`
: Source file begin and end lines of the protocol opening directive (if applicable)

`public(Resources)`
: List of public predicates and operators declared by the protocol

`protected(Resources)`
: List of protected predicates and operators declared by the protocol

`private(Resources)`
: List of private predicates and operators declared by the protocol

`declares(Predicate, Properties)`
: List of *properties* for a predicate declared by the protocol

`alias(Predicate, Properties)`
: List of *properties* for a *predicate alias* declared by the protocol (the properties include `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, and `line_count(Line)` with `Line` being the begin line of the alias directive)

Some of the properties, such as line numbers, are only available when the protocol is defined in a source file compiled with the *source_data* flag turned on.

1.6.8 Implementing protocols

Any number of objects or categories can implement a protocol. The syntax is very simple:

```
:- object(Object,
    implements(Protocol)).
    ...
:- end_object.
```

or, in the case of a category:

```
:- category(Object,
    implements(Protocol)).
    ...
:- end_category.
```

To make all public predicates declared via an implemented protocol protected or to make all public and protected predicates private we prefix the protocol's name with the corresponding keyword. For instance:

```
:- object(Object,
    implements(private::Protocol)).
    ...
:- end_object.
```

or:

```
:- object(Object,
    implements(protected::Protocol)).
    ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
    implements(public::Protocol)).
    ...
:- end_object.
```

The same rules apply to protocols implemented by categories.

1.6.9 Built-in protocols

Logtalk defines a set of built-in protocols that are always available for any application.

The built-in protocol `expanding`

The built-in expanding protocol declares the *term_expansion/2* and *goal_expansion/2* predicates. See the description of the *hook* compiler flag for more details.

The built-in protocol `monitoring`

The built-in monitoring protocol declares the *before/3* and *after/3* public event handler predicates. See the *Event-driven programming* section for more details.

The built-in protocol `forwarding`

The built-in forwarding protocol declares the *forward/1* user-defined message forwarding handler, which is automatically called (if defined) by the runtime for any message that the receiving object does not understand. See also the *[]/1* control construct.

1.6.10 Multi-threading applications

When writing multi-threading applications, user-defined predicates calling built-in predicates such as `create_protocol/3` and `abolish_protocol/1` may need to be declared synchronized in order to avoid race conditions.

1.7 Categories

Categories are *fine-grained units of code reuse* and can be regarded as a dual concept of protocols. Categories provide a way to encapsulate a set of related predicate declarations and definitions that do not represent a complete object and that only make sense when composed with other predicates. Categories may also be used to break a complex object into functional units. A category can be imported by several objects (without code duplication), including objects participating in prototype or class-based hierarchies. This concept of categories shares some ideas with Smalltalk-80 functional categories [Goldberg83], Flavors mix-ins [Moon86] (without necessarily implying multi-inheritance), and Objective-C categories [Cox86]. Categories may also *complement* existing objects, thus providing a *hot patching* mechanism inspired by the Objective-C categories functionality.

Logtalk defines a built-in category, core_messages, which is described at the end of this section.

1.7.1 Defining a new category

We can define a new category in the same way we write Prolog code: by using a text editor. Logtalk source files may contain one or more objects, categories, or protocols. If you prefer to define each entity in its own source file, it is recommended that the file be named after the category. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. Intermediate Prolog source files (generated by the Logtalk compiler) have, by default, a `_lgt` suffix and a `.pl` extension. Again, this can be set to match the needs of a particular Prolog compiler in the corresponding adapter file. For example, we may define a category named `documenting` and save it in a `documenting.lgt` source file that will be compiled to a `documenting_lgt.pl` Prolog file (depending on the *backend compiler*, the names of the intermediate Prolog files may include a directory hash and a process identifier to prevent file name clashes when embedding Logtalk applications or running parallel Logtalk processes).

Category names can be atoms or compound terms (when defining parametric categories). Objects, categories, and protocols share the same name space: we cannot have a category with the same name as an object or a protocol.

Category code (directives and predicates) is textually encapsulated by using two Logtalk directives: *category/1-4* and *end_category/0*. The most simple category will be one that is self-contained, not depending on any other Logtalk entity:

```
:- category(Category).
    ...
:- end_category.
```

If a category implements one or more protocols, then the opening directive will be:

```
:- category(Category,
    implements([Protocol1, Protocol2, ...])).
    ...
:- end_category.
```

A category may be defined as a composition of other categories by writing:

```
:- category(Category,
    extends([Category1, Category2, ...])).
    ...
:- end_category.
```

This feature should only be used when extending a category without breaking its functional cohesion (for example, when a modified version of a category is needed for importing on several unrelated objects). The preferred way of composing several categories is by importing them into an object. When a category overrides a predicate defined in an extended category, the overridden definition can still be called by using the *(^^)/1* control construct.

Categories cannot inherit from objects. In addition, categories cannot define clauses for dynamic predicates. This restriction applies because a category can be imported by several objects and because we cannot use the database handling built-in methods with categories (messages can only be sent to objects). A consequence of this restriction is that a category cannot declare a predicate (or non-terminal) as both multifile and dynamic. However, categories may contain declarations for dynamic predicates, and they can contain predicates that handle dynamic predicates. For example:

```
:- category(attributes).

    :- public(attribute/2).
```

(continues on next page)

(continued from previous page)

```
    :- public(set_attribute/2).
    :- public(del_attribute/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

    attribute(Attribute, Value) :-
        % called in the context of "self"
        ::attribute_(Attribute, Value).

    set_attribute(Attribute, Value) :-
        % retract old clauses in "self"
        ::retractall(attribute_(Attribute, _)),
        % assert new clause in "self"
        ::assertz(attribute_(Attribute, Value)).

    del_attribute(Attribute, Value) :-
        % retract clause in "self"
        ::retract(attribute_(Attribute, Value)).

:- end_category.
```

Each object importing this category will have its own `attribute_/2` private and dynamic predicate. The predicates `attribute/2`, `set_attribute/2`, and `del_attribute/2` always access and modify the dynamic predicate contained in the object receiving the corresponding messages (i.e., *self*). But it's also possible to define predicates that handle dynamic predicates in the context of *this* instead of *self*. For example:

```
:- category(attributes).

    :- public(attribute/2).
    :- public(set_attribute/2).
    :- public(del_attribute/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

    attribute(Attribute, Value) :-
        % call in the context of "this"
        attribute_(Attribute, Value).

    set_attribute(Attribute, Value) :-
        % retract old clauses in "this"
        retractall(attribute_(Attribute, _)),
        % asserts clause in "this"
        assertz(attribute_(Attribute, Value)).

    del_attribute(Attribute, Value) :-
        % retract clause in "this"
        retract(attribute_(Attribute, Value)).

:- end_category.
```

When defining a category that declares and handles dynamic predicates, working in the context of *this* ties

those dynamic predicates to the object importing the category, while working in the context of *self* allows each object inheriting from the object that imports the category to have its own set of clauses for those dynamic predicates.

1.7.2 Hot patching

A category may also explicitly complement one or more existing objects, thus providing *hot patching* functionality inspired by Objective-C categories:

```
:- category(Category,
    complements([Object1, Object2, ....])).
    ...
:- end_category.
```

This allows us to add missing directives (e.g., to define *aliases* for complemented object predicates), replace broken predicate definitions, add new predicates, and add protocols and categories to existing objects without requiring access or modifications to their source code. Common scenarios are adding logging or debugging predicates to a set of objects. Complemented objects need to be compiled with the *complements* compiler flag set `allow` (to allow both patching and adding functionality) or `restrict` (to allow only adding new functionality). A complementing category takes preference over a previously loaded complementing category for the same object, thus allowing patching a previous patch if necessary.

When replacing a predicate definition, it is possible to call the overridden definition in the object from the new definition in the category by using the *(@)/1* control construct. This construct is only meaningful when used within categories and requires a compile-time bound goal argument, which is called in *this* (i.e., in the context of the complemented object or the object importing a category). As an example, consider the following object:

```
:- object(bird).

    :- set_logtalk_flag(complements, allow).

    :- public(make_sound/0).
    make_sound :-
        write('Chirp, chirp!'), nl.

:- end_object.
```

We can use the `(@)/1` control construct to wrap the original `make_sound/0` predicate definition by writing:

```
:- category(logging,
    complements(bird)).

    make_sound :-
        write('Started making sound...'), nl,
        @make_sound,
        write('... finished making sound.'), nl.

:- end_category.
```

After loading the object and the category, calling the `make_sound/0` predicate will result in the following output:

```
| ?- bird::make_sound.

Started making sound...
Chirp, chirp!
... finished making sound.
yes
```

Note that *super calls* from predicates defined in complementing categories lookup inherited definitions as if the calls were made from the complemented object instead of the category ancestors. This allows more comprehensive object patching. But it also means that, if you want to patch an object so that it imports a category that extends another category and uses super calls to access the extended category predicates, you will need to define a (possibly empty) complementing category that extends the category that you want to add.

An unfortunate consequence of allowing an object to be patched at runtime using a complementing category is that it disables the use of *static binding* optimizations for messages sent to the complemented object, as it can always be later patched, thus rendering the static binding optimizations invalid.

Another important caveat is that, while a complementing category can replace a predicate definition, local callers of the replaced predicate will still call the non-patched version of the predicate. This is a consequence of the lack of a portable solution at the *backend Prolog compiler* level for replacing static predicate definitions.

1.7.3 Finding defined categories

We can find, by backtracking, all defined categories by using the *current_category/1* built-in predicate with an unbound argument:

```
| ?- current_category(Category).
```

This predicate can also be used to test if a category is defined by calling it with a valid category identifier (an atom or a compound term).

1.7.4 Creating a new category in runtime

A category can be dynamically created at runtime by using the *create_category/4* built-in predicate:

```
| ?- create_category(Category, Relations, Directives, Clauses).
```

The first argument should be either a variable or the name of the new category (a Prolog atom, which must not match with an existing entity name). The remaining three arguments correspond to the relations described in the opening category directive and to the category code contents (directives and clauses).

For example, the call:

```
| ?- create_category(
        ccc,
        [implements(ppp)],
        [private(bar/1)],
        [(foo(X):-bar(X)), bar(1), bar(2)]
    ).
```

is equivalent to compiling and loading the category:

```
:- category(ccc,
    implements(ppp)).

    :- dynamic.

    :- private(bar/1).

    foo(X) :-
        bar(X).

    bar(1).
    bar(2).

:- end_category.
```

If we need to create a lot of (dynamic) categories at runtime, then it is best to define a metaclass or a prototype with a predicate that will call this built-in predicate in order to provide more sophisticated behavior.

1.7.5 Abolishing an existing category

Dynamic categories can be abolished using the *abolish_category/1* built-in predicate:

```
| ?- abolish_category(Category).
```

The argument must be an identifier of a defined dynamic category; otherwise, an error will be thrown.

1.7.6 Category directives

Category directives are used to define category properties, document category dependencies on other Logtalk entities, and load the contents of files into a category.

Dynamic categories

As usually happens with Prolog code, a category can be either static or dynamic. A category created during the execution of a program is always dynamic. A category defined in a file can be either dynamic or static. Dynamic categories are declared by using the *dynamic/0* directive in the category source code:

```
:- dynamic.
```

The directive must precede any predicate directives or clauses. Please be aware that using dynamic code results in a performance hit when compared to static code. We should only use dynamic categories when these need to be abolished during program execution.

Category documentation

A category can be documented with arbitrary user-defined information by using the *info/1* entity directive. See the *Documenting* section for details.

Loading files into a category

The *include/1* directive can be used to load the contents of a file into a category. See the *Objects* section for an example of using this directive.

Declaring object aliases

The *uses/1* directive can be used to declare object aliases. The typical uses of this directive are to shorten long object names and to simplify experimenting with different object implementations of the same protocol when using explicit message-sending.

1.7.7 Category relationships

Logtalk provides two sets of built-in predicates that enable us to query the system about the relationships that a category has with other entities.

The built-in predicates *implements_protocol/2-3* and *conforms_to_protocol/2-3* allow us to find which categories implement which protocols:

```
| ?- implements_protocol(Category, Protocol, Scope).
```

or, if we also want to consider inherited protocols:

```
| ?- conforms_to_protocol(Category, Protocol, Scope).
```

Note that, if we use an unbound first argument, we will need to use the *current_category/1* built-in predicate to ensure that the returned entity is a category and not an object.

To find which objects import which categories, we can use the *imports_category/2-3* built-in predicates:

```
| ?- imports_category(Object, Category).
```

or, if we also want to know the importation scope:

```
| ?- imports_category(Object, Category, Scope).
```

Note that a category may be imported by several objects.

To find which categories extend other categories, we can use the *extends_category/2-3* built-in predicates:

```
| ?- extends_category(Category1, Category2).
```

or, if we also want to know the extension scope:

```
| ?- extends_category(Category1, Category2, Scope).
```

Note that a category may be extended by several categories.

To find which categories explicitly complement existing objects we can use the *complements_object/2* built-in predicate:

```
| ?- complements_object(Category, Object).
```

Note that a category may explicitly complement several objects.

1.7.8 Category properties

We can find the properties of defined categories by calling the built-in predicate *category_property/2*:

```
| ?- category_property(Category, Property).
```

The following category properties are supported:

static
The category is static

dynamic
The category is dynamic (and thus can be abolished in runtime by calling the *abolish_category/1* built-in predicate)

built_in
The category is a built-in category (and thus always available)

file(Path)
Absolute path of the source file defining the category (if applicable)

file(Basename, Directory)
Basename and directory of the source file defining the category (if applicable); `Directory` always ends with a `/`

lines(BeginLine, EndLine)
Source file begin and end lines of the category definition (if applicable)

directive(BeginLine, EndLine)
Source file begin and end lines of the category opening directive (if applicable)

events
Messages sent from the category generate events

source_data
Source data available for the category

public(Resources)
List of public predicates and operators declared by the category

protected(Resources)
List of protected predicates and operators declared by the category

private(Resources)
List of private predicates and operators declared by the category

declares(Predicate, Properties)
List of *properties* for a predicate declared by the category

defines(Predicate, Properties)
List of *properties* for a predicate defined by the category

includes(Predicate, Entity, Properties)
List of *properties* for an object multifile predicate that are defined in the specified entity (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the first multifile predicate clause)

provides(Predicate, Entity, Properties)
List of *properties* for other entity multifile predicate that are defined in the category (the properties include `number_of_clauses(Number)`, `number_of_rules(Number)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the first multifile predicate clause)

alias(Entity, Properties)
List of *properties* for an *entity alias* declared by the object (the properties include `object` in case of an object alias, `module` in case of a module alias, `for(Original)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the `uses/1` or `use_module/1` directive)

alias(Predicate, Properties)
List of *properties* for a *predicate alias* declared by the category (the properties include `predicate`, `for(Original)`, `from(Entity)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, and `line_count(Start)` with `Start` being the begin line of the alias directive)

calls(Call, Properties)
List of *properties* for predicate calls made by the category (`Call` is either a predicate indicator or a control construct such as `(::)/1-2` or `(^^)/1` with a predicate indicator as argument; note that `Call` may not be ground in case of a call to a control construct where its argument is only know at runtime; the properties include `caller(Caller)`, `alias(Alias)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, and `line_count(Start)` with `Caller`, `Alias`, and `NonTerminal` being predicate indicators and `Start` being the begin line of the predicate clause or directive making the call)

updates(Predicate, Properties)
List of *properties* for dynamic predicate updates (and also access using the `clause/2` predicate) made by the object (`Predicate` is either a predicate indicator or a control construct such as `(::)/1-2` or `(:)/2` with a predicate indicator as argument; note that `Predicate` may not be ground in case of a control construct argument only know at runtime; the properties include `updater(Updater)`, `alias(Alias)`, `non_terminal(NonTerminal)`, `lines(Start,End)`, and `line_count(Start)` with `Updater` being a (possibly multifile) predicate indicator, `Alias` and `NonTerminal` being predicate indicators, and `Start` being the begin line of the predicate clause or directive updating the predicate)

number_of_clauses(Number)
Total number of predicate clauses defined in the category (includes both user-defined clauses and auxiliary clauses generated by the compiler or by the *expansion hooks* but does not include clauses for multifile predicates defined for other entities or clauses for the category own multifile predicates contributed by other entities)

number_of_rules(Number)
Total number of predicate rules defined in the category (includes both user-defined rules and auxiliary rules generated by the compiler or by the *expansion hooks* but does not include rules for multifile predicates defined for other entities or rules for the category own multifile predicates contributed by other entities)

number_of_user_clauses(Number)
Total number of user-defined predicate clauses defined in the category (does not include clauses for multifile predicates defined for other entities or clauses for the category own multifile predicates contributed by other entities)

number_of_user_rules(Number)
Total number of user-defined predicate rules defined in the category (does not include rules for multifile predicates defined for other entities or rules for the category own multifile predicates contributed by other entities)

Some properties, such as line numbers, are only available when the category is defined in a source file compiled with the *source_data* flag turned on. Moreover, line numbers are only supported in *backend Prolog compilers* that provide access to the start line of a read term. When such support is not available, the value `-1` is returned for the start and end lines.

The properties that return the number of clauses (rules) report the clauses (rules) *textually defined in the object* for both multifile and non-multifile predicates. Thus, these numbers exclude clauses (rules) for multifile predicates *contributed* by other entities.

1.7.9 Importing categories

Any number of objects can import a category. In addition, an object may import any number of categories. The syntax is very simple:

```
:- object(Object,
    imports([Category1, Category2, ...])).
    ...
:- end_object.
```

To make all public predicates imported via a category protected, or to make all public and protected predicates private, we prefix the category's name with the corresponding keyword:

```
:- object(Object,
    imports(private::Category)).
    ...
:- end_object.
```

or:

```
:- object(Object,
    imports(protected::Category)).
    ...
:- end_object.
```

Omitting the scope keyword is equivalent to writing:

```
:- object(Object,
    imports(public::Category)).
    ...
:- end_object.
```

1.7.10 Calling category predicates

Category predicates can be called from within an object by sending a message to *self* or using a *super* call. Consider the following category:

```
:- category(output).

    :- public(out/1).

    out(X) :-
        write(X), nl.

:- end_category.
```

The predicate `out/1` can be called from within an object importing the category by simply sending a message to *self*. For example:

```
:- object(worker,
    imports(output)).

    ...
    do(Task) :-
        execute(Task, Result),
        ::out(Result).
    ...

:- end_object.
```

This is the recommended way of calling a category predicate that can be specialized/overridden in a descendant object, as the predicate definition lookup will start from *self*.

A direct call to a predicate definition found in an imported category can be made using the *(^ ^)/1* control construct. For example:

```
:- object(worker,
    imports(output)).

    ...
    do(Task) :-
        execute(Task, Result),
        ^^out(Result).
    ...

:- end_object.
```

This alternative should only be used when the user knows a priori that the category predicates will not be specialized or redefined by descendant objects of the object importing the category. Its advantage is that, when the *optimize* flag is turned on, the Logtalk compiler will try to optimize the calls by using *static binding*. When *dynamic binding* is used due to e.g. the lack of sufficient information at compilation time, the performance is similar to calling the category predicate using a message to *self* (in both cases, a predicate lookup caching mechanism is used).

1.7.11 Parametric categories

Category predicates can be parameterized in the same way as object predicates by using a compound term as the category identifier where all the arguments of the compound term are variables. These variables, the *category parameters*, can be accessed by calling the *parameter/2* or *this/1* built-in local methods in the category predicate clauses or by using *parameter variables*.

Category parameter values can be defined by the importing objects. For example:

```
:- object(speech(Season, Event),
    imports([dress(Season), speech(Event)])).
    ...
:- end_object.
```

Note that access to category parameters is only possible from within the category. In particular, calls to the *this/1* built-in local method from category predicates always access the importing object identifier (and thus object parameters, not category parameters).

1.7.12 Built-in categories

Logtalk defines a built-in category that is always available for any application.

The built-in category `core_messages`

The built-in core_messages category provides default translations for all compiler and runtime printed messages, such as warnings and errors. It does not define any public predicates.

1.7.13 Multi-threading applications

When writing multi-threading applications, user-defined predicates calling built-in predicates such as `create_category/4` and `abolish_category/1` may need to be declared synchronized in order to avoid race conditions.

1.8 Predicates

Predicate directives and clauses can be encapsulated inside objects and categories. Protocols can only contain predicate directives. From the point of view of a traditional imperative object-oriented language, predicates allow both object state and object behavior to be represented. Mutable object state can be represented using dynamic object predicates but should only be used when strictly necessary, as it breaks declarative semantics.

1.8.1 Reserved predicate names

For practical and performance reasons, some predicate names have a fixed interpretation. These predicates are declared in the built-in protocols. They are: *goal_expansion/2* and *term_expansion/2*, declared in the expanding protocol; *before/3* and *after/3*, declared in the monitoring protocol; and *forward/1*, declared in the forwarding protocol. By default, the compiler prints a warning when a definition for one of these predicates is found but the reference to the corresponding built-in protocol is missing.

1.8.2 Declaring predicates

Logtalk provides a clear distinction between *declaring a predicate* and *defining a predicate* and thus clear *closed-world assumption* semantics. Messages or calls for declared but undefined predicates fail. Messages or calls for unknown (i.e., non-declared) predicates throw an error. Note that this is a fundamental requirement for supporting *protocols*: we must be able to declare a predicate without necessarily defining it.

All object (or category) predicates that we want to access from other objects (or categories) must be explicitly declared. A predicate declaration must contain, at least, a *scope* directive. Other directives may be used to document the predicate or to ensure proper compilation of the predicate clauses.

Scope directives

A predicate scope directive specifies *from where* the predicate can be called, i.e. its *visibility*. Predicates can be *public*, *protected*, *private*, or *local*. Public predicates can be called from any object. Protected predicates can only be called from the container object or from a container descendant. Private predicates can only be called from the container object. Predicates are local when they are not declared in a scope directive. Local predicates, like private predicates, can only be called from the container object (or category), but they are *invisible* to the reflection built-in methods (*current_predicate/1* and *predicate_property/2*) and to the message error handling mechanisms (i.e., sending a message corresponding to a local predicate results in a `predicate_declaration` existence error instead of a scope error).

The scope declarations are made using the directives *public/1*, *protected/1*, and *private/1*. For example:

```
:- public(init/1).

:- protected(valid_init_option/1).

:- private(process_init_options/1).
```

If a predicate does not have a (local or inherited) scope declaration, it is assumed that the predicate is local. Note that we do not need to write scope declarations for all defined predicates. One exception is local dynamic predicates: declaring them as private predicates may allow the Logtalk compiler to generate optimized code for asserting and retracting clauses.

Note that a predicate scope directive doesn't specify *where* a predicate is, or can be, defined. For example, a private predicate can only be called from an object holding its scope directive. But it can be defined in descendant objects. A typical example is an object playing the role of a class defining a private (possibly dynamic) predicate for its descendant instances. Only the class can call (and possibly assert/retract clauses for) the predicate, but its clauses can be found/defined in the instances themselves.

Scope directives may also be used to declare grammar rule non-terminals and operators. For example:

```
:- public(url//1).

:- public(op(800, fx, tag)).
```

Note that, in the case of operators, the operator definitions don't become global when the entity containing the directives is compiled and loaded. This prevents an application from breaking when, for example, an updated third-party library adds new operators. It also allows loading entities that provide conflicting operator definitions. Here the usual programming idiom is to copy the operator definitions to a `uses/2` directive. For example, the *lgtunit* tool makes available a `(=~=)/2` predicate (for approximate float equality) that is intended to be used as an infix operator:

```
:- uses(lgtunit, [
    op(700, xfx, =~=), (=~=)/2
]).
```

Thus, in practice, the solution to use library entity operators in client entities is the same for using library entity predicates with implicit message-sending.

Mode directive

Often predicates can only be called using specific argument patterns. The valid arguments and instantiation modes of those arguments can be documented using the *mode/2* directive. For example:

```
:- mode(member(?term, ?list), zero_or_more).
```

The first directive argument describes a valid *calling mode*. The minimum information will be the instantiation mode of each argument. The first four possible values are described in the ISO Prolog Core standard [ISO95] (but with the meaning of the - instantiation mode redefined here). The remaining two can also be found in use in some Prolog systems.

+
: Argument must be instantiated (but not necessarily ground).

-
: Argument should be a free (non-instantiated) variable. When bound, the call will unify the computed term with the given argument.

?
: Argument can either be instantiated or free.

@
: Argument will not be further instantiated (modified).

++
: Argument must be ground.

--
: Argument must be unbound. Used mainly when returning an opaque term (e.g., a stream handle).

Note that the `+` and `@` instantiation modes have the same meaning for atomic arguments. E.g. you can write either `+atom` or `@atom` but the first alternative is preferred.

The ISO - instantiation mode is equivalent to the Logtalk `--` mode, allowing the use - to document predicates with output arguments that don't require those arguments to be unbound at call time and also accept bound arguments without throwing an exception.

These six mode atoms are also declared as prefix operators by the Logtalk compiler. This makes it possible to include type information for each argument as in the example above. Some possible type values are: `event`, `object`, `category`, `protocol`, `callable`, `term`, `nonvar`, `var`, `atomic`, `atom`, `number`, `integer`, `float`, `compound`, and `list`. The first four are Logtalk specific. The remaining are common Prolog types. We can also use our own types that can be either atoms or ground compound terms. See the *types* library documentation for an extensive list of pre-defined types that cover most common use cases.

The second directive argument documents the *number of proofs*, but not necessarily distinct solutions, for the specified mode. As an example, the `member(X, [1,1,1,1])` goal has only one distinct solution but four proofs for that solution. Note that different modes for the same predicate often have different determinism. The possible values are:

zero
: Predicate always fails (e.g., the `false/0` standard predicate).

one
: Predicate always succeeds once (e.g., the `flush_output/0` standard predicate).

zero_or_one
: Predicate either fails or succeeds (e.g., the `atom/1` standard predicate).

zero_or_more
: Predicate has zero or more proofs (e.g., the `current_predicate/1` standard predicate).

one_or_more
Predicate has one or more proofs (e.g., the `repeat/0` standard predicate).

zero_or_error
Predicate either fails or throws an error.

one_or_error
Predicate either succeeds once or throws an error (e.g., the `open/3` standard predicate).

zero_or_one_or_error
Predicate succeeds once or fails or throws an error (e.g., the `get_char/1` standard predicate).

zero_or_more_or_error
Predicate may fail or succeed multiple times or throw an error (e.g., the `retract/1` standard predicate).

one_or_more_or_error
Predicate may succeed one or more times or throw an error.

error
Predicate will throw an error (e.g., the `type_error/2` built-in method).

The last six values support documenting that some call modes may throw an error or will throw an error **despite the call arguments complying with the expected types and instantiation modes**. As an example, consider the `open/3` standard predicate:

```
:- mode(open(@source_sink, @io_mode, --stream), one_or_error).
```

In this case, the mode directive tells the user that a valid call can still throw an error (there may be e.g. a permission error opening the specified source or sink).

Notice that using the `zero`, `one`, `zero_or_one`, `zero_or_more`, or `one_or_more` modes is not only for predicates that never throw an exception; they can also be used for any predicate that doesn't throw an exception when the arguments are valid. For example, the `current_predicate/1` standard predicate throws an exception if the argument is neither a variable nor a predicate indicator, but it succeeds zero or more times when its argument is valid:

```
:- mode(current_predicate(?predicate_indicator), zero_or_more).
```

Some predicates have more than one valid mode, thus implying several mode directives. For example, to document the possible use modes of the standard `atom_concat/3` predicate, we would write:

```
:- mode(atom_concat(?atom, ?atom, +atom), one_or_more).
:- mode(atom_concat(+atom, +atom, -atom), one).
```

The first `mode/2` directive specifies that the `atom_concat/3` predicate can be used to split an atom into a prefix and a suffix. The second `mode/2` directive specifies that concatenating two atoms results in a new atom. There are often several alternative `mode/2` directives that can be used to specify a predicate. For example, an alternative to the second `mode/2` directive above would be:

```
:- mode(atom_concat(+atom, +atom, ?atom), zero_or_one).
```

In this case, the same information is provided by both alternatives. But the first alternative is simpler and thus preferred.

Some old Prolog compilers supported some sort of mode directives to improve performance. To the best of my knowledge, there is no modern Prolog compiler supporting this kind of directive for that purpose. The current Logtalk version simply parses this directive for collecting its information for use in the *reflection API* (assuming the *source_data* flag is turned on). In any case, the use of mode directives is a good starting point for documenting your predicates.

Meta-predicate directive

Some predicates may have arguments that will be called as goals, interpreted as *closures* that will be used for constructing goals, or passing meta-arguments to calls to other meta-predicates. To ensure that these goals will be executed in the correct context (i.e., in the *calling context*, not in the meta-predicate *definition context*), we need to use the *meta_predicate/1* directive (in the case of *meta non-terminals*, there's also a *meta_non_terminal/1* directive). For example:

```
:- meta_predicate(findall(*, 0, *)).
:- meta_predicate(map(2, *, *)).
```

The meta-predicate mode arguments in this directive have the following meaning:

0
: Meta-argument that will be called as a goal.

N
: Meta-argument that will be a closure used to construct a call by extending it with `N` arguments. The value of `N` must be a positive integer.

::
: Argument that is context-aware but that will not be called as a goal or a closure. It can contain, however, sub-terms that will be called as goals or closures.

^
: Goal that may be existentially quantified (`Vars^Goal`).

*****
: Normal argument.

The following meta-predicate mode arguments are for use only when writing backend Prolog *adapter files* to deal with proprietary built-in meta-predicates and meta-directives:

/
: Predicate indicator (`Name/Arity`), list of predicate indicators, or conjunction of predicate indicators.

//
: Non-terminal indicator (`Name//Arity`), list of predicate indicators, or conjunction of predicate indicators.

[0]
: List of goals.

[N]
: List of closures.

[/]
: List of predicate indicators.

[//]
: List of non-terminal indicators.

To the best of my knowledge, the use of non-negative integers to specify closures was first introduced on Quintus Prolog for providing information for predicate cross-reference tools.

Note that Logtalk meta-predicate semantics are different from Prolog meta-predicate semantics (assuming a predicate-based module system as common):

- Meta-arguments are always called in the meta-predicate calling context, independent of using explicit or implicit message-sending (to the object defining the meta-predicate when not local). Most Prolog systems have different semantics for explicit versus implicit module qualification.

- Logtalk is not based on a predicate prefixing mechanism. Therefore, the meta-predicate directive is required for any predicate with meta-arguments (including when simply passing the meta-arguments to a call to another meta-predicate). This is usually not required in Prolog systems due to the module prefixing of meta-arguments.
- Sending a message from a meta-predicate definition to call a meta-predicate defined in another object resets the calling context for any passed meta-argument to the object sending the message (including for messages to *self*). Meta-arguments behave differently in Prolog systems due to their module prefixing.
- Logtalk protects from common scenarios where specially crafted meta-predicate definitions are used to break object (and category) encapsulation by changing the meta-arguments passed by client code or trying to subvert the implicit calling context to call client predicates other than the predicates passed as meta-arguments.

Warning

As each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described meta-predicate, even if the meta-predicate declaration is inherited from another entity, to ensure proper compilation of meta-arguments.

Discontiguous directive

The clause of an object (or category) predicate may not be contiguous. In that case, we must declare the predicate discontiguous by using the *discontiguous/1* directive:

```
:- discontiguous(foo/1).
```

This is a directive that we should avoid using: it makes your code harder to read, and it is not supported by some Prolog backends.

Warning

As each Logtalk entity is compiled independently of other entities, this directive must be included in every object or category that contains a definition for the described predicate (even if the predicate declaration is inherited from another entity).

Dynamic directive

An object predicate can be static or dynamic. By default, all predicates (and non-terminals) of static objects defined in source files are static. To declare a dynamic predicate (or non-terminal), we use the *dynamic/1* directive. For example:

```
:- dynamic(foo/1).
```

Predicates of objects dynamically created at runtime (using the *create_object/4* built-in predicate) and predicates of dynamic objects defined in source files (using the *dynamic/0* directive) are implicitly dynamic.

Dynamic predicates can be used to represent persistent mutable object state. Note that static objects may declare and define dynamic predicates. Categories can only declare dynamic predicates (with the importing objects holding the predicate definitions).

> **Warning**
>
> As each Logtalk entity is compiled independently from other entities, this directive must be included in every object that contains a definition for the described predicate (even if the predicate declaration is inherited from another object or imported from a category). If we omit the dynamic declaration then the predicate definition will be compiled static.

Operator directive

An object (or category) predicate can be declared as an operator using the familiar *op/3* directive:

```
:- op(Priority, Specifier, Operator).
```

Operators are local to the object (or category) where they are declared. This means that, if you declare a public predicate as an operator, you cannot use operator notation when sending to an object (where the predicate is visible) the respective message (as this would imply visibility of the operator declaration in the context of the *sender* of the message). If you want to declare global operators and, at the same time, use them inside an entity, just write the corresponding directives at the top of your source file, before the entity opening directive.

Note that operators can also be declared using a scope directive. Only these operators are visible to the *current_op/3* reflection method.

When the same operators are used on several entities within the same source file, the corresponding directives must either be repeated in each entity or appear before any entity that uses them. But in the later case, this results in a global scope for the operators. If you prefer the operators to be local to the source file, just *undefine* them at the end of the file. For example:

```
% before any entity that uses the operator
:- op(400, xfx, results).

...

% after all entities that used the operator
:- op(0, xfx, results).
```

Global operators can be declared in the application loader file.

Uses directive

When a predicate makes heavy use of predicates defined on other objects, its predicate clauses can be verbose due to all the necessary message-sending goals. Consider the following example:

```
foo :-
    ...,
    findall(X, list::member(X, L), A),
    list::append(A, B, C),
    list::select(Y, C, R),
    ...
```

Logtalk provides a directive, *uses/2*, which allows us to simplify the code above. One of the usage templates for this directive is:

```
:- uses(Object, [
    Name1/Arity1, Name2/Arity2, ...
]).
```

Rewriting the code above using this directive results in a simplified and more readable predicate definition:

```
:- uses(list, [
    append/3, member/2, select/3
]).

foo :-
    ...,
    findall(X, member(X, L), A),
    append(A, B, C),
    select(Y, C, R),
    ...
```

Logtalk also supports an extended version of this directive that allows the declaration of *predicate aliases* using the notation `Predicate as Alias` (or the alternative notation `Predicate::Alias`). For example:

```
:- uses(btrees, [new/1 as new_btree/1]).
:- uses(queues, [new/1 as new_queue/1]).
```

You may use this extended version for solving conflicts between predicates declared on several `uses/2` directives or just for giving new names to the predicates that will be more meaningful on their using context.

Predicate aliases can also be used to define *predicate shorthands*, simplifying code maintenance. For example:

```
:- uses(pretty_printer, [
    indent(4, Term) as indent(Term)
]).
```

Assuming multiple calls to the shorthand, a change to the indent value will require a change to a single line instead of changing every call.

Another common use of predicate aliases is changing the order of the predicate arguments without using *lambda expressions*. For example:

```
:- uses(meta, [
    fold_left(Closure, Result0, List, Result) as foldl(Closure, List, Result0, Result)
]).
```

See the directive documentation for details and other examples.

The `uses/2` directive allows simpler predicate definitions as long as there are no conflicts between the predicates declared in the directive and the predicates defined in the object (or category) containing the directive. A predicate (or its alias if defined) cannot be listed in more than one `uses/2` directive. In addition, a `uses/2` directive cannot list a predicate (or its alias if defined) that is defined in the object (or category) containing the directive. Any conflicts are reported by Logtalk as compilation errors.

The object identifier argument can also be a *parameter variable* when using the directive in a parametric object or a parametric category. In this case, *dynamic binding* will necessarily be used for all listed predicates (and non-terminals). The parameter variable must be instantiated at runtime when the messages are sent. This feature simplifies experimenting with multiple implementations of the same protocol (for example, to evaluate the performance of each implementation for a particular case). It also simplifies writing tests that check multiple implementations of the same protocol.

An object (or category) can make a predicate listed in a `uses/2` (or `use_module/2`) directive part of its protocol by simply adding a scope directive for the predicate. For example, in the `statistics` library we have:

```
:- public(modes/2).
:- uses(numberlist, [modes/2]).
```

Therefore, a goal such as `sample::modes(Sample, Modes)` implicitly calls `numberlist::modes(Sample, Modes)` without requiring an explicit local definition for the `modes/2` predicate (which would trigger a compilation error).

Alias directive

Logtalk allows the definition of an alternative name for an inherited or imported predicate (or for an inherited or imported grammar rule non-terminal) through the use of the *alias/2* directive:

```
:- alias(Entity, [
    Predicate1 as Alias1,
    Predicate2 as Alias2,
    ...
]).
```

This directive can be used in objects, protocols, or categories. The first argument, `Entity`, must be an entity referenced in the opening directive of the entity containing the `alias/2` directive. It can be an extended or implemented protocol, an imported category, an extended prototype, an instantiated class, or a specialized class. The second argument is a list of pairs of predicate indicators (or grammar rule non-terminal indicators) using the `as` infix operator.

A common use for the `alias/2` directive is to give an alternative name to an inherited predicate in order to improve readability. For example:

```
:- object(square,
    extends(rectangle)).

    :- alias(rectangle, [width/1 as side/1]).

    ...

:- end_object.
```

The directive allows both `width/1` and `side/1` to be used as messages to the object `square`. Thus, using this directive, there is no need to explicitly declare and define a "new" `side/1` predicate. Note that the `alias/2` directive does not rename a predicate, it only provides an alternative, additional name; the original name continues to be available (although it may be masked due to the default inheritance conflict mechanism).

Another common use for this directive is to solve conflicts when two inherited predicates have the same name and arity. We may want to call the predicate that is masked out by the Logtalk lookup algorithm (see the *Inheritance* section) or we may need to call both predicates. This is simply accomplished by using the `alias/2` directive to give alternative names to masked-out or conflicting predicates. Consider the following example:

```
:- object(my_data_structure,
    extends(list, set)).
```

(continues on next page)

(continued from previous page)

```
    :- alias(list, [member/2 as list_member/2]).
    :- alias(set,  [member/2 as set_member/2]).

    ...

:- end_object.
```

Assuming that both `list` and `set` objects define a `member/2` predicate, without the `alias/2` directives, only the definition of `member/2` predicate in the object `list` would be visible on the object `my_data_structure`, as a result of the application of the Logtalk predicate lookup algorithm. By using the `alias/2` directives, all the following messages would be valid (assuming a public scope for the predicates):

```
% uses list member/2
| ?- my_data_structure::list_member(X, L).

% uses set member/2
| ?- my_data_structure::set_member(X, L).

% uses list member/2
| ?- my_data_structure::member(X, L).
```

When used this way, the `alias/2` directive provides functionality similar to programming constructs of other object-oriented languages that support multi-inheritance (the most notable example probably being the renaming of inherited features in Eiffel).

Note that the `alias/2` directive never hides a predicate that is visible on the entity containing the directive as a result of the Logtalk lookup algorithm. However, it may be used to make visible a predicate that otherwise would be masked by another predicate, as illustrated in the above example.

The `alias/2` directive may also be used to give access to an inherited predicate, which otherwise would be masked by another inherited predicate, while keeping the original name as follows:

```
:- object(my_data_structure,
    extends(list, set)).

    :- alias(list, [member/2 as list_member/2]).
    :- alias(set,  [member/2 as set_member/2]).

    member(X, L) :-
        ^^set_member(X, L).

    ...

:- end_object.
```

Thus, when sending the message `member/2` to `my_data_structure`, the predicate definition in `set` will be used instead of the one contained in `list`.

Documenting directive

A predicate can be documented with arbitrary user-defined information by using the *info/2* directive:

```
:- info(Name/Arity, List).
```

The second argument is a list of `Key is Value` terms. See the *Documenting* section for details.

Multifile directive

A predicate can be declared *multifile* by using the *multifile/1* directive:

```
:- multifile(Name/Arity).
```

This allows clauses for a predicate to be defined in several objects and/or categories. This is a directive that should be used with care. It's commonly used in the definition of *hook predicates*. Multifile predicates (and non-terminals) may also be declared dynamic using the same predicate (or non-terminal) notation (multifile predicates are static by default).

Logtalk precludes using a multifile predicate for breaking object encapsulation by checking that the object (or category) declaring the predicate (using a scope directive) defines it also as multifile. This entity is said to contain the *primary declaration* for the multifile predicate. Entities containing primary multifile predicate declarations must always be compiled before entities defining clauses for those multifile predicates. The Logtalk compiler will print a warning if the scope directive is missing. Note also that the `multifile/1` directive is mandatory when defining multifile predicates.

Consider the following simple example:

```
:- object(main).

    :- public(a/1).
    :- multifile(a/1).
    a(1).

:- end_object.
```

After compiling and loading the `main` object, we can define other objects (or categories) that contribute with clauses for the multifile predicate. For example:

```
:- object(other).

    :- multifile(main::a/1).
    main::a(2).
    main::a(X) :-
        b(X).

    b(3).
    b(4).

:- end_object.
```

After compiling and loading the above objects, you can use queries such as:

```
| ?- main::a(X).

X = 1 ;
X = 2 ;
X = 3 ;
X = 4
yes
```

Note that the order of multifile predicate clauses depends on several factors, including loading order and compiler implementation details. Therefore, your code should never assume or rely on a specific order of the multifile predicate clauses.

When a clause of a multifile predicate is a rule, its body is compiled within the context of the object or category defining the clause. This allows clauses for multifile predicates to call local object or category predicates. But the values of the *sender*, *this*, and *self* in the implicit execution context are passed from the clause head to the clause body. This is necessary to ensure that these values are always valid and to allow multifile predicate clauses to be defined in categories. A call to the `parameter/2` execution context methods, however, retrieves parameters of the entity defining the clause, not from the entity for which the clause is defined. The parameters of the entity for which the clause is defined can be accessed by simple unification at the clause head.

Multifile predicate rules should not contain cuts, as these may prevent other clauses for the predicate from being used by callers. The compiler prints by default a warning when a cut is found in a multifile predicate definition.

Local calls to the database methods from multifile predicate clauses defined in an object take place in the object's own database instead of the database of the entity holding the multifile predicate primary declaration. Similarly, local calls to the `expand_term/2` and `expand_goal/2` methods from a multifile predicate clause look for clauses of the `term_expansion/2` and `goal_expansion/2` hook predicates starting from the entity defining the clause instead of the entity holding the multifile predicate primary declaration. Local calls to the `current_predicate/1`, `predicate_property/2`, and `current_op/3` methods from multifile predicate clauses defined in an object also lookup predicates and their properties in the object's own database instead of the database of the entity holding the multifile predicate primary declaration.

Coinductive directive

A predicate can be declared *coinductive* by using the *coinductive/1* directive. For example:

```
:- coinductive(comember/2).
```

Logtalk support for coinductive predicates is experimental and requires a *backend Prolog compiler* with minimal support for cyclic terms. The value of the read-only *coinduction flag* is set to `supported` for the backend Prolog compilers providing that support.

Synchronized directive

A predicate can be declared *synchronized* by using the *synchronized/1* directive. For example:

```
:- synchronized(write_log_entry/2).
:- synchronized([produce/1, consume/1]).
```

See the section on *synchronized predicates* for details.

1.8.3 Defining predicates

Object predicates

We define object predicates as we have always defined Prolog predicates, the only difference being that we have four more control structures (the three message-sending operators plus the external call operator) to play with. For example, if we wish to define an object containing common utility list predicates like `append/2` or `member/2` we could write something like:

```
:- object(list).

    :- public(append/3).
    append([], L, L).
    append([H| T], L, [H| T2]) :-
        append(T, L, T2).

    :- public(member/2).
    member(H, [H| _]).
    member(H, [_| T]) :-
        member(H, T).

:- end_object.
```

Note that, abstracting from the opening and closing object directives and the scope directives, what we have written is also valid Prolog code. Calls in a predicate definition body default to the local predicates unless we use the message-sending operators or the external call operator. This simplifies conversion from plain Prolog code to Logtalk objects: often we just need to add the necessary encapsulation and scope directives to the old code.

Category predicates

A category can only contain clauses for static predicates. But there are no restrictions in declaring and calling dynamic predicates from inside a category. Because a category can be imported by multiple objects, dynamic predicates must be called either in the context of *self*, using the *message to self* control structure, *(::)/1*, or in the context of *this* (i.e., in the context of the object importing the category). For example, if we want to define a category implementing attributes using the dynamic database of *self* we could write:

```
:- category(attributes).

    :- public(get/2).
    :- public(set/2).

    :- private(attribute_/2).
```

(continues on next page)

(continued from previous page)

```
    :- dynamic(attribute_/2).

    get(Var, Value) :-
        ::attribute_(Var, Value).

    set(Var, Value) :-
        ::retractall(attribute_(Var, _)),
        ::asserta(attribute_(Var, Value).

:- end_category.
```

In this case, the `get/2` and `set/2` predicates will always access/update the correct definition, contained in the object receiving the messages.

In alternative, if we want a category implementing attributes using the dynamic database of *this*, we would write instead:

```
:- category(attributes).

    :- public(get/2).
    :- public(set/2).

    :- private(attribute_/2).
    :- dynamic(attribute_/2).

    get(Var, Value) :-
        attribute_(Var, Value).

    set(Var, Value) :-
        retractall(attribute_(Var, _)),
        asserta(attribute_(Var, Value).

:- end_category.
```

In this case, each object importing the category will have its own clauses for the `attribute_/2` private dynamic predicate.

Meta-predicates

Meta-predicates may be defined inside objects and categories as any other predicate. A meta-predicate is declared using the *meta_predicate/1* directive as described earlier in this section. When defining a meta-predicate, the arguments in the clause heads corresponding to the meta-arguments must be variables. All meta-arguments are called in the context of the object or category calling the meta-predicate. In particular, when sending a message that corresponds to a meta-predicate, the meta-arguments are called in the context of the object or category sending the message.

The most simple example is a meta-predicate with a meta-argument that is called as a goal. E.g. the *ignore/1* built-in predicate could be defined as:

```
:- public(ignore/1).
:- meta_predicate(ignore(0)).
```

(continues on next page)

(continued from previous page)

```
ignore(Goal) :-
   (Goal -> true; true).
```

The `0` in the meta-predicate template tells us that the meta-argument is a goal that will be called by the meta-predicate.

Some meta-predicates have meta-arguments that are not goals but *closures*. Logtalk supports the definition of meta-predicates that are called with closures instead of goals as long as the definition uses the *call/1-N* built-in predicate to call the closure with the additional arguments. A classical example is a list mapping predicate:

```
:- public(map/2).
:- meta_predicate(map(1, *)).

map(_, []).
map(Closure, [Arg| Args]) :-
    call(Closure, Arg),
    map(Closure, Args).
```

Note that in this case the meta-predicate directive specifies that the closure will be extended with exactly one additional argument. When calling a meta-predicate, a closure can correspond to a user-defined predicate, a built-in predicate, a *lambda expression*, or a control construct.

In some cases, it is not a meta-argument but one of its sub-terms that is called as a goal or used as a closure. For example:

```
:- public(call_all/1).
:- meta_predicate(call_all(::)).

call_all([]).
call_all([Goal| Goals]) :-
    call(Goal),
    call_all(Goals).
```

The `::` mode indicator in the meta-predicate template allows the corresponding argument in the meta-predicate definition to be a non-variable term and instructs the compiler to look into the argument sub-terms for goal and closure *meta-variables*.

When a meta-predicate calls another meta-predicate, both predicates require `meta_predicate/1` directives. For example, the `map/2` meta-predicate defined above is usually implemented by exchanging the argument order to take advantage of first-argument indexing:

```
:- meta_predicate(map(1, *)).
map(Closure, List) :-
    map_(List, Closure).

:- meta_predicate(map_(*, 1)).
map_([], _).
map_([Head| Tail], Closure) :-
    call(Closure, Head),
    map_(Tail, Closure).
```

Note that Logtalk, unlike most Prolog module systems, is not based on a predicate prefixing mechanism. Thus, the meta-argument calling context is not part of the meta-argument itself.

Lambda expressions

The use of lambda expressions as meta-predicate goal and *closure* arguments often saves writing auxiliary predicates for the sole purpose of calling the meta-predicates. A simple example of a lambda expression is:

```
| ?- meta::map([X,Y]>>(Y is 2*X), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

In this example, a lambda expression, `[X,Y]>>(Y is 2*X)`, is used as an argument to the `map/3` list mapping predicate, defined in the library object `meta`, in order to double the elements of a list of integers. Using a lambda expression avoids writing an auxiliary predicate for the sole purpose of doubling the list elements. The *lambda parameters* are represented by the list `[X,Y]`, which is connected to the *lambda goal*, `(Y is 2*X)`, by the `(>>)/2` operator. The `map/3` predicate calls the lambda goal with fresh/unique variables, represented by the `X` and `Y` parameters, for each pair of elements of the second and third list arguments.

Currying is supported. I.e. it is possible to write a lambda expression whose goal is another lambda expression. The above example can be rewritten as:

```
| ?- meta::map([X]>>([Y]>>(Y is 2*X)), [1,2,3], Ys).
Ys = [2,4,6]
yes
```

Lambda expressions may also contain *lambda-free variables*. I.e. variables that are global to the lambda expression and shared with the surrounding meta-call context. Consider the following variant of the previous example:

```
| ?- between(1, 3, N), meta::map({N}/[X,Y]>>(Y is N*X), [1,2,3], L).
N = 1, L = [1,2,3] ;
N = 2, L = [2,4,6] ;
N = 3, L = [3,6,9]
yes
```

In this case, the lambda-free variable, `N`, bound by the `between/3` goal, is fixed across all implicit calls made by the `map/3` goal.

A second example of free variables in a lambda expression using GNU Prolog as the backend compiler:

```
| ?- meta::map({Z}/[X,Y]>>(Z#=X+Y), [1,2,3], Zs).
Z = _#22(3..268435455)
Zs = [_#3(2..268435454),_#66(1..268435453),_#110(0..268435452)]
yes
```

The ISO Prolog construct `{}/1` is used for representing the lambda-free variables as this representation is often associated with set representation. Note that the order of the free variables is of no consequence (on the other hand, a list is used for the lambda parameters as their order does matter).

Both lambda free variables and lambda parameters can be any Prolog term. Consider the following example by Markus Triska:

```
| ?- meta::map([A-B,B-A]>>true, [1-a,2-b,3-c], Zs).
Zs = [a-1,b-2,c-3]
yes
```

Lambda expressions can be used, as expected, in non-deterministic queries, as in the following example using SWI-Prolog as the backend compiler and Markus Triska's CLP(FD) library:

```
| ?- meta::map({Z}/[X,Y]>>(clpfd:(Z#=X+Y)), Xs, Ys).
Xs = [],
Ys = [] ;
Xs = [_G1369],
Ys = [_G1378],
_G1369+_G1378#=Z ;
Xs = [_G1579, _G1582],
Ys = [_G1591, _G1594],
_G1582+_G1594#=Z,
_G1579+_G1591#=Z ;
Xs = [_G1789, _G1792, _G1795],
Ys = [_G1804, _G1807, _G1810],
_G1795+_G1810#=Z,
_G1792+_G1807#=Z,
_G1789+_G1804#=Z ;
...
```

As illustrated by the above examples, lambda expression syntax reuses the ISO Prolog construct `{}/1` and the standard operators `(/)/2` and `(>>)/2`, thus avoiding defining new operators, which is always tricky for a portable system such as Logtalk. The operator `(>>)/2` was chosen as it suggests an arrow, similar to the syntax used in other languages such as OCaml and Haskell to connect lambda parameters with lambda functions. This syntax was also chosen in order to simplify parsing, error checking, and compilation of lambda expressions. The full specification of the lambda expression syntax can be found in the *language grammar*.

The compiler checks whenever possible that all variables in a lambda expression are either classified as free variables or as lambda parameters. Non-classified variables in a lambda goal (including any anonymous variables) should be regarded as a programming error. The compiler also checks whenever possible if a variable is classified as both a free variable and a lambda parameter. There are a few cases where a variable playing a dual role is intended, but, in general, this also results from a programming error. A third check verifies that no lambda parameter variable is used elsewhere in a clause. Such cases are either programming errors, when the variable appears before the lambda expression, or bad programming style, when the variable is used after the lambda expression. These linter warnings are controlled by the *lambda_variables* flag. Note that the dynamic features of the language and lack of sufficient information at compile-time may prevent the compiler from checking all uses of lambda expressions. To improve linter coverage, compile code using lambda expressions with the *optimize flag* turned on, as that will result in additional cases of meta-arguments being evaluated for possible optimizations.

> **Warning**
>
> Variables listed in lambda parameters must not be shared with other goals in a clause.

An optimizing meta-predicate and lambda expression compiler, based on the *term-expansion mechanism*, is provided as a standard library for practical performance.

A common use of lambda expressions as closure meta-arguments is to workaround closures always being extended by *appending* additional arguments to construct a goal. For example, assume that we want to filter a list of atoms by a given length. We can use the standard `atom_length/2` predicate despite the argument order by writing:

```
filter(Length, Atoms, Filtered) :-
    meta::include({Length}/[Atom]>>atom_length(Atom,Length), Atoms, Filtered).
```

But Logtalk supports a faster alternative by using predicate aliases to change the argument order when calling library or built-in predicates:

```
:- uses(user, [
   atom_length(Atom, Length) as length_atom(Length, Atom)
]).

filter(Length, Atoms, Filtered) :-
    meta::include(length_atom(Length), Atoms, Filtered).
```

In this case, the performance is no longer dependent on compiling away lambda expressions. The resulting code is also easier to read (and thus debug and maintain). But the `uses/2` directive is implicitly defining an auxiliary predicate, which is exactly what we wanted to avoid in the first place by using a lambda expression.

Redefining built-in predicates

Logtalk built-in predicates and Prolog built-in predicates can be redefined inside objects and categories. Although the redefinition of Logtalk built-in predicates should be avoided, the support for redefining Prolog built-in predicates is a practical requirement given the different sets of proprietary built-in predicates provided by backend Prolog systems.

The compiler supports a *redefined_built_ins* flag, whose default value is *silent*, that can be set to *warning* to alert the user of any redefined Logtalk or Prolog built-in predicate.

The redefinition of Prolog built-in predicates can be combined with the *conditional compilation directives* when writing portable applications where some of the supported backends don't provide a built-in predicate found in the other backends. As an example, consider the de facto standard `msort/2` predicate (which sorts a list while keeping duplicates). This predicate is provided as a built-in predicate in most but not all backends. The `list` library object includes the code:

```
:- if(predicate_property(msort(_, _), built_in)).

    msort(List, Sorted) :-
        {msort(List, Sorted)}.

:- else.

    length(List, Length) :-
        ...

:- endif.
```

I.e. the object will use the built-in predicate when available. Otherwise, it will use the predicate definition provided by the `list` object.

The redefinition of built-in predicates can also be accomplished using *predicate shorthands*. This can be useful when porting code while minimizing the changes. For example, assume that existing code uses the `format/2` de facto standard predicate for writing messages. To convert the code to use the *message printing mechanism*, we could write:

```
:- uses(logtalk, [
    print_message(comment, core, Format+Arguments) as format(Format, Arguments)
]).

process(Crate, Contents) :-
```

(continues on next page)

(continued from previous page)

```
    format('Processing crate ~w...', [Crate]),
    ...,
    format('Filing with ~w...', [Contents]),
    ....
```

The predicate shorthand instructs the compiler to rewrite all `format/2` goals as `logtalk::print_message/3` goals, thus allowing us to reuse the code without changes.

1.8.4 Definite clause grammar rules

Definite clause grammar rules (DCGs) provide a convenient notation to represent the parsing and rewrite rules common of most grammars in Prolog. In Logtalk, definite clause grammar rules can be encapsulated in objects and categories. Currently, the ISO/IEC WG17 group is working on a draft specification for a definite clause grammars Prolog standard. Therefore, in the meantime, Logtalk follows the common practice of Prolog compilers supporting definite clause grammars, extending it to support calling grammar rules contained in categories and objects. A common example of a definite clause grammar is the definition of a set of rules for parsing simple arithmetic expressions:

```
:- object(calculator).

    :- public(parse/2).

    parse(Expression, Value) :-
        phrase(expr(Value), Expression).

    expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
    expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
    expr(X) --> term(X).

    term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
    term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
    term(Z) --> number(Z).

    number(C) --> "+", number(C).
    number(C) --> "-", number(X), {C is -X}.
    number(X) --> [C], {0'0 =< C, C =< 0'9, X is C - 0'0}.

:- end_object.
```

After compiling and loading this object, we can test the grammar rules using the `parse/2` message:

```
| ?- calculator::parse("1+2-3*4", Result).

Result = -9
yes
```

The non-terminals can be called from predicates using the private built-in methods *phrase/2* and *phrase/3* as shown in the example above. When we want to use the built-in methods `phrase/2` and `phrase/3`, the non-terminal used as in the first argument must be within the scope of the *sender*. For the above example, assuming that we want the predicate corresponding to the `expr//1` non-terminal to be public, the corresponding scope directive would be:

```
:- public(expr//1).
```

The `//` infix operator used above tells the Logtalk compiler that the scope directive refers to a grammar rule non-terminal, not to a predicate. The idea is that the predicate corresponding to the translation of the `expr//1` non-terminal will have a number of arguments equal to one plus the number of additional arguments necessary for processing the implicit difference list of tokens.

In the body of a grammar rule, we can call rules that are inherited from ancestor objects, imported from categories, or contained in other objects. This is accomplished by using non-terminals as messages. Using a non-terminal as a message to *self* allows us to call grammar rules in categories and ancestor objects. To call grammar rules encapsulated in other objects, we use a non-terminal as a message to those objects. Consider the following example, containing grammar rules for parsing natural language sentences:

```
:- object(sentence,
    imports(determiners, nouns, verbs)).

    :- public(parse/2).

    parse(List, true) :-
        phrase(sentence, List).
    parse(_, false).

    sentence --> noun_phrase, verb_phrase.

    noun_phrase --> ::determiner, ::noun.
    noun_phrase --> ::noun.

    verb_phrase --> ::verb.
    verb_phrase --> ::verb, noun_phrase.

:- end_object.
```

The categories imported by the object would contain the necessary grammar rules for parsing determiners, nouns, and verbs. For example:

```
:- category(determiners).

    :- private(determiner//0).

    determiner --> [the].
    determiner --> [a].

:- end_category.
```

Along with the message-sending operators (`(::)/1`, `(::)/2`, and `(^^)/1`), we may also use other control constructs such as `(<<)/2`, `(\+)/1`, `!/0`, `(;)/2`, `(->)/2`, `{}/1`, `call//1-N`, and `catch/3` in the body of a grammar rule. When using a backend Prolog compiler that supports modules, we may also use the `(:)/2` control construct.

Warning

The semantics of `(\+)/1` and `(->)/2` control constructs in grammar rules with a terminal or a non-terminal in the **first** argument are problematic due to unrestricted lookahead that may or may not be

valid depending on the grammar rule implicit arguments. By default, the linter will print warnings for such calls (controlled by the *grammar_rules* flag). Preferably restrict the use of the `(\+)/1` control construct to `{}/1` arguments and the use of the `(->)/2` control construct to `{}/1` test arguments.

In addition, grammar rules may contain meta-calls (a variable taking the place of a non-terminal), which are translated to calls of the built-in method `phrase//1`. The *meta_non_terminal/1* directive allows the declaration of non-terminals that have arguments that are meta-called from grammar rules. For example:

```
:- meta_non_terminal(zero_or_more(1, *)).

zero_or_more(Closure, [Terminal| Terminals]) -->
    call(Closure, Terminal), !, zero_or_more(Closure, Terminals).
zero_or_more(_, []) -->
    [].
```

You may have noticed that Logtalk defines *{}/1* as a control construct for bypassing the compiler when compiling a clause body goal. As exemplified above, this is the same control construct that is used in grammar rules for bypassing the expansion of rule body goals when a rule is converted into a clause. Both control constructs can be combined in order to call a goal from a grammar rule body, while bypassing at the same time the Logtalk compiler. Consider the following example:

```
bar :-
    write('bar predicate called'), nl.


:- object(bypass).

    :- public(foo//0).

    foo --> {{bar}}.

:- end_object.
```

After compiling and loading this code, we may try the following query:

```
| ?- logtalk << phrase(bypass::foo, _, _).

bar predicate called
yes
```

This is the expected result, as the expansion of the grammar rule into a clause leaves the `{bar}` goal untouched, which, in turn, is converted into the goal `bar` when the clause is compiled. Note that we tested the `bypass::foo//0` non-terminal by calling the `phrase/3` built-in method in the context of the `logtalk` built-in object. This workaround is necessary due to the Prolog backend implementation of the `phrase/3` predicate not being aware of the Logtalk `(::)/2` message-sending control construct semantics.

A grammar rule non-terminal may be declared as dynamic or discontiguous, as any object predicate, using the same `Name//Arity` notation illustrated above for the scope directives. In addition, grammar rule non-terminals can be documented using the *info/2* directive, as in the following example:

```
:- public(sentence//0).

:- info(sentence//0, [
```

(continues on next page)

(continued from previous page)

```
    comment is 'Rewrites sentence into noun and verb phrases.'
]).
```

Note

Future Logtalk versions may compile grammar rules differently from Prolog traditional compilation to prevent name clashes between non-terminals and predicates. Therefore, you should always call non-terminals from predicates using the `phrase/2-3` built-in methods and always call predicates from grammar rules using the `call//1` built-in method. This recommended practice, besides making your code forward compatible with future Logtalk versions, also makes the code more clear. The linter prints warnings when these guidelines are not followed (notably, when a predicate is called as a non-terminal or a non-terminal is called as a predicate).

1.8.5 Built-in methods

Built-in methods are built-in object and category predicates. These include methods to access message execution context, to find sets of solutions, to inspect objects, for database handling, for term and goal expansion, and for printing messages. Some of them are counterparts to standard Prolog built-in predicates that take into account Logtalk semantics. Similar to Prolog built-in predicates, built-in methods cannot be redefined.

Logic and control methods

The *!/0*, *true/0*, *fail/0*, *false/0*, and *repeat/0* standard control constructs and logic predicates are interpreted as built-in public methods and thus can be used as messages to any object. In practice, they are only used as messages when sending multiple messages to the same object (see the section on *message broadcasting*).

Execution context methods

Logtalk defines five built-in private methods to access an object execution context. These methods are in the common usage scenarios translated to a single unification performed at compile-time with a clause head context argument. Therefore, they can be freely used without worrying about performance penalties. When called from inside a category, these methods refer to the execution context of the object importing the category. These methods are private and cannot be used as messages to objects.

To find the object that received the message under execution, we may use the *self/1* method. We may also retrieve the object that has sent the message under execution using the *sender/1* method.

The method *this/1* enables us to retrieve the name of the object for which the predicate clause whose body is being executed is defined instead of using the name directly. This helps to avoid breaking the code if we decide to change the object name and forget to change the name references. This method may also be used from within a category. In this case, the method returns the object importing the category on whose behalf the predicate clause is being executed.

Here is a short example including calls to these three object execution context methods:

```
:- object(test).

    :- public(test/0).

```

(continues on next page)

(continued from previous page)

```
    test :-
        this(This),
        write('Calling predicate definition in '),
        writeq(This), nl,
        self(Self),
        write('to answer a message received by '),
        writeq(Self), nl,
        sender(Sender),
        write('that was sent by '),
        writeq(Sender), nl, nl.

:- end_object.


:- object(descendant,
    extends(test)).

:- end_object.
```

After compiling and loading these two objects, we can try the following goal:

```
| ?- descendant::test.

Calling predicate definition in test
to answer a message received by descendant
that was sent by user
yes
```

Note that the goals `self(Self)`, `sender(Sender)`, and `this(This)`, being translated to unifications with the clause head context arguments at compile-time, are effectively removed from the clause body. Therefore, a clause such as:

```
predicate(Arg) :-
    self(Self),
    atom(Arg),
    ... .
```

is compiled with the goal `atom(Arg)` as the first condition on the clause body. As such, the use of these context execution methods does not interfere with the optimizations that some Prolog compilers perform when the first clause body condition is a call to a built-in type-test predicate or a comparison operator.

For parametric objects and categories, the method *parameter/2* enables us to retrieve current parameter values (see the section on *parametric objects* for a detailed description). For example:

```
:- object(block(_Color)).

    :- public(test/0).

    test :-
        parameter(1, Color),
        write('Color parameter value is '),
        writeq(Color), nl.
```

(continues on next page)

(continued from previous page)

```
:- end_object.
```

An alternative to the `parameter/2` predicate is to use *parameter variables*:

```
:- object(block(_Color_)).

    :- public(test/0).

    test :-
        write('Color parameter value is '),
        writeq(_Color_), nl.

:- end_object.
```

After compiling and loading either version of the object, we can try the following goal:

```
| ?- block(blue)::test.

Color parameter value is blue
yes
```

Calls to the `parameter/2` method are translated to a compile-time unification when the second argument is a variable. When the second argument is bound, the calls are translated to a call to the built-in predicate `arg/3`.

When type-checking predicate arguments, it is often useful to include the predicate execution context when reporting an argument error. The *context/1* method provides access to that context. For example, assume a predicate `foo/2` that takes an atom and an integer as arguments. We could type-check the arguments by writing (using the library `type` object):

```
foo(A, N) :-
    % type-check arguments
    context(Context),
    type::check(atom, A, Context),
    type::check(integer, N, Context),
    % arguments are fine; go ahead
    ... .
```

Error handling and throwing methods

Besides the *catch/3* and *throw/1* methods inherited from Prolog, Logtalk also provides a set of convenience methods to throw standard `error/2` exception terms: *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, and *system_error/0*. When using these methods, the second argument of the `error/2` exception term is bound to the execution context (as it would be provided by the *context/1* method).

Database methods

Logtalk provides a set of built-in methods for *object database* handling similar to the usual database Prolog predicates: *abolish/1*, *asserta/1*, *assertz/1*, *clause/2*, *retract/1*, and *retractall/1*. These methods always operate on the database of the object receiving the corresponding message. When called locally, these predicates take into account any *uses/2* or *use_module/2* directives that refer to the dynamic predicate being handled. For example, in the following object, the clauses for the `data/1` predicate are retracted and asserted in `user` due to the `uses/2` directive:

```
:- object(an_object).

    :- uses(user, [data/1]).

    :- public(some_predicate/1).
    some_predicate(Arg) :-
        retractall(data(_)),
        assertz(data(Arg)).

:- end_object.
```

When working with dynamic grammar rule non-terminals, you may use the built-in method *expand_term/2* convert a grammar rule into a clause that can then be used with the database methods.

Logtalk also supports `asserta/2`, `assertz/2`, `clause/3`, and `erase/1` built-in methods when run with a backend that supports the corresponding legacy built-in predicates that work with *clause references*.

Meta-call methods

Logtalk supports the generalized *call/1-N* meta-predicate. This built-in private meta-predicate must be used in the implementation of meta-predicates that work with *closures* instead of goals. In addition, Logtalk supports the built-in private meta-predicates *ignore/1*, *once/1*, and *(\+)/1*. These methods cannot be used as messages to objects.

All solutions methods

The usual all solutions meta-predicates are built-in private methods in Logtalk: *bagof/3*, *findall/3*, *findall/4*, and *setof/3*. There is also a *forall/2* method that implements generate-and-test loops. These methods cannot be used as messages to objects.

Reflection methods

Logtalk provides a comprehensive set of built-in predicates and built-in methods for querying about entities and predicates. Some of the information, however, requires that the source files are compiled with the *source_data* flag turned on.

The *reflection API* supports two different views on entities and their contents, which we may call the *transparent box view* and the *black box view*. In the transparent box view, we look into an entity disregarding how it will be used and returning all information available on it, including predicate declarations and predicate definitions. This view is supported by the entity property built-in predicates. In the black box view, we look into an entity from a usage point of view using built-in methods for inspecting object operators and predicates that are within scope from where we are making the call: *current_op/3*, which returns operator

specifications; *predicate_property/2*, which returns predicate properties; and *current_predicate/1*, which enables us to query about user-defined predicate definitions. See below for a more detailed description of these methods.

Definite clause grammar parsing methods and non-terminals

Logtalk supports two definite clause grammar parsing built-in private methods, *phrase/2* and *phrase/3*, with definitions similar to the predicates with the same name found on most Prolog compilers that support definite clause grammars. These methods cannot be used as messages to objects.

Logtalk also supports *phrase//1*, *call//1-N*, and *eos//0* built-in non-terminals. The `call//1-N` non-terminals take a *closure* (which can be a lambda expression) plus zero or more additional arguments and are processed by appending the input list of tokens and the list of remaining tokens to the arguments.

1.8.6 Predicate properties

We can find the properties of visible predicates by calling the *predicate_property/2* built-in method. For example:

```
| ?- bar::predicate_property(foo(_), Property).
```

Note that this method takes into account the predicate's scope declarations. In the above example, the call will only return properties for public predicates.

An object's set of visible predicates is the union of all the predicates declared for the object with all the built-in methods and all the Logtalk and Prolog built-in predicates.

The following predicate properties are supported:

`scope(Scope)`
: The predicate scope (useful for finding the predicate scope with a single call to `predicate_property/2`)

`public`, `protected`, `private`
: The predicate scope (useful for testing if a predicate has a specific scope)

`static`, `dynamic`
: All predicates are either static or dynamic (note, however, that a dynamic predicate can only be abolished if it was dynamically declared)

`logtalk`, `prolog`, `foreign`
: A predicate can be defined in Logtalk source code, Prolog code, or in foreign code (e.g., in C)

`built_in`
: The predicate is a built-in predicate

`multifile`
: The predicate is declared multifile (i.e., it can have clauses defined in multiple files or entities)

`meta_predicate(Template)`
: The predicate is declared as a meta-predicate with the specified template

`coinductive(Template)`
: The predicate is declared as a coinductive predicate with the specified template

`declared_in(Entity)`
: The predicate is declared (using a scope directive) in the specified entity

defined_in(Entity)
The predicate definition is looked up in the specified entity (note that this property does not necessarily imply that clauses for the predicate exist in `Entity`; the predicate can simply be false as per the *closed-world assumption*)

redefined_from(Entity)
The predicate is a redefinition of a predicate definition inherited from the specified entity

non_terminal(NonTerminal//Arity)
The predicate resulted from the compilation of the specified grammar rule non-terminal

alias_of(Predicate)
The predicate (name) is an alias for the specified predicate

alias_declared_in(Entity)
The predicate alias is declared in the specified entity

synchronized
The predicate is declared as synchronized (i.e., it's a deterministic predicate synchronized using a mutex when using a backend Prolog compiler supporting a compatible multi-threading implementation)

Some properties are only available when the entities are defined in source files and when those source files are compiled with the *source_data* flag turned on:

recursive
The predicate definition includes at least one recursive rule

inline
The predicate definition is inlined

auxiliary
The predicate is not user-defined but rather automatically generated by the compiler or the *term-expansion mechanism*

mode(Mode, Solutions)
Instantiation, type, and determinism mode for the predicate (which can have multiple modes)

info(ListOfPairs)
Documentation key-value pairs as specified in the user-defined `info/2` directive

number_of_clauses(N)
The number of clauses for the predicate existing at compilation time (note that this property is not updated at runtime when asserting and retracting clauses for dynamic predicates)

number_of_rules(N)
The number of rules for the predicate existing at compilation time (note that this property is not updated at runtime when asserting and retracting clauses for dynamic predicates)

declared_in(Entity, Line)
The predicate is declared (using a scope directive) in the specified entity in a source file at the specified line (if applicable)

defined_in(Entity, Line)
The predicate is defined in the specified entity in a source file at the specified line (if applicable)

redefined_from(Entity, Line)
The predicate is a redefinition of a predicate definition inherited from the specified entity, which is defined in a source file at the specified line (if applicable)

alias_declared_in(Entity, Line)
The *predicate alias* is declared in the specified entity in a source file at the specified line (if applicable)

The properties `declared_in/1-2`, `defined_in/1-2`, and `redefined_from/1-2` do not apply to built-in methods and Logtalk or Prolog built-in predicates. Note that if a predicate is declared in a category imported by the object, it will be the category name — not the object name — that will be returned by the property `declared_in/1`. The same is true for protocol declared predicates.

Some properties, such as line numbers, are only available when the entity holding the predicates is defined in a source file compiled with the *source_data* flag turned on. Moreover, line numbers are only supported in *backend Prolog compilers* that provide access to the start line of a read term. When such support is not available, the value `-1` is returned for the start and end lines.

1.8.7 Finding declared predicates

We can find, by backtracking, all visible user predicates by calling the *current_predicate/1* built-in method. This method takes into account predicate scope declarations. For example, the following call will only return user predicates that are declared public:

```
| ?- some_object::current_predicate(Name/Arity).
```

The predicate property `non_terminal/1` may be used to retrieve all grammar rule non-terminals declared for an object. For example:

```
current_non_terminal(Object, Name//Args) :-
    Object::current_predicate(Name/Arity),
    functor(Predicate, Functor, Arity),
    Object::predicate_property(Predicate, non_terminal(Name//Args)).
```

Usually, the non-terminal and the corresponding predicate share the same functor, but users should not rely on this always being true.

1.8.8 Calling Prolog predicates

Logtalk is designed for both *robustness* and *portability*. In the context of calling Prolog predicates, robustness requires that the compilation of Logtalk source code must not have *accidental* dependencies on Prolog code that happens to be loaded at the time of the compilation. One immediate consequence is that only Prolog *built-in* predicates are visible from within objects and categories. But Prolog systems provide a widely diverse set of built-in predicates, easily raising portability issues. Relying on non-standard predicates is often unavoidable, however, due to the narrow scope of Prolog standards. Logtalk applications may also require calling user-defined Prolog predicates, either in `user` or in Prolog modules.

Calling Prolog built-in predicates

In predicate clauses and object `initialization/1` directives, predicate calls that are not prefixed with a message-sending, super call, or module qualification operator (`::`, `^^`, or `:`), are compiled to either calls to local predicates or as calls to Logtalk/Prolog built-in predicates. A predicate call is compiled as a call to a local predicate if the object (or category) contains a scope directive, a multifile directive, a dynamic directive, or a definition for the called predicate. When that is not the case, the compiler checks if the call corresponds to a Logtalk or Prolog built-in predicate. Consider the following example:

```
foo :-
    ...,
    write(bar),
    ...
```

The call to the `write/1` predicate will be compiled as a call to the corresponding Prolog standard built-in predicate unless the object (or category) containing the above definition also contains a predicate named `write/1` or a directive for the predicate.

When calling non-standard Prolog built-in predicates or using non-standard Prolog arithmetic functions, we may run into portability problems while trying your applications with different backend Prolog compilers. We can use the compiler *portability flag* to generate warnings for calls to non-standard predicates and arithmetic functions. We can also help document those calls using the *uses/2* directive. For example, a few Prolog systems provide an `atom_string/2` non-standard predicate. We can write (in the object or category calling the predicate):

```
:- uses(user, [atom_string/2])
```

This directive is based on the fact that built-in predicates are visible in plain Prolog (i.e., in `user`). Besides helping to document the dependency on a non-standard built-in predicate, this directive will also silence the compiler portability warning.

Calling Prolog non-standard built-in meta-predicates

Prolog built-in meta-predicates may only be called locally within objects or categories, i.e. they cannot be used as messages. Compiling calls to non-standard, Prolog built-in meta-predicates can be tricky, however, as there is no standard way of checking if a built-in predicate is also a meta-predicate and finding out which are its meta-arguments. But Logtalk supports overriding the original meta-predicate template when not programmatically available or usable. For example, assume a `det_call/1` Prolog built-in meta-predicate that takes a goal as argument. We can add to the object (or category) calling it the directive:

```
:- meta_predicate(user::det_call(0)).
```

Another solution is to explicitly declare non-standard built-in Prolog meta-predicates in the corresponding adapter file using the internal predicate `'$lgt_prolog_meta_predicate'/3`. For example:

```
'$lgt_prolog_meta_predicate'(det_call(_), det_call(0), predicate).
```

The third argument can be either the atom `predicate` or the atom `control_construct`, a distinction that is useful when compiling in debug mode.

Calling Prolog foreign predicates

Prolog systems often support defining *foreign* predicates, i.e. predicates defined using languages other than Prolog using a *foreign language interface*. There isn't, however, any standard for defining, making available, and recognizing foreign predicates. From a Logtalk perspective, the two most common scenarios are calling a foreign predicate (from within an object or a category) and making a set of foreign predicates available as part of an object (or category) protocol. Assuming, as this is the most common case, that foreign predicates are globally visible once made available (using a Prolog system-specific loading or linking procedure), we can simply call them as user-defined plain predicates, as explained in the next section. When defining an object (or category) that makes available foreign predicates, the advisable solution is to name the predicates after the object (or category) and then define object (or category) predicates that call the foreign predicates. Most backend adapter files include support for recognizing foreign predicates that allows the Logtalk compiler to inline calls to the predicates (thus avoiding call indirection overheads).

Calling Prolog user-defined plain predicates

User-defined Prolog plain predicates (i.e., predicates that are not defined in a Prolog module) can be called from within objects or categories by sending the corresponding message to `user`. For example:

```
foo :-
    ...,
    user::bar,
    ...
```

In alternative, we can use the *uses/2* directive and write:

```
:- uses(user, [bar/0]).

foo :-
    ...,
    bar,
    ...
```

Note that `user` is a pseudo-object in Logtalk containing all predicate definitions that are not encapsulated (either in a Logtalk entity or a Prolog module).

When the Prolog predicate is not a meta-predicate, we can also use the *{}/1* compiler bypass control construct. For example:

```
foo :-
    ...,
    {bar},
    ...
```

But note that in this case the *reflection API* will not record the dependency of the `foo/0` predicate on the Prolog `bar/0` predicate as we are effectively bypassing the compiler.

Calling Prolog module predicates

Prolog module predicates can be called from within objects or categories by using explicit qualification. For example:

```
foo :-
    ...,
    module:bar,
    ...
```

You can also use the *use_module/2* directive to call the module predicates using implicit qualification:

```
:- use_module(module, [bar/0]).

foo :-
    ...,
    bar,
    ...
```

Note that the first argument of the `use_module/2` directive, when used within an object or a category, is a *module name*, not a *file specification* (also be aware that Prolog modules are sometimes defined in files with names that differ from the module names).

As loading a Prolog module varies between Prolog systems, the actual loading directive or goal is preferably done from the application *loader file*. An advantage of this approach is that it contributes to a clean separation between *loading* and *using* a resource, with the loader file being the central point that loads all application resources (complex applications often use a *hierarchy* of loader files, but the main idea remains the same).

As an example, assume that we need to call predicates defined in a CLP(FD) Prolog library, which can be loaded using `library(clpfd)` as the file specification. In the loader file, we would add:

```
:- use_module(library(clpfd), []).
```

Specifying an empty import list is often used to avoid adding the module-exported predicates to plain Prolog. In the objects and categories we can then call the library predicates, using implicit or explicit qualification, as explained. For example:

```
:- object(puzzle).

    :- public(puzzle/1).

    :- use_module(clpfd, [
        all_different/1, ins/2, label/1,
        (#=)/2, (#\=)/2,
        op(700, xfx, #=), op(700, xfx, #\=)
    ]).

    puzzle([S,E,N,D] + [M,O,R,E] = [M,O,N,E,Y]) :-
        Vars = [S,E,N,D,M,O,R,Y],
        Vars ins 0..9,
        all_different(Vars),
                  S*1000 + E*100 + N*10 + D +
                  M*1000 + O*100 + R*10 + E #=
        M*10000 + O*1000 + N*100 + E*10 + Y,
        M #\= 0, S #\= 0,
        label([M,O,N,E,Y]).

:- end_object.
```

> **Warning**
>
> The actual module code **must** be loaded prior to the compilation of Logtalk source code that uses it. In particular, programmers should not expect that the module be auto-loaded (including when using a backend Prolog compiler that supports an auto-loading mechanism).

The module identifier argument can also be a *parameter variable* when using the directive in a parametric object or a parametric category. In this case, dynamic binding will necessarily be used for all listed predicates (and non-terminals). The parameter variable must be instantiated at runtime when the calls are made.

Logtalk supports the declaration of *predicate aliases* and *predicate shorthands* in `use_module/2` directives used within objects and categories. For example, the ECLiPSe IC Constraint Solvers define a `(::)/2` variable domain operator that clashes with the Logtalk `(::)/2` message-sending operator. We can solve the conflict by writing:

```
:- use_module(ic, [(::)/2 as ins/2]).
```

With this directive, calls to the `ins/2` predicate alias will be automatically compiled by Logtalk to calls to the

`(::)/2` predicate in the `ic` module.

Calling Prolog module meta-predicates

The Logtalk library provides implementations of common meta-predicates, which can be used in place of module meta-predicates (e.g., list mapping meta-predicates). If that is not the case, the Logtalk compiler may need help to understand the module meta-predicate templates. Despite some recent progress in standardization of the syntax of `meta_predicate/1` directives and of the `meta_predicate/1` property returned by the `predicate_property/2` reflection predicate, portability is still a major problem. Thus, Logtalk allows the original `meta_predicate/1` directive to be **overridden** with a local directive that Logtalk can make sense of. It also allows providing a `meta_predicate/1` directive when it's missing from the module defining the meta-predicate. Note that Logtalk is not based on a predicate prefixing mechanism as found in module systems. This fundamental difference precludes an automated solution at the Logtalk compiler level.

As an example, assume that you want to call from an object (or a category) a module meta-predicate with the following meta-predicate directive:

```
:- module(foo, [bar/2]).

:- meta_predicate(bar(*, :)).
```

The `:` meta-argument specifier is ambiguous. It tell us that the second argument of the meta-predicate is module sensitive, but it does not tell us *how*. Some legacy module libraries and some Prolog systems use `:` to mean `0` (i.e., a meta-argument that will be meta-called). Some others use `:` for meta-arguments that are not meta-called but that still need to be augmented with module information. Whichever the case, the Logtalk compiler doesn't have enough information to unambiguously parse the directive and correctly compile the meta-arguments in the meta-predicate call. Therefore, the Logtalk compiler will generate an error stating that `:` is not a valid meta-argument specifier when trying to compile a `foo:bar/2` goal. There are two alternative solutions for this problem. The advised solution is to override the meta-predicate directive by writing, inside the object (or category) where the meta-predicate is called:

```
:- meta_predicate(foo:bar(*, *)).
```

or:

```
:- meta_predicate(foo:bar(*, 0)).
```

depending on the true meaning of the second meta-argument. The second alternative, only usable when the meta-argument can be handled as a normal argument, is to simply use the *{}/1* compiler bypass control construct to call the meta-predicate as-is:

```
... :- {foo:bar(..., ...)}, ...
```

The downside of this alternative is that it hides the dependency on the module library from the reflection API and thus from the developer tools.

1.8.9 Defining Prolog multifile predicates

Some Prolog module libraries, e.g. constraint packages, expect clauses for some library predicates to be defined in other modules. This is accomplished by declaring the library predicate *multifile* and by explicitly prefixing predicate clause heads with the library module identifier. For example:

```
:- multifile(clpfd:run_propagator/2).
clpfd:run_propagator(..., ...) :-
    ...
```

Logtalk supports the definition of Prolog module multifile predicates in objects and categories. While the clause head is compiled as-is, the clause body is compiled in the same way as a regular object or category predicate, thus allowing calls to local object or category predicates. For example:

```
:- object(...).

    :- multifile(clpfd:run_propagator/2).
    clpfd:run_propagator(..., ...) :-
        % calls to local object predicates
        ...

:- end_object.
```

The Logtalk compiler will print a warning if the `multifile/1` directive is missing. These multifile predicates may also be declared dynamic using the same `Module:Name/Arity` notation.

1.8.10 Asserting and retracting Prolog predicates

To assert and retract clauses for Prolog dynamic predicates, we can use an explicitly qualified module argument. For example:

```
:- object(...).

    :- dynamic(m:bar/1).

    foo(X) :-
        retractall(m:bar(_)),
        assertz(m:bar(X)),
        ...

:- end_object.
```

In alternative, we can use *use_module/2* directives to declare the module predicates. For example:

```
:- object(...).

    :- use_module(m, [bar/1]).
    :- dynamic(m:bar/1).

    foo(X) :-
        % retract and assert bar/1 clauses in module m
        retractall(bar(_)),
        assertz(bar(X)),
```

(continues on next page)

(continued from previous page)

```
        ...

:- end_object.
```

When the Prolog dynamic predicates are defined in `user`, the recommended and most portable practice (as not all backends support a module system) is to use a *uses/2* directive:

```
:- object(...).

    :- uses(user, [bar/1]).
    :- dynamic(user::bar/1).

    foo(X) :-
        % retract and assert bar/1 clauses in user
        retractall(bar(_)),
        assertz(bar(X)),
        ...

:- end_object.
```

Note that in the alternatives using `uses/2` or `use_module/2` directives, the argument of the database handling predicates must be known at compile-time. If that is not the case, you must use either an explicitly-qualified argument or the *{}/1* control construct instead. For example:

```
:- object(...).

    add(X) :-
        % assert clause X in module m
        assertz(m:X),
        ...

    remove(Y) :-
        % retract all clauses in user whose head unifies with Y
        {retractall(Y)},
        ...

:- end_object.
```

1.9 Inheritance

The inheritance mechanisms found in object-oriented programming languages allow the specialization of previously defined objects, avoiding the unnecessary repetition of code and allowing the definition of common functionality for sets of objects. In the context of logic programming, we can interpret inheritance as a form of *theory extension*: an object will virtually contain, besides its own predicates, all the predicates inherited from other objects that are not redefined locally. Inheritance is not, however, the only mechanism for theory extension. Logtalk also supports *composition* using *categories*.

Logtalk uses a depth-first lookup procedure for finding predicate declarations and predicate definitions, as explained below, when a *message* is sent to an object. The lookup procedures locate the entity holding the *predicate declaration* and the entity holding the *predicate definition* using the predicate name and arity. The *alias/2* predicate directive may be used for defining alternative names for inherited predicates, for solving

inheritance conflicts, and for giving access to all inherited definitions (thus overriding the default lookup procedure).

The lookup procedures are used when sending a message (using the *(::)/2*, *(::)/1*, and *[]/1* control constructs) and when making *super* calls (using the *(^^)/1* control construct). The exact details of the lookup procedures depend on the *role* played by the object receiving the message or making the *super* call, as explained next. The lookup procedures are also used by the *current_predicate/1* and *predicate_property/2* reflection predicates.

1.9.1 Protocol inheritance

Protocol inheritance refers to the inheritance of predicate declarations (*scope directives*). These can be contained in objects, protocols, or categories. Logtalk supports single and multi-inheritance of protocols: an object or a category may implement several protocols, and a protocol may extend several protocols.

Lookup order for prototype hierarchies

The lookup order for predicate declarations is first the object, second the implemented protocols (and the protocols that these may extend), third the imported categories (and the protocols that they may implement), and finally the objects that the object extends (following their declaration order). This lookup is performed in depth-first order. When an object inherits two different declarations for the same predicate, by default, only the first one will be considered.

Lookup order for class hierarchies

The lookup order for predicate declarations is first the object classes (following their declaration order), second the classes implemented protocols (and the protocols that these may extend), third the classes imported categories (and the protocols that they may implement), and finally the superclasses of the object classes. This lookup is performed in depth-first order. If the object inherits two different declarations for the same predicate, by default, only the first one will be considered.

1.9.2 Implementation inheritance

Implementation inheritance refers to the inheritance of predicate definitions. These can be contained in objects or in categories. Logtalk supports multi-inheritance of implementation: an object may import several categories or extend, specialize, or instantiate several objects.

Lookup order for prototype hierarchies

The lookup order for predicate definitions is similar to the lookup for predicate declarations, except that implemented protocols are ignored (as they can only contain predicate directives).

Lookup order for class hierarchies

The lookup order for predicate definitions is similar to the lookup for predicate declarations, except that implemented protocols are ignored (as they can only contain predicate directives) and that the lookup starts at the instance itself (that received the message) before proceeding, if no predicate definition is found there, to the instance classes imported categories and then to the class superclasses.

Redefining inherited predicate definitions

When we define a predicate that is already inherited from an ancestor object or an imported category, the inherited definition is hidden by the new definition. This is called inheritance overriding: a local definition overrides any inherited definitions. For example, assume that we have the following two objects:

```
:- object(root).

    :- public(bar/1).
    bar(root).

    :- public(foo/1).
    foo(root).

:- end_object.


:- object(descendant,
    extends(root)).

    foo(descendant).

:- end_object.
```

After compiling and loading these objects, we can check the overriding behavior by trying the following queries:

```
| ?- root::(bar(Bar), foo(Foo)).

Bar = root
Foo = root
yes


| ?- descendant::(bar(Bar), foo(Foo)).

Bar = root
Foo = descendant
yes
```

However, we can explicitly code other behaviors. Some examples follow.

Specializing inherited predicate definitions

Specialization of inherited definitions: the new definition calls the inherited definition and makes additional calls. This is accomplished by calling the *(^ ^)/1 super call* operator in the new definition. For example, assume a `init/0` predicate that must account for object specific initializations along the inheritance chain:

```
:- object(root).

    :- public(init/0).

    init :-
        write('root init'), nl.

:- end_object.


:- object(descendant,
    extends(root)).

    init :-
        write('descendant init'), nl,
        ^^init.

:- end_object.
```

```
| ?- descendant::init.

descendant init
root init
yes
```

Union of inherited and local predicate definitions

Union of the new with the inherited definitions: all the definitions are taken into account, the calling order being defined by the inheritance mechanisms. This can be accomplished by writing a clause that just calls, using the *(^ ^)/1 super call* operator, the inherited definitions. The relative position of this clause among the other definition clauses sets the calling order for the local and inherited definitions. For example:

```
:- object(root).

    :- public(foo/1).

    foo(1).
    foo(2).

:- end_object.


:- object(descendant,
    extends(root)).

    foo(3).
```

(continues on next page)

(continued from previous page)

```
    foo(Foo) :-
        ^^foo(Foo).

:- end_object.
```

```
| ?- descendant::foo(Foo).

Foo = 3 ;
Foo = 1 ;
Foo = 2 ;
no
```

Selective inheritance of predicate definitions

Selective inheritance of predicate definitions (also known as differential inheritance) is normally used in the representation of exceptions to inherited default definitions. We can use the *(^^)/1 super call* operator to test and possibly reject some of the inherited definitions. A common example is representing flightless birds:

```
:- object(bird).

    :- public(mode/1).

    mode(walks).
    mode(flies).

:- end_object.


:- object(penguin,
    extends(bird)).

    mode(swims).
    mode(Mode) :-
        ^^mode(Mode),
        Mode \== flies.

:- end_object.
```

```
| ?- penguin::mode(Mode).

Mode = swims ;
Mode = walks ;
no
```

1.9.3 Public, protected, and private inheritance

To make all *public predicates* declared via implemented protocols, imported categories, or ancestor objects *protected predicates* or to make all public and protected predicates *private predicates*, we prefix the entity's name with the corresponding keyword. For example:

```
:- object(Object,
    implements(private::Protocol)).

    % all the Protocol public and protected
    % predicates become private predicates
    % for the Object clients

    ...

:- end_object.
```

or:

```
:- object(Class,
    specializes(protected::Superclass)).

    % all the Superclass public predicates become
    % protected predicates for the Class clients

    ...

:- end_object.
```

Omitting the scope keyword is equivalent to using the public scope keyword. For example:

```
:- object(Object,
    imports(public::Category)).

    ...

:- end_object.
```

This is the same as:

```
:- object(Object,
    imports(Category)).

    ...

:- end_object.
```

This way we ensure backward compatibility with older Logtalk versions and a simplified syntax when protected or private inheritance is not used.

1.9.4 Multiple inheritance

Logtalk supports multiple inheritance by enabling an object to extend, instantiate, or specialize more than one object. Likewise, a protocol may extend multiple protocols, and a category may extend multiple categories. In this case, the depth-first lookup algorithms described above traverse the list of entities per relation from left to right. Consider as an example the following object opening directive:

```
:- object(foo,
    extends((bar, baz))).
```

The lookup procedure will look first into the parent object *bar* and its related entities before looking into the parent object *baz*. The *alias/2* predicate directive can always be used to solve multiple inheritance conflicts. It should also be noted that the multi-inheritance support does not affect performance when we use single inheritance.

1.9.5 Composition versus multiple inheritance

It is not possible to discuss inheritance mechanisms without referring to the long and probably endless debate on single versus multiple inheritance. The single inheritance mechanism can be implemented efficiently, but it imposes several limitations on reusing, even if the multiple characteristics we intend to inherit are orthogonal. On the other hand, the multiple inheritance mechanisms are attractive in their apparent capability of modeling complex situations. However, they include a potential for conflict between inherited definitions whose variety does not allow a single and satisfactory solution for all the cases.

No solution that we might consider satisfactory for all the problems presented by the multiple inheritance mechanisms has been found. From the simplicity of some extensions that use the Prolog search strategy, such as [McCabe92] or [Moss94], to the sophisticated algorithms of CLOS [Bobrow_et_al_88], there is no adequate solution for all the situations. Besides, the use of multiple inheritance carries some complex problems in the domain of software engineering, particularly in the reuse and maintenance of the applications. All these problems are substantially reduced if we preferably use in our software development composition mechanisms instead of specialization mechanisms [Taenzer89]. Multiple inheritance is best used as an analysis and project abstraction, rather than as an implementation technique [Shan_et_al_93]. Note that Logtalk provides first-class support for composition using *categories*.

1.10 Event-driven programming

The addition of event-driven programming capacities to the Logtalk language [Moura94] is based on a simple but powerful idea:

> The computations must result not only from message-sending but also from the **observation** of message-sending.

The need to associate computations to the occurrence of events was very early recognized in knowledge representation languages, programming languages [Stefik_et_al_86], [Moon86], operative systems [Tanenbaum87], and graphical user interfaces.

With the integration between object-oriented and event-driven programming, we intend to achieve the following goals:

- Minimize the *coupling* between objects. An object should only contain what is intrinsic to it. If an object observes another object, that means that it should depend only on the public protocol of the object observed and not on the implementation of that protocol.
- Provide a mechanism for building *reflexive systems* in Logtalk based on the dynamic behavior of objects in complement to the reflective information on object predicates and relations.

- Provide a mechanism for easily defining method *pre- and post-conditions* that can be toggled using the *events* compiler flag. The pre- and post-conditions may be defined in the same object containing the methods or distributed between several objects acting as method monitors.
- Provide a *publish-subscribe* mechanism where public messages play the role of events.

1.10.1 Definitions

The words *event* and *monitor* have multiple meanings in computer science. To avoid misunderstandings, we start by defining them in the Logtalk context.

Event

In an object-oriented system, all computations start through message sending. It thus becomes quite natural to declare that the only event that can occur in this kind of system is precisely the sending of a message. An event can thus be represented by the ordered tuple `(Object, Message, Sender)`.

If we consider message processing an indivisible activity, we can interpret the sending of a message and the return of the control to the object that has sent the message as two distinct events. This distinction allows us to have more precise control over a system's dynamic behavior. In Logtalk, these two types of events have been named `before` and `after`, respectively for sending a message and for returning control to the sender. Therefore, we refine our event representation using the ordered tuple `(Event, Object, Message, Sender)`.

The implementation of events in Logtalk enjoys the following properties:

Independence between the two types of events
: We can choose to watch only one event type or to process each one of the events associated with a message-sending goal in an independent way.

All events are automatically generated by the message-sending mechanism
: The task of generating events is transparently accomplished by the message-sending mechanism. The user only needs to define the events that will be monitored.

The events watched at any moment can be dynamically changed during program execution
: The notion of event allows the user not only to have the possibility of observing but also of controlling and modifying an application behavior, namely by dynamically changing the observed events during program execution. It is our goal to provide the user with the possibility of modeling the largest number of situations.

Monitor

Complementary to the notion of event is the notion of monitor. A monitor is an object that is automatically notified by the message-sending mechanism whenever a registered event occurs. Any object that defines the event-handling predicates can play the role of a monitor.

The implementation of monitors in Logtalk enjoys the following properties:

Any object can act as a monitor
: The monitor status is a role that any object can perform during its existence. The minimum protocol necessary is declared in the built-in monitoring protocol. Strictly speaking, the reference to this protocol is only needed when specializing event handlers. Nevertheless, it is considered good programming practice to always refer to the protocol when defining event handlers.

Unlimited number of monitors for each event
: Several monitors can observe the same event for distinct reasons. Therefore, the number of monitors per event is bounded only by the available computing resources.

The monitor status of an object can be dynamically changed at runtime
: This property does not imply that an object must be dynamic to act as a monitor (the monitor status of an object is not stored in the object).

Event handlers cannot modify the event arguments
: Notably, if the message contains unbound variables, these cannot be bound by the calls to the monitor event handlers.

1.10.2 Event generation

Assuming that the *events* flag is set to `allow` for the object (or category) sending the messages we want to observe, for each message that is sent using the *(::)/2* control construct, the runtime system automatically generates two events. The first — *before event* — is generated when the message is sent. The second — *after event* — is generated after the message has successfully been executed.

Note that *self* messages (using the *(::)/1* control construct) and *super* calls (using the *(^^)/1* control construct) don't generate events.

1.10.3 Communicating events to monitors

Whenever a spied event occurs, the message-sending mechanism calls the corresponding event handlers directly for all registered monitors. These calls are internally made, thus bypassing the message-sending primitives in order to avoid potential endless loops. The event handlers consist of user definitions for the public predicates declared in the built-in monitoring protocol (see below for more details).

1.10.4 Performance concerns

Ideally, the existence of monitored messages should not affect the processing of the remaining messages. On the other hand, for each message that has been sent, the system must verify if its respective event is monitored. Whenever possible, this verification should be performed in constant time and independently of the number of monitored events. The representation of events takes advantage of the first argument indexing performed by most Prolog compilers, which ensure — in the general case — access in constant time.

Event support can be turned off on a per-object (or per-category) basis using the *events* compiler flag. With event support turned off, Logtalk uses optimized code for processing message-sending calls that skips the checking of monitored events, resulting in a small but measurable performance improvement.

1.10.5 Monitor semantics

The established semantics for monitor actions consists of considering its success as a necessary condition so that a message can succeed:

- All actions associated with events of type `before` must succeed so that the message processing can start.
- All actions associated with events of type `after` also have to succeed so that the message itself succeeds. The failure of any action associated with an event of type `after` forces backtracking over the message execution (the failure of a monitor never causes backtracking over the preceding monitor actions).

Note that this is the most general choice. If we require a transparent presence of monitors in a message processing, we just have to define the monitor actions in such a way that they never fail (which is very simple to accomplish).

1.10.6 Activation order of monitors

Ideally, whenever there are several monitors defined for the same event, the calling order should not interfere with the result. However, this is not always possible. In the case of an event of type `before`, the failure of a monitor prevents a message from being sent and prevents the execution of the remaining monitors. In the case of an event of type `after`, a monitor failure will force backtracking over message execution. Different orders of monitor activation can therefore lead to different results if the monitor actions imply object modifications unrecoverable in case of backtracking. Therefore, the order for monitor activation should be assumed as arbitrary. In effect, to assume or to try to impose a specific sequence requires a global knowledge of an application dynamics, which is not always possible. Furthermore, that knowledge can reveal itself as incorrect if there is any change in the execution conditions. Note that, given the independence between monitors, it does not make sense that a failure forces backtracking over the actions previously executed.

1.10.7 Event handling

Logtalk provides three built-in predicates for event handling. These predicates support defining, enumerating, and abolishing events. Applications that use events extensively usually define a set of objects that use these built-in predicates to implement more sophisticated and higher-level behavior.

Defining new events

New events can be defined using the *define_events/5* built-in predicate:

```
| ?- define_events(Event, Object, Message, Sender, Monitor).
```

Note that if any of the `Event`, `Object`, `Message`, and `Sender` arguments is a free variable or contains free variables, this call will define a **set** of matching events.

Abolishing defined events

Events that are no longer needed may be abolished using the *abolish_events/5* built-in predicate:

```
| ?- abolish_events(Event, Object, Message, Sender, Monitor).
```

If called with free variables, this goal will remove all matching events.

Finding defined events

The events that are currently defined can be retrieved using the *current_event/5* built-in predicate:

```
| ?- current_event(Event, Object, Message, Sender, Monitor).
```

Note that this predicate will return **sets** of matching events if some of the returned arguments are free variables or contain free variables.

Defining event handlers

The monitoring built-in protocol declares two public predicates, *before/3* and *after/3*, that are automatically called to handle `before` and `after` events. Any object that plays the role of monitor must define one or both of these event handler methods:

```
before(Object, Message, Sender) :-
    ... .

after(Object, Message, Sender) :-
    ... .
```

The arguments in both methods are instantiated by the message-sending mechanism when a monitored event occurs. For example, assume that we want to define a monitor called `tracer` that will track any message sent to an object by printing a descriptive text to the standard output. Its definition could be something like:

```
:- object(tracer,
    % built-in protocol for event handler methods
    implements(monitoring)).

    before(Object, Message, Sender) :-
        write('call: '), writeq(Object),
        write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

    after(Object, Message, Sender) :-
        write('exit: '), writeq(Object),
        write(' <-- '), writeq(Message),
        write(' from '), writeq(Sender), nl.

:- end_object.
```

Assume that we also have the following object:

```
:- object(any).

    :- public(bar/1).
    bar(bar).

    :- public(foo/1).
    foo(foo).

:- end_object.
```

After compiling and loading both objects and setting the *events* flag to `allow`, we can start tracing every message sent to any object by calling the *define_events/5* built-in predicate:

```
| ?- set_logtalk_flag(events, allow).

yes

| ?- define_events(_, _, _, _, tracer).

yes
```

From now on, every message sent from `user` to any object will be traced to the standard output stream:

```
| ?- any::bar(X).

call: any <-- bar(X) from user
exit: any <-- bar(bar) from user
X = bar

yes
```

To stop tracing, we can use the *abolish_events/5* built-in predicate:

```
| ?- abolish_events(_, _, _, _, tracer).

yes
```

The monitoring protocol declares the event handlers as public predicates. If necessary, *protected or private implementation of the protocol* may be used in order to change the scope of the event handler predicates. Note that the message-sending processing mechanism is able to call the event handlers irrespective of their scope. Nevertheless, the scope of the event handlers may be restricted in order to prevent other objects from calling them.

The pseudo-object *user* can also act as a monitor. This object expects the `before/3` and `after/3` predicates to be defined in the plain Prolog database. To avoid predicate existence errors when setting `user` as a monitor, this object declares the predicates multifile. Thus, any plain Prolog code defining the predicates should include the directives:

```
:- multifile(before/3).
:- multifile(after/3).
```

1.11 Multi-threading programming

Logtalk provides **experimental** support for multi-threading programming on selected Prolog compilers. Logtalk makes use of the low-level Prolog built-in predicates that implement message queues and interface with POSIX threads and mutexes (or a suitable emulation), providing a small set of high-level predicates and directives that allows programmers to easily take advantage of modern multi-processor and multi-core computers without worrying about the tricky details of creating, synchronizing, or communicating with threads, mutexes, and message queues. Logtalk multi-threading programming integrates with object-oriented programming by providing a *threaded engines* API, enabling objects and categories to prove goals concurrently, and supporting synchronous and asynchronous messages.

1.11.1 Enabling multi-threading support

Multi-threading support may be disabled by default. It can be enabled on the Prolog adapter files of supported compilers by setting the read-only *threads* compiler flag to `supported`.

1.11.2 Enabling objects to make multi-threading calls

The *threaded/0* object directive is used to enable an object to make multi-threading calls:

```
:- threaded.
```

1.11.3 Multi-threading built-in predicates

Logtalk provides a small set of built-in predicates for multi-threading programming. For simple tasks where you simply want to prove a set of goals, each one in its own thread, Logtalk provides a *threaded/1* built-in predicate. The remaining predicates allow for fine-grained control, including postponing retrieval of thread goal results at a later time, supporting non-deterministic thread goals, and making *one-way* asynchronous calls. Together, these predicates provide high-level support for multi-threading programming, covering most common use cases.

Proving goals concurrently using threads

A set of goals may be proved concurrently by calling the Logtalk built-in predicate *threaded/1*. Each goal in the set runs in its own thread.

When the `threaded/1` predicate argument is a *conjunction* of goals, the predicate call is akin to *and-parallelism*. For example, assume that we want to find all the prime numbers in a given interval, `[N, M]`. We can split the interval into two parts and then span two threads to compute the prime numbers in each sub-interval:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded((
        prime_numbers(N2, M, [], Acc),
        prime_numbers(N, N1, Acc, Primes)
    )).

prime_numbers(N, M, Acc, Primes) :-
    ...
```

The `threaded/1` call terminates when the two implicit threads terminate. In a computer with two or more processors (or with a processor with two or more cores), the code above can be expected to provide better computation times when compared with single-threaded code for sufficiently large intervals.

When the `threaded/1` predicate argument is a *disjunction* of goals, the predicate call is akin to *or-parallelism*, here reinterpreted as a set of goals *competing* to find a solution. For example, consider the different methods that we can use to find the roots of real functions. Depending on the function, some methods will be faster than others. Some methods will converge to the solution while others may diverge and never find it. We can try all the methods simultaneously by writing:

```
find_root(Function, A, B, Error, Zero) :-
    threaded((
        bisection::find_root(Function, A, B, Error, Zero)
    ;   newton::find_root(Function, A, B, Error, Zero)
    ;   muller::find_root(Function, A, B, Error, Zero)
    )).
```

The above `threaded/1` goal succeeds when one of the implicit threads succeeds in finding the function root, leading to the termination of all the remaining competing threads.

The `threaded/1` built-in predicate is most useful for lengthy, independent, deterministic computations where the computational costs of each goal outweigh the overhead of the implicit thread creation and management.

Proving goals asynchronously using threads

A goal may be proved asynchronously using a new thread by calling the *threaded_call/1-2* built-in predicate . Calls to this predicate are always true and return immediately (assuming a callable argument). The term representing the goal is copied, not shared with the thread. The thread computes the first solution to the goal, posts it to the implicit message queue of the object from where the `threaded_call/1` predicate was called, and suspends waiting for either a request for an alternative solution or for the program to commit to the current solution.

The results of proving a goal asynchronously in a new thread may be later retrieved by calling the *threaded_exit/1-2* built-in predicate within the same object where the call to the `threaded_call/1` predicate was made. The `threaded_exit/1` calls suspend execution until the results of the `threaded_call/1` calls are sent back to the object message queue.

The `threaded_exit/1` predicate allows us to retrieve alternative solutions through backtracking (if you want to commit to the first solution, you may use the *threaded_once/1-2* predicate instead of the `threaded_call/1` predicate). For example, assuming a `lists` object implementing the usual `member/2` predicate, we could write:

```
| ?- threaded_call(lists::member(X, [1,2,3])).

X = _G189
yes

| ?- threaded_exit(lists::member(X, [1,2,3])).

X = 1 ;
X = 2 ;
X = 3 ;
no
```

In this case, the `threaded_call/1` and the `threaded_exit/1` calls are made within the pseudo-object `user`. The implicit thread running the `lists::member/2` goal suspends itself after providing a solution, waiting for a request for an alternative solution; the thread is automatically terminated when the runtime engine detects that backtracking to the `threaded_exit/1` call is no longer possible.

Calls to the `threaded_exit/1` predicate block the caller until the object message queue receives the reply to the asynchronous call. The predicate *threaded_peek/1-2* may be used to check if a reply is already available without removing it from the thread queue. The `threaded_peek/1` predicate call succeeds or fails immediately without blocking the caller. However, keep in mind that repeated use of this predicate is equivalent to polling a message queue, which may hurt performance.

Be careful when using the `threaded_exit/1` predicate inside failure-driven loops. When all the solutions have been found (and the thread generating them is therefore terminated), re-calling the predicate will generate an exception. Note that failing instead of throwing an exception is not an acceptable solution, as it could be misinterpreted as a failure of the `threaded_call/1` argument.

The example in the previous section with prime numbers could be rewritten using the `threaded_call/1` and `threaded_exit/1` predicates:

```
prime_numbers(N, M, Primes) :-
    M > N,
    N1 is N + (M - N) // 2,
    N2 is N1 + 1,
    threaded_call(prime_numbers(N2, M, [], Acc)),
    threaded_call(prime_numbers(N, N1, Acc, Primes)),
    threaded_exit(prime_numbers(N2, M, [], Acc)),
    threaded_exit(prime_numbers(N, N1, Acc, Primes)).

prime_numbers(N, M, Acc, Primes) :-
    ...
```

When using asynchronous calls, the link between a `threaded_exit/1` call and the corresponding `threaded_call/1` call is established using unification. If there are multiple `threaded_call/1` calls for a matching `threaded_exit/1` call, the connection can potentially be established with any of them (this is akin to what happens with tabling). Nevertheless, you can easily use a call *tag* by using the alternative *threaded_call/2*, *threaded_once/2*, and *threaded_exit/2* built-in predicates. For example:

```
?- threaded_call(member(X, [1,2,3]), Tag).

Tag = 1
yes

?- threaded_call(member(X, [1,2,3]), Tag).

Tag = 2
yes

?- threaded_exit(member(X, [1,2,3]), 2).

X = 1 ;
X = 2 ;
X = 3
yes
```

When using these predicates, the tags shall be considered as an opaque term; users shall not rely on its type. Tagged asynchronous calls can be canceled by using the *threaded_cancel/1* predicate.

1.11.4 One-way asynchronous calls

Sometimes we want to prove a goal in a new thread without caring about the results. This may be accomplished by using the built-in predicate *threaded_ignore/1*. For example, assume that we are developing a multi-agent application where an agent may send a "happy birthday" message to another agent. We could write:

```
..., threaded_ignore(agent::happy_birthday), ...
```

The call succeeds with no reply of the goal success, failure, or even exception ever being sent back to the object making the call. Note that this predicate implicitly performs a deterministic call of its argument.

1.11.5 Asynchronous calls and synchronized predicates

Proving a goal asynchronously using a new thread may lead to problems when the goal results in side effects such as input/output operations or modifications to an *object database*. For example, if a new thread is started with the same goal before the first one finished its job, we may end up with mixed output, a corrupted database, or unexpected goal failures. In order to solve this problem, predicates (and grammar rule non-terminals) with side effects can be declared as *synchronized* by using the *synchronized/1* predicate directive. Proving a query to a synchronized predicate (or synchronized non-terminal) is internally protected by a mutex, thus allowing for easy thread synchronization. For example:

```
% ensure thread synchronization
:- synchronized(db_update/1).

db_update(Update) :-
    % predicate with side-effects
    ...
```

A second example: assume an object defining two predicates for writing, respectively, even and odd numbers in a given interval to the standard output. Given a large interval, a goal such as:

```
| ?- threaded_call(obj::odd_numbers(1,100)),
     threaded_call(obj::even_numbers(1,100)).

1 3 2 4 6 8 5 7 10 ...
...
```

will most likely result in a mixed-up output. By declaring the `odd_numbers/2` and `even_numbers/2` predicates synchronized:

```
:- synchronized([
    odd_numbers/2,
    even_numbers/2]).
```

one goal will only start after the other one finished:

```
| ?- threaded_ignore(obj::odd_numbers(1,99)),
     threaded_ignore(obj::even_numbers(1,99)).

1 3 5 7 9 11 ...
...
2 4 6 8 10 12 ...
...
```

Note that, in a more realistic scenario, the two `threaded_ignore/1` calls would be made concurrently from different objects. Using the same synchronized directive for a set of predicates implies that they all use the same mutex, as required for this example.

As each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described predicate, even if the predicate declaration is inherited from another entity, in order to ensure proper compilation. Note that a synchronized predicate cannot be declared dynamic. To ensure atomic updates of a dynamic predicate, declare as synchronized the predicate performing the update.

Synchronized predicates may be used as wrappers for messages sent to objects that are not multi-threading aware. For example, assume a `log` object defining a `write_log_entry/2` predicate that writes log en-

tries to a file, thus using side effects on its implementation. We can specify and define, for example, a `sync_write_log_entry/2` predicate as follows:

```
:- synchronized(sync_write_log_entry/2).

sync_write_log_entry(File, Entry) :-
    log::write_log_entry(File, Entry).
```

and then call the `sync_write_log_entry/2` predicate instead of the `write_log_entry/2` predicate from multi-threaded code.

The synchronization directive may be used when defining objects that may be reused in both single-threaded and multi-threaded Logtalk applications. The directive simply makes calls to the synchronized predicates deterministic when the objects are used in a single-threaded application.

1.11.6 Synchronizing threads through notifications

Declaring a set of predicates as synchronized can only ensure that they are not executed at the same time by different threads. Sometimes we need to suspend a thread not on a synchronization lock but on some condition that must hold true for a thread goal to proceed. I.e. we want a thread goal to be suspended until a condition becomes true instead of simply failing. The built-in predicate *threaded_wait/1* allows us to suspend a predicate execution (running in its own thread) until a notification is received. Notifications are posted using the built-in predicate *threaded_notify/1*. A notification is a Prolog term that a programmer chooses to represent some condition becoming true. Any Prolog term can be used as a notification argument for these predicates. Related calls to the `threaded_wait/1` and `threaded_notify/1` must be made within the same object, *this*, as the object message queue is used internally for posting and retrieving notifications.

Each notification posted by a call to the `threaded_notify/1` predicate is consumed by a single `threaded_wait/1` predicate call (i.e., these predicates implement a peer-to-peer mechanism). Care should be taken to avoid deadlocks when two (or more) threads both wait and post notifications to each other.

1.11.7 Threaded engines

Threaded engines provide an alternative to the multi-threading predicates described in the previous sections. An *engine* is a computing thread whose solutions can be lazily computed and retrieved. In addition, an engine also supports a term queue that allows passing arbitrary terms to the engine.

An engine is created by calling the *threaded_engine_create/3* built-in predicate. For example:

```
| ?- threaded_engine_create(X, member(X, [1,2,3]), worker).
yes
```

The first argument is an *answer template* to be used for retrieving solution bindings. The user can name the engine, as in this example where the atom `worker` is used, or have the runtime generate a name, which should be treated as an opaque term.

Engines are scoped by the object within which the `threaded_engine_create/3` call takes place. Thus, different objects can create engines with the same names with no conflicts. Moreover, engines share the visible predicates of the object creating them.

The engine computes the first solution of its goal argument and suspends waiting for it to be retrieved. Solutions can be retrieved one at a time using the *threaded_engine_next/2* built-in predicate:

```
| ?- threaded_engine_next(worker, X).
X = 1
yes
```

The call blocks until a solution is available and fails if there are no solutions left. After returning a solution, this predicate signals the engine to start computing the next one. Note that this predicate is deterministic. In contrast with the `threaded_exit/1-2` built-in predicates, retrieving the next solution requires calling the predicate again instead of backtracking into its call. For example:

```
collect_all(Engine, [Answer| Answers]) :-
    threaded_engine_next(Engine, Answer),
    !,
    collect_all(Engine, Answers).
collect_all(_, []).
```

There is also a reified alternative version of the predicate, *threaded_engine_next_reified/2*, which returns `the(Answer)`, `no`, and `exception(Error)` terms as answers. Using this predicate, collecting all solutions to an engine uses a different programming pattern:

```
... :-
    ...,
    threaded_engine_next_reified(Engine, Reified),
    collect_all_reified(Reified, Engine, Answers),
    ...

collect_all_reified(no, _, []).
collect_all_reified(the(Answer), Engine, [Answer| Answers]) :-
    threaded_engine_next_reified(Engine, Reified),
    collect_all_reified(Reified, Engine, Answers).
```

Engines must be explicitly terminated using the *threaded_engine_destroy/1* built-in predicate:

```
| ?- threaded_engine_destroy(worker).
yes
```

A common usage pattern for engines is to define a recursive predicate that uses the engine term queue to retrieve a task to be performed. For example, assume we define the following predicate:

```
loop :-
    threaded_engine_fetch(Task),
    handle(Task),
    loop.
```

The *threaded_engine_fetch/1* built-in predicate fetches a task for the engine term queue. The engine clients would use the *threaded_engine_post/2* built-in predicate to post tasks into the engine term queue. The engine would be created using the call:

```
| ?- threaded_engine_create(none, loop, worker).

yes
```

The `handle/1` predicate, after performing a task, can use the *threaded_engine_yield/1* built-in predicate to make the task results available for consumption using the `threaded_engine_next/2` and `threaded_engine_next_reified/2` built-in predicates. Blocking semantics are used by these two predi-

cates: the `threaded_engine_yield/1` predicate blocks until the returned solution is consumed, while the `threaded_engine_next/2` predicate blocks until a solution becomes available.

1.11.8 Multi-threading performance

The performance of multi-threading applications is highly dependent on the *backend Prolog compiler*, the operating-system, and the use of *dynamic binding* and dynamic predicates. All compatible backend Prolog compilers that support multi-threading features make use of POSIX threads or *pthreads*. The performance of the underlying pthreads implementation can exhibit significant differences between operating systems. An important point is synchronized access to dynamic predicates. As different threads may try to simultaneously access and update dynamic predicates, these operations may use a lock-free algorithm or be protected by a lock, usually implemented using a mutex. In the latter case, poor mutex lock operating-system performance, combined with a large number of collisions by several threads trying to acquire the same lock, can result in severe performance penalties. Thus, whenever possible, avoid using dynamic predicates and dynamic binding.

1.12 Error handling

Error handling is accomplished in Logtalk by using the standard `catch/3` and `throw/1` predicates [ISO95] together with a set of built-in methods that simplify generating errors decorated with expected context.

Errors thrown by Logtalk have, whenever possible, the following format:

```
error(Error, logtalk(Goal, ExecutionContext))
```

In this exception term, `Goal` is the goal that triggered the error `Error` and `ExecutionContext` is the context in which `Goal` is called. For example:

```
error(
    permission_error(modify,private_predicate,p/0),
    logtalk(foo::abolish(p/0), _)
)
```

Note, however, that `Goal` and `ExecutionContext` can be unbound or only partially instantiated when the corresponding information is not available (e.g., due to compiler optimizations that throw away the necessary error context information). The `ExecutionContext` argument is an opaque term that can be decoded using the logtalk::execution_context/7 predicate.

1.12.1 Raising Exceptions

The *error handling section* in the reference manual lists a set of convenient built-in methods that generate `error/2` exception terms with the expected context argument. For example, instead of manually constructing a type error as in:

```
...,
context(Context),
throw(error(type_error(atom, 42), Context)).
```

we can simply write:

```
...,
type_error(atom, 42).
```

The provided error built-in methods cover all standard error types found in the ISO Prolog Core standard.

1.12.2 Type-checking

One of the most common cases where errors may be generated is when type-checking predicate arguments and input data before processing it. The standard library includes a type object that defines an extensive set of types, together with predicates for validating and checking terms. The set of types is user extensible. New types can be defined by adding clauses for the `type/1` and `check/2` multifile predicates. For example, assume that we want to be able to check *temperatures* expressed in Celsius, Fahrenheit, or Kelvin scales. We start by declaring (in an object or category) the new type:

```
:- multifile(type::type/1).
type::type(temperature(_Unit)).
```

Next, we need to define the actual code that would verify that a temperature is valid. As the different scales use a different value for absolute zero, we can write:

```
:- multifile(type::check/2).
type::check(temperature(Unit), Term) :-
    check_temperature(Unit, Term).

% given that temperature has only a lower bound, we make use of the library
% property/2 type to define the necessary test expression for each unit
check_temperature(celsius, Term) :-
    type::check(property(float, [Temperature]>>(Temperature >= -273.15)), Term).
check_temperature(fahrenheit, Term) :-
    type::check(property(float, [Temperature]>>(Temperature >= -459.67)), Term).
check_temperature(kelvin, Term) :-
    type::check(property(float, [Temperature]>>(Temperature >= 0.0)), Term).
```

With this definition, a term is first checked that it is a float value before checking that it is in the expected open interval. But how do we use this new type? If we just want to test if a temperature is valid, we can write:

```
..., type::valid(temperature(celsius), 42.0), ...
```

The type::valid/2 predicate succeeds or fails depending on the second argument being of the type specified in the first argument. If instead of success or failure we want to generate an error for invalid values, we can use the type::check/2 predicate instead:

```
..., type::check(temperature(celsius), 42.0), ...
```

If we require an `error/2` exception term with the error context, we can use instead the type::check/3 predicate:

```
...,
context(Context),
type::check(temperature(celsius), 42.0, Context),
...
```

Note that `context/1` calls are inlined and messages to the library `type` object use *static binding* when compiling with the *optimize flag* turned on, thus enabling efficient type-checking.

1.12.3 Expected terms

Support for representing and handling *expected terms* is provided by the *expecteds* library. Expected terms allow deferring errors to later stages of an application in lieu to raising an exception as soon as an error is detected.

1.12.4 Compiler warnings and errors

The current Logtalk compiler uses the standard `read_term/3` built-in predicate to read and compile a Logtalk source file. This improves the compatibility with *backend Prolog compilers* and their proprietary syntax extensions and standard compliance quirks. But one consequence of this design choice is that invalid Prolog terms or syntax errors may abort the compilation process with limited information given to the user (due to the inherent limitations of the `read_term/3` predicate).

Assuming that all the terms in a source file are valid, there is a set of errors and potential errors, described below, that the compiler will try to detect and report, depending on the used compiler flags (see the *Compiler flags* section of this manual on lint flags for details).

Unknown entities

The Logtalk compiler warns about any referenced entity that is not currently loaded. The warning may reveal a misspelled entity name or just an entity that will be loaded later. Out-of-order loading should be avoided when possible as it prevents some code optimizations, such as *static binding* of messages to methods.

Singleton variables

Singleton variables in a clause are often misspelled variables and, as such, are one of the most common errors when programming in Prolog. Assuming that the *backend Prolog compiler* implementation of the `read_term/3` predicate supports the standard `singletons/1` option, the compiler warns about any singleton variable found while compiling a source file.

Redefinition of Prolog built-in predicates

The Logtalk compiler will warn us of any redefinition of a Prolog built-in predicate inside an object or category. Sometimes the redefinition is intended. In other cases, the user may not be aware that a particular *backend Prolog compiler* may already provide the predicate as a built-in predicate or may want to ensure code portability among several Prolog compilers with different sets of built-in predicates.

Redefinition of Logtalk built-in predicates

Similar to the redefinition of Prolog built-in predicates, the Logtalk compiler will warn us if we try to redefine a Logtalk built-in. But the redefinition will probably be an error in most (if not all) cases.

Redefinition of Logtalk built-in methods

An error will be thrown if we attempt to redefine a Logtalk built-in method inside an entity. The default behavior is to report the error and abort the compilation of the offending entity.

Misspell calls of local predicates

A warning will be reported if Logtalk finds (in the body of a predicate definition) a call to a local predicate that is not defined, built-in (either in Prolog or in Logtalk) or declared dynamic. In most cases these calls are simple misspell errors.

Portability warnings

A warning will be reported if a predicate clause contains a call to a non-standard built-in predicate or arithmetic function. Portability warnings are also reported for non-standard flags or flag values. These warnings often cannot be avoided due to the limited scope of the ISO Prolog standard.

Deprecated elements

A warning will be reported if a deprecated directive, control construct, or predicate is used. These warnings should be fixed as soon as possible, as support for any deprecated features will likely be discontinued in future versions.

Missing directives

A warning will be reported for any missing dynamic, discontiguous, meta-predicate, and public predicate directive.

Duplicated directives

A warning will be reported for any duplicated scope, multifile, dynamic, discontiguous, meta-predicate, and meta-non-terminal directives. Note that conflicting directives for the same predicate are handled as errors, not as duplicated directive warnings.

Duplicated clauses

A warning will be reported for any duplicated entity clauses. This check is computationally heavy, however, and usually turned off by default.

Goals that are always true or false

A warning will be reported for any goal that is always true or false. This is usually caused by typos in the code. For example, writing `X == y` instead of `X == Y`.

Trivial fails

A warning will be reported for any call to a local static predicate with no matching clause.

Suspicious calls

A warning will be reported for calls that are syntactically correct but most likely a semantic error. An example is *(::)/1* calls in clauses that apparently are meant to implement recursive predicate definitions where the user intention is to call the local predicate definition.

Lambda variables

A warning will be reported for *lambda expressions* with unclassified variables (not listed as either *lambda free* or *lambda parameter* variables), for variables playing a dual role (as both lambda free and lambda parameter variables), and for lambda parameters used elsewhere in a clause.

Redefinition of predicates declared in `uses/2` or `use_module/2` directives

A error will be reported for any attempt to define locally a predicate that is already declared in an *uses/2* or *use_module/2* directive.

Other warnings and errors

The Logtalk compiler will throw an error if it finds a predicate clause or a directive that cannot be parsed. The default behavior is to report the error and abort the compilation.

1.12.5 Runtime errors

This section briefly describes runtime errors that result from misuse of Logtalk built-in predicates, built-in methods, or from message-sending. For a complete and detailed description of runtime errors, please consult the Reference Manual.

Logtalk built-in predicates

Most Logtalk built-in predicates check the type and mode of the calling arguments, throwing an exception in case of misuse.

Logtalk built-in methods

Most Logtalk built-in method check the type and mode of the calling arguments, throwing an exception in case of misuse.

Message sending

The message-sending mechanisms always check if the receiver of a message is a defined object and if the message corresponds to a declared predicate within the scope of the sender. The built-in protocol forwarding declares a predicate, *forward/1*, which is automatically called (if defined) by the runtime for any message that the receiving object does not understand. The usual definition for this error handler is to delegate or forward the message to another object that might be able to answer it:

```
forward(Message) :-
    % forward the message while preserving the sender
    [Object::Message].
```

If preserving the original sender is not required, this definition can be simplified to:

```
forward(Message) :-
    Object::Message.
```

More sophisticated definitions are, of course, possible.

1.13 Reflection

Logtalk provides support for both *structural* and *behavioral* reflection. Structural reflection supports computations over an application structure. Behavioral reflection supports computations over what an application does while running. The structural and behavioral reflection APIs are used by all the *developer tools*, which are regular applications.

1.13.1 Structural reflection

Structural reflection allows querying the properties of objects, categories, protocols, and predicates. This API provides two views on the structure of an application: a *transparent-box view* and a *black-box view*, described next.

Transparent-box view

The transparent-box view provides a structural view of the contents and properties of entities, predicates, and source files akin to accessing the corresponding source code. I.e. this is the view we use when asking questions such as: *What predicates are declared in this protocol? Which predicates are called by this predicate? Where are clauses for this multifile predicate defined?*

For entities, built-in predicates are provided for *enumerating entities*, *enumerating entity properties* (including entity declared, defined, called, and updated predicates; i.e. full predicate cross-referencing data), and *enumerating entity relations* (for full entity cross-referencing data). For a detailed description of the supported entity properties, see the sections on *object properties*, *protocol properties*, and *category properties*. For examples of querying entity relations, see the sections on *object relations*, *protocol relations*, and *category relations*.

> **Note**
>
> Some entity and predicate properties are only available when the source files are compiled with the *source_data* flag turned on.

The logtalk built-in object provides predicates for querying loaded source files and their properties.

Black-box view

The black-box view provides a view that takes into account entity encapsulation and thus only allows querying about predicates and operators that are within the scope of the entity calling the reflection methods. This is the view we use when asking questions such as: *What messages can be sent to this object?*

Built-in methods are provided for querying the *predicates that are declared and can be called or used as messages* and for querying the *predicate properties*. It is also possible to enumerate *entity operators*. See the sections on *finding declared predicates* and on *predicate properties* for more details.

1.13.2 Behavioral reflection

Behavioral reflection provides insight on what an application does when running. Specifically, by observing and acting on the messages being exchanged between objects. See the section on *event-driven programming* for details. There is also a *dependents* library that provides an implementation of Smalltalk dependents mechanism.

For use in debugging tools, there is also a small reflection API providing *trace and debug event predicates* provided by the logtalk built-in object.

1.14 Writing and running applications

For successful programming in Logtalk, you need a good working knowledge of Prolog and an understanding of the principles of object-oriented programming. Most guidelines for writing good Prolog code apply as well to Logtalk programming. To those guidelines, you should add the basics of good object-oriented design.

One of the advantages of a system like Logtalk is that it enables us to use the currently available object-oriented methodologies, tools, and metrics [Champaux92] in logic programming. That said, writing applications in Logtalk is similar to writing applications in Prolog: we define new predicates describing what is true about our domain objects, about our problem solution. We encapsulate our predicate directives and definitions inside new objects, categories, and protocols that we create by hand with a text editor or by using the Logtalk built-in predicates. Some of the information collected during the analysis and design phases can be integrated into the objects, categories, and protocols that we define by using the available entity and predicate documenting directives.

1.14.1 Starting Logtalk

We run Logtalk inside a normal Prolog session, after loading the necessary files. Logtalk extends but does not modify your Prolog compiler. We can freely mix Prolog queries with the sending of messages, and our applications can be made of both normal Prolog clauses and object definitions.

Depending on your Logtalk installation, you may use a script or a shortcut to start Logtalk with your chosen Prolog compiler. On POSIX operating-systems, Bash shell integration scripts should be available from the command-line. On Windows, PowerShell integration scripts should be available from the command-line and integration shortcuts should be available from the Start Menu. Scripts are named upon the used backend Prolog compilers.

For example, assuming a POSIX operating-system and GNU Prolog as the backend:

```
$ gplgt
...
```

Depending on your Logtalk installation, you may need to type instead `gplgt.sh`. On Windows, using PowerShell 7.2 or a later version and ECLiPSe as the backend:

```
PS> eclipselgt.ps1
...
```

1.14.2 Running parallel Logtalk processes

Running parallel Logtalk processes is enabled by setting the *clean* flag to `on`. This is the default flag value in the backend adapter files. With this setting, the intermediate Prolog files generated by the Logtalk compiler include the process identifier in the names, thus preventing file name clashes when running parallel processes. When the flag is turned off, the generated intermediate Prolog file names don't include the process identifier and are kept between runs. This is usually done to avoid repeated recompilation of stable code when developing large applications or when running multiple test sets for performance (by avoiding repeated recompilation of the *lgtunit* tool).

To run parallel Logtalk processes with the `clean` flag turned off, each process must use its own *scratch directory*. This is accomplished by defining the `scratch_directory` library alias to a per-process location **before** loading the compiler/runtime. For example, assuming we're using GNU Prolog as the backend, a possible definition could be:

```
:- multifile(logtalk_library_path/2).
:- dynamic(logtalk_library_path/2).

logtalk_library_path(scratch_directory, Directory) :-
    temporary_name(lgtXXXXXX, Name),
    decompose_file_name(Name, _, Prefix, _),
    atom_concat('/tmp/', Prefix, Directory),
    (   file_exists(Directory) ->
        true
    ;   make_directory(Directory)
    ).
```

Assuming the code above is saved in a `parallel_logtalk_processes_setup.pl` file, we would then start Logtalk using:

```
$ gplgt --init-goal "consult('parallel_logtalk_processes_setup.pl')"
```

The details on how to define and load the definition of the `scratch_directory` library alias are, however, backend specific (due to the lack of Prolog standardization) and possibly also operating-system specific (different locations for the temporary directory). The Logtalk library includes a `parallel_logtalk_processes_setup.pl` file with support for selected backends and usage instructions.

1.14.3 Source files

Logtalk source files may define any number of entities (objects, categories, or protocols). Source files may also contain Prolog code interleaved with Logtalk entity definitions. Plain Prolog code is usually copied as-is to the corresponding Prolog output file (except, of course, if subject to the *term-expansion mechanism*). Prolog modules are compiled as objects. The following Prolog directives are processed when read (thus affecting the compilation of the source code that follows): `ensure_loaded/1`, `use_module/1-2`, `op/3`, and `set_prolog_flag/2`. The *initialization/1* directive may be used for defining an initialization goal to be executed when loading a source file.

Logtalk source files can include the text of other files by using the *include/1* directive. Although there is also a standard Prolog `include/1` directive, any occurrences of this directive in a Logtalk source file is handled by the Logtalk compiler, not by the *backend Prolog compiler*, to improve portability.

When writing a Logtalk source file, the following advice applies:

- When practical and when performance is critical, define each entity on its own source file.
- Source file loading order can impact performance (e.g., if an object imports a category defined in a source file loaded after the object source file, no static binding optimizations will be possible).
- Initialization directives that result in the compilation and loading of other source files (e.g., libraries) should preferably be written in the application loader file to ensure the availability of the entities they define when compiling the application source files (thus enabling static binding optimizations).

Naming conventions

When defining each entity in its own source file, it is recommended that the source file be named after the entity identifier. For parametric objects, the identifier arity can be appended to the identifier functor. By default, all Logtalk source files use the extension `.lgt` but this is optional and can be set in the adapter files. For example, we may define an object named `vehicle` and save it in a `vehicle.lgt` source file. A `sort(_)` parametric object would be saved it on a `sort_1.lgt` source file.

Source file text encoding

The text encoding used in a source file may be declared using the *encoding/1* directive when running Logtalk with backend Prolog compilers that support multiple encodings (check the *encoding_directive* flag in the adapter file of your Prolog compiler).

1.14.4 Multi-pass compiler

Logtalk is implemented using a *multi-pass* compiler. In comparison, some Prolog systems use a multi-pass compiler while others use a single-pass compiler. While there are pros and cons with each solution, the most relevant consequence in this context is for the content of source files. In Logtalk, entities and predicates only become available (for the runtime system) after the source file is successfully compiled and loaded. This may prevent some compiler optimizations, notably *static binding*, if some of the referred entities are defined in the same source file. On the other hand, the order of predicate directives and predicate definitions is irrelevant. In contrast, in a system implemented using a single-pass compiler, the order of the source file terms can and often is significant for proper and successful compilation. In these systems, predicates may become available for calling as soon as they are compiled, even if the rest of the source file is yet to be compiled.

The Logtalk compiler reads source files using the Prolog standard `read_term/3` predicate. This ensures compatibility with any syntax extensions that the used backend may implement. In the first compiler stage, all source file terms are read, and data about all defined entities, directives, predicates, and grammar rules is collected. Any defined *term-expansion rules* are applied to the read terms. Grammar rules are expanded into

predicate clauses unless expanded by user-defined term-expansion rules. The second stage compiles all initialization goals and clause bodies, taking advantage of the data collected in the first stage and applying any defined goal-expansion rules. Depending on the compilation mode, the generated code can be instrumented for debugging tools or optimized for performance. Linter checks are performed during these two first stages. The final step in the second stage is to write the generated intermediate Prolog code into a temporary file. In the third and final stage, this intermediate Prolog file is compiled and loaded by the used backend. These intermediate files are deleted by default after loading (see the *clean* flag description for details).

1.14.5 Compiling and loading your applications

Your applications will be made of source files containing your objects, protocols, and categories. The source files can be compiled to disk by calling the *logtalk_compile/1* built-in predicate:

```
| ?- logtalk_compile([source_file1, source_file2, ...]).
```

This predicate runs the compiler on each file and, if no fatal errors are found, outputs Prolog source files that can then be consulted or compiled in the usual way by your Prolog compiler.

To compile to disk and also load into memory the source files, we can use the *logtalk_load/1* built-in predicate:

```
| ?- logtalk_load([source_file1, source_file2, ...]).
```

This predicate works in the same way as the predicate `logtalk_compile/1` but also loads the compiled files into memory.

Both predicates expect a source file name or a list of source file names as an argument. The Logtalk source file name extension, as defined in the adapter file (by default, `.lgt`), can be omitted.

If you have more than a few source files, then you may want to use a *loader file* helper file containing the calls to the `logtalk_load/1-2` predicates. Consulting or compiling the loader file will then compile and load all your Logtalk entities into memory (see below for details).

With most *backend Prolog compilers*, you can use the shorthand `{File}` for `logtalk_load(File)` and `{File1, File2, ...}` for `logtalk_load([File1, File2, ...])`. The use these shorthands should be restricted to the Logtalk/Prolog top-level interpreter, as they are not part of the language specification and may be commented out in case of conflicts with backend Prolog compiler features.

The built-in predicate *logtalk_make/0* can be used to reload all modified source files. With most backend Prolog compilers, you can also use the `{*}` top-level shortcut. Files are also reloaded when the compilation mode changes. An extended version of this predicate, *logtalk_make/1*, accepts multiple targets, including `all`, `clean`, `check`, `circular`, `documentation`, `caches`, `debug`, `normal`, and `optimal`. For example, assume that you have loaded your application files and found a bug. You can easily recompile the files in debug mode by using the `logtalk_make(debug)` goal. After debugging and fixing the bug, you can reload the files in normal mode using the `logtalk_make(normal)` or in optimized mode using the `logtalk_make(optimal)` goal. See the predicates documentation for a complete list of targets and top-level shortcuts. In particular, the `logtalk_make(clean)` goal can be specially useful before switching backend Prolog compilers, as the generated intermediate files may not be compatible. The `logtalk_make(caches)` goal is usually used when benchmarking compiler performance improvements.

1.14.6 Compiler errors, warnings, and comments

Following a Prolog tradition inherited from Quintus Prolog, the compiler prefixes (by default) errors with a `!` and warnings with a `*`. For example:

```
!     Existence error: directive object/1 does not exist
!       in directive end_object/0
!       in file /home/jdoe/logtalk/examples/errors/unmatched_directive.lgt at or above line␣
↪27

*     No matching clause for goal: baz(a)
*       while compiling object main_include_compiler_warning
*       in file /home/jdoe/logtalk/examples/errors/include_compiler_warning.lgt between␣
↪lines 38-39
```

Compiler comments are prefixed by `%`. For example:

```
?- {ack(loader)}.
% [ /home/jdoe/logtalk/examples/ack/ack.lgt loaded ]
% [ /home/jdoe/logtalk/examples/ack/loader.lgt loaded ]
% (0 warnings)
true.
```

1.14.7 Loader files

If you look into the Logtalk distribution, you will notice that most source code directories (e.g., of tools, libraries, and examples) contain a *driver file* that can be used to load all included source files and any required libraries. These loader files are usually named `loader.lgt` or contain the word *loader* in their name. Loader files are ordinary source files and thus compiled and loaded like any source file. By also defining a loader file for your project, you can then load it by simply typing:

```
| ?- {loader}.
```

Another driver file, usually named `tester.lgt` (or containing the word *tester* in its name) is commonly used to load and run tests. By also defining a tester file for your project, you can then run its tests by simply typing:

```
| ?- {tester}.
```

Usually these driver files contain calls to the built-in predicates *set_logtalk_flag/2* (e.g., for setting global, *project-specific*, flag values) and *logtalk_load/1* or *logtalk_load/2* (for loading project files), wrapped inside a Prolog `initialization/1` directive for portability. For instance, if your code is split into three source files named `source1.lgt`, `source2.lgt`, and `source3.lgt`, then the contents of your loader file could be:

```
:- initialization((
    % set project-specific global flags
    set_logtalk_flag(events, allow),
    % load the project source files
    logtalk_load([source1, source2, source3])
)).
```

Another example of directives that are often used in a loader file would be `op/3` directives declaring global operators needed by your project. Loader files are also often used for setting source file-specific compiler

flags (this is useful even when you only have a single source file if you always load it with the same set of compiler flags). For example:

```
:- initialization((
    % set project-specific global flags
    set_logtalk_flag(source_data, off),
    % load the project source files
    logtalk_load(
        [source1, source2, source3],
        % source file-specific flags
        [portability(warning)]),
    logtalk_load(
        [source4, source5],
        % source file-specific flags
        [portability(silent)])
)).
```

To take the best advantage of loader and tester files, define a clause for the multifile and dynamic `logtalk_library_path/2` predicate for the directory containing your source files as explained in the next section.

When your project also uses Prolog module resources, the loader file is also the advised place to load them, preferably without any exports. For example:

```
:- use_module(library(clpfd), []).
...

:- initialization((
    ...
)).
```

Complex projects often use a main loader file that loads the loader files of each of the project components. Thus, loader files provide a central point to understand a project organization and dependencies.

It is worth mentioning a common mistake when writing the first loader files. New users sometimes try to set compiler flags using `logtalk_load/2` when loading a loader file. For example, by writing:

```
| ?- logtalk_load(loader, [optimize(on)]).
```

This will not work as you might expect, as the compiler flags will only be used in the compilation of the `loader.lgt` file itself and will not affect the compilation of files loaded through the `initialization/1` directive contained on the loader file.

1.14.8 Libraries of source files

Logtalk defines a *library* simply as a directory containing source files. Library locations can be specified by defining or asserting clauses for the dynamic and multifile predicate *logtalk_library_path/2*. For example:

```
:- multifile(logtalk_library_path/2).
:- dynamic(logtalk_library_path/2).

logtalk_library_path(shapes, '$LOGTALKUSER/examples/shapes/').
```

The first argument of the predicate is used as an alias for the path on the second argument. Library aliases may also be used on the second argument. For example:

```
:- multifile(logtalk_library_path/2).
:- dynamic(logtalk_library_path/2).

logtalk_library_path(lgtuser, '$LOGTALKUSER/').
logtalk_library_path(examples, lgtuser('examples/')).
logtalk_library_path(viewpoints, examples('viewpoints/')).
```

This allows us to load a library source file without the need to first change the current working directory to the library directory and then back to the original directory. For example, in order to load a `loader.lgt` file, contained in a library named `viewpoints`, we just need to type:

```
| ?- logtalk_load(viewpoints(loader)).
```

The best way to take advantage of this feature is to load at startup a source file containing clauses for the `logtalk_library_path/2` predicate needed for all available libraries (typically, using a *settings file*, as discussed below). This allows us to load library source files or entire libraries without worrying about library paths, improving code portability. The directory paths on the second argument should always end with the path directory separator character. Most backend Prolog compilers allow the use of environment variables in the second argument of the `logtalk_library_path/2` predicate. Use of POSIX relative paths (e.g., `'../'` or `'./'`) for top-level library directories (e.g., `lgtuser` in the example above) is not advised, as different backend Prolog compilers may start with different initial working directories, which may result in portability problems of your loader files.

This *library notation* provides functionality inspired by the `file_search_path/2` mechanism introduced by Quintus Prolog and later adopted by some other Prolog compilers but with a key difference: there is no fragile search mechanism, and the Logtalk `make` can be used to check for duplicated library aliases. Multiple definitions for the same alias are problematic when using external dependencies, as any third-party update to those dependencies can introduce file name clashes. Note that the potential for these clashes cannot be reliably minimized by a careful ordering of the `logtalk_library_path/2` predicate clauses due to this predicate being multifile and dynamic.

1.14.9 Settings files

Although it is always possible to edit the *backend Prolog compiler* adapter files, the recommended solution to customize compiler flags is to create a `settings.lgt` file in the Logtalk user folder or in the user home folder. Depending on the backend Prolog compiler and the operating-system, is also possible to define per-project settings files by creating a `settings.lgt` file in the project directory and by starting Logtalk from this directory. At startup, Logtalk tries to load a `settings.lgt` file from the following directories, searched in sequence:

- Startup directory (`$LOGTALK_STARTUP_DIRECTORY`)
- Logtalk user directory (`$LOGTALKUSER`)
- User home directory (`$HOME`; `%USERPROFILE%` on Windows if `%HOME%` is not defined)
- Application data directory (`%APPDATA%\Logtalk`; only on Windows)
- Config directory (`$XDG_CONFIG_HOME/logtalk`)
- Default config directory (`$HOME/.config/logtalk/`)

The startup directory is only searched when the read-only *settings_file* flag is set to `allow`. When no settings files are found, Logtalk will use the default compiler flag values set on the backend Prolog compiler adapter files. When limitations of the backend Prolog compiler or the operating-system prevent Logtalk from finding the settings files, these can always be loaded manually after Logtalk startup.

Settings files are normal Logtalk source files (although when automatically loaded by Logtalk they are compiled and loaded silently with any errors being reported but otherwise ignored). The usual contents is an `initialization/1` Prolog directive containing calls to the *set_logtalk_flag/2* Logtalk built-in predicate and asserting clauses for the *logtalk_library_path/2* multifile dynamic predicate. Note that the *set_logtalk_flag/2* directive cannot be used as its scope is local to the source file being compiled.

One of the troubles of writing portable applications is the different feature sets of Prolog compilers. Using the Logtalk support for conditional compilation and the *prolog_dialect* flag we can write a single settings file that can be used with several *backend Prolog compilers*:

```
:- if(current_logtalk_flag(prolog_dialect, yap)).

    % YAP specific settings
    ...

:- elif(current_logtalk_flag(prolog_dialect, gnu)).

    % GNU Prolog specific settings
    ...

:- else.

    % generic Prolog settings

:- endif.
```

The Logtalk distribution includes a `samples/settings-sample.lgt` sample file with commented out code snippets for common settings.

1.14.10 Compiler linter

The compiler includes a *linter* that checks for a wide range of possible problems in source files. Notably, the compiler checks for unknown entities, unknown predicates, undefined predicates (i.e., predicates that are declared but not defined), missing directives (including missing `dynamic/1` and `meta_predicate/1` directives), redefined built-in predicates, calls to non-portable predicates, singleton variables, goals that are always true or always false (i.e., goals that can be replaced by `true` or `fail`), and trivial fails (i.e., calls to predicates with no match clauses). Most of the linter warnings are controlled by *compiler flags*. See the next section for details.

1.14.11 Compiler flags

The *logtalk_load/1* and *logtalk_compile/1* always use the current set of default compiler flags as specified in your settings file and the Logtalk adapter files or changed for the current session using the built-in predicate *set_logtalk_flag/2*. Although the default flag values cover the usual cases, you may want to use a different set of flag values while compiling or loading some of your Logtalk source files. This can be accomplished by using the *logtalk_load/2* or the *logtalk_compile/2* built-in predicates. These two predicates accept a list of options affecting how a Logtalk source file is compiled and loaded:

```
| ?- logtalk_compile(Files, Options).
```

or:

```
| ?- logtalk_load(Files, Options).
```

In fact, the `logtalk_load/1` and `logtalk_compile/1` predicates are just shortcuts to the extended versions called with the default compiler flag values. The options are represented by a compound term where the functor is the flag name and the sole argument is the flag value.

We may also change the default flag values from the ones loaded from the adapter file by using the *set_logtalk_flag/2* built-in predicate. For example:

```
| ?- set_logtalk_flag(unknown_entities, silent).
```

The current default flags values can be enumerated using the *current_logtalk_flag/2* built-in predicate:

```
| ?- current_logtalk_flag(unknown_entities, Value).

Value = silent
yes
```

Logtalk also implements a *set_logtalk_flag/2* directive, which can be used to set flags within a source file or within an entity. For example:

```
% compile objects in this source file with event support
:- set_logtalk_flag(events, allow).

:- object(foo).

    % compile this object with support
    % for dynamic predicate declarations
    :- set_logtalk_flag(dynamic_declarations, allow).
    ...

:- end_object.

...
```

Note that the scope of the `set_logtalk_flag/2` directive is local to the entity or to the source file containing it.

Note

Applications should never rely on default flag values for working properly. Whenever the compilation of a source file or an entity requires a specific flag value, the flag should be set explicitly in the entity, in the source file, or in the loader file.

Read-only flags

Some flags have read-only values and thus cannot be changed at runtime. Their values are defined in the Prolog backend *adapter files* These are:

settings_file
: Allows or disables loading of a *settings file* at startup. Possible values are `allow`, `restrict`, and `deny`. The usual default value is `allow` but it can be changed by editing the adapter file when e.g. embedding Logtalk in a compiled application. With a value of `allow`, settings files are searched in the startup directory, in the Logtalk user directory, in the user home directory, in the `APPDATA` if running on Windows, and in the XDG configuration directory. With a value of `restrict`, the search for the settings files skips the startup directory.

prolog_dialect
: Identifier of the *backend Prolog compiler* (an atom). This flag can be used for *conditional compilation* of Prolog compiler specific code.

prolog_version
: Version of the *backend Prolog compiler* (a compound term, `v(Major, Minor, Patch)`, whose arguments are integers). This flag availability depends on the Prolog compiler. Checking the value of this flag fails for any Prolog compiler that does not provide access to version data.

prolog_compatible_version
: Compatible version of the *backend Prolog compiler* (a compound term, usually with the format `@>=(v(Major, Minor, Patch))`, whose arguments are integers). This flag availability depends on the Prolog compiler. Checking the value of this flag fails for any Prolog compiler that does not provide access to version data.

unicode
: Informs Logtalk if the *backend Prolog compiler* supports the Unicode standard. Possible flag values are `unsupported`, `full` (all Unicode planes supported), and `bmp` (supports only the Basic Multilingual Plane).

encoding_directive
: Informs Logtalk if the *backend Prolog compiler* supports the *encoding/1* directive. This directive is used for declaring the text encoding of source files. Possible flag values are `unsupported`, `full` (can be used in both Logtalk source files and compiler generated Prolog files), and `source` (can be used only in Logtalk source files).

tabling
: Informs Logtalk if the *backend Prolog compiler* provides tabling programming support. Possible flag values are `unsupported` and `supported`.

engines
: Informs if the *backend Prolog compiler* provides the required low level multi-threading programming support for Logtalk *threaded engines*. Possible flag values are `unsupported` and `supported`.

threads
: Informs if the *backend Prolog compiler* provides the required low level multi-threading programming support for all high-level Logtalk *multi-threading features*. Possible flag values are `unsupported` and `supported`.

modules
: Informs Logtalk if the *backend Prolog compiler* provides suitable module support. Possible flag values are `unsupported` and `supported` (independently of this flag, Logtalk provides limited support for compiling Prolog modules as objects).

coinduction
: Informs Logtalk if the *backend Prolog compiler* provides the required minimal support for cyclic terms

necessary for working with *coinductive predicates*. Possible flag values are `unsupported` and `supported`.

Version flags

`version_data(Value)`
Read-only flag whose value is the compound term `logtalk(Major,Minor,Patch,Status)`. The first three arguments are integers and the last argument is an atom, possibly empty, representing version status: `aN` for alpha versions, `bN` for beta versions, `rcN` for release candidates (with `N` being a natural number), and `stable` for stable versions. The `version_data` flag is also a de facto standard for Prolog compilers.

Lint flags

`linter(Option)`
Meta-flag for managing the values of all the linter flags as a group. Possible option values are `on` to set all the individual linter flags to `warning`, `off` to set all the individual linter flags to `silent`, and `default` to set all the individual linter flag values to their defaults as defined in the backend adapter files (the usual default). This flag **must** always be defined in the backend adapter files with the value of `default`.

`unknown_entities(Option)`
Controls the unknown entity warnings, resulting from loading an entity that references some other entity that is not currently loaded. Possible option values are `warning` (the usual default) and `silent`. Note that these warnings are not always avoidable, specially when using reflective designs of class-based hierarchies.

`unknown_predicates(Option)`
Defines the compiler behavior when unknown messages or calls to unknown predicates (or non-terminals) are found. An unknown message is a message sent to an object that is not part of the object protocol. An unknown predicate is a called predicate that is neither locally declared or defined. Possible option values are `error`, `warning` (the usual default), and `silent` (not recommended).

`undefined_predicates(Option)`
Defines the compiler behavior when calls to declared but undefined predicates (or non-terminals) are found. Note that these calls will fail at runtime as per closed-world assumption. Possible option values are `error`, `warning` (the usual default), and `silent` (not recommended).

`steadfastness(Option)`
Controls warnings about *possible* non *steadfast* predicate definitions due to variable aliasing at a clause head and a cut in the clause body. Possible option values are `warning` and `silent` (the usual default due to the possibility of false positives).

`portability(Option)`
Controls the non-ISO specified Prolog built-in predicate and non-ISO specified Prolog built-in arithmetic function calls warnings plus use of non-standard Prolog flags and/or flag values. Possible option values are `warning` and `silent` (the usual default).

`deprecated(Option)`
Controls the deprecated predicate warnings. Possible option values are `warning` (the usual default) and `silent`.

`missing_directives(Option)`
Controls the missing predicate directive warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`duplicated_directives(Option)`
Controls the duplicated predicate directive warnings. Possible option values are `warning` (the usual

default) and `silent` (not recommended). Note that conflicting directives for the same predicate are handled as errors, not as duplicated directive warnings.

`trivial_goal_fails(Option)`
Controls the printing of warnings for calls to local static predicates with no matching clauses. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`always_true_or_false_goals(Option)`
Controls the printing of warnings for goals that are always true or false. Possible option values are `warning` (the usual default) and `silent` (not recommended). A unexpected exception in the goal being checked is also reported.

`grammar_rules(Option)`
Controls the printing of grammar rules related warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`arithmetic_expressions(Option)`
Controls the printing of arithmetic expressions related warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`lambda_variables(Option)`
Controls the printing of lambda variable related warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`suspicious_calls(Option)`
Controls the printing of suspicious call warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`redefined_built_ins(Option)`
Controls the Logtalk and Prolog built-in predicate redefinition warnings. Possible option values are `warning` and `silent` (the usual default). Warnings about redefined Prolog built-in predicates are often the result of running a Logtalk application on several Prolog compilers, as each Prolog compiler defines its set of built-in predicates.

`redefined_operators(Option)`
Controls the Logtalk and Prolog built-in operator redefinition warnings. Possible option values are `warning` (the usual default) and `silent`. Redefining Logtalk operators or standard Prolog operators can break term parsing, causing syntax errors or change how terms are parsed, introducing bugs.

`singleton_variables(Option)`
Controls the singleton variable warnings. Possible option values are `warning` (the usual default) and `silent` (not recommended).

`naming(Option)`
Controls warnings about entity, predicate, and variable names per official coding guidelines (which advise using underscores for entity and predicate names and camel case for variable names). Additionally, variable names should not differ only in case. Possible option values are `warning` and `silent` (the usual default due to the current limitation to ASCII names and the computational cost of the checks).

`duplicated_clauses(Option)`
Controls warnings of duplicated entity clauses (and duplicated entity grammar rules). Possible option values are `warning` and `silent` (the usual default due to the required heavy computations). When the term-expansion mechanism is used and results in duplicated clauses, the reported line numbers are for lines of the original clauses that were expanded.

`disjunctions(Option)`
Controls warnings on clauses where the body is a disjunction. Possible option values are `warning` (the usual default) and `silent`. As per coding guidelines, in most cases, these clauses can be rewritten using a clause per disjunction branch for improved code readability.

conditionals(Option)
Controls warnings on if-then-else and soft-cut control constructs. Possible option values are `warning` (the usual default) and `silent`. Warnings include misuse of cuts, potential bugs in the test part, and missing else part (lack of compliance with coding guidelines).

catchall_catch(Option)
Controls warnings on `catch/3` goals that catch all exceptions. Possible option values are `warning` and `silent` (the usual default). Lack of standardization often makes it tricky or cumbersome to avoid too generic `catch/3` goals when writing portable code.

left_recursion(Option)
Controls warnings of left-recursion on clauses and grammar rules. Specifically, when the clause or grammar rule head and the leftmost goal in the body are variants. Possible option values are `warning` (the usual default) and `silent`.

tail_recursive(Option)
Controls warnings of non-tail recursive predicate (and non-terminal) definitions. The lint check does not detect all cases of non-tail recursive predicate definitions, however. Also, definitions that make two or more recursive calls are not reported as usually they cannot be changed to be tail recursive. Possible option values are `warning` and `silent` (the usual default).

encodings(Option)
Controls the source file text encoding warnings. Possible option values are `warning` (the usual default) and `silent`.

general(Option)
Controls warnings that are not controlled by a specific flag. Possible option values are `warning` (the usual default) and `silent`.

Optional features compilation flags

complements(Option)
Allows objects to be compiled with support for complementing categories turned off in order to improve performance and security. Possible option values are `allow` (allow complementing categories to override local object predicate declarations and definitions), `restrict` (allow complementing categories to add predicate declarations and definitions to an object but not to override them), and `deny` (ignore complementing categories; the usual default). This option can be used on a per-object basis. Note that changing this option is of no consequence for objects already compiled and loaded.

dynamic_declarations(Option)
Allows objects to be compiled with support for dynamic declaration of new predicates turned off in order to improve performance and security. Possible option values are `allow` and `deny` (the usual default). This option can be used on a per-object basis. Note that changing this option is of no consequence for objects already compiled and loaded. This option is only checked when sending an *asserta/1* or *assertz/1* message to an object. Local asserting of new predicates is always allowed.

events(Option)
Allows message-sending calls to be compiled with or without *event-driven programming* support. Possible option values are `allow` and `deny` (the usual default). Objects (and categories) compiled with this option set to `deny` use optimized code for message-sending calls that do not trigger events. As such, this option can be used on a per-object (or per-category) basis. Note that changing this option is of no consequence for objects already compiled and loaded.

context_switching_calls(Option)
Allows context-switching calls (`(<<)/2`) to be either allowed or denied. Possible option values are `allow` and `deny`. The default flag value is `allow`. Note that changing this option is of no consequence for objects already compiled and loaded.

Backend Prolog compiler and loader flags

underscore_variables(Option)
Controls the interpretation of variables that start with an underscore (excluding the anonymous variable) that occur once in a term as either don't care variables or singleton variables. Possible option values are `dont_care` (the default for all supported backends) and `singletons`. Although a changeable flag, its value is backend dependent and thus expected to be set only in the backend adapter files.

prolog_compiler(Flags)
List of compiler flags for the generated Prolog files. The valid flags are specific to the used Prolog backend compiler. The usual default is the empty list. These flags are passed to the backend Prolog compiler built-in predicate that is responsible for compiling to disk a Prolog file. For Prolog compilers that don't provide separate predicates for compiling and loading a file, use instead the *prolog_loader* flag.

prolog_loader(Flags)
List of loader flags for the generated Prolog files. The valid flags are specific to the used Prolog backend compiler. The usual default is the empty list. These flags are passed to the backend Prolog compiler built-in predicate that is responsible for loading a (compiled) Prolog file.

Other flags

scratch_directory(Directory)
Sets the directory to be used to store the temporary files generated when compiling Logtalk source files. This directory can be specified using an atom or using *library notation*. The directory must always end with a slash. The default value is a sub-directory of the source files directory, either `'./lgt_tmp/'` or `'./lgt_tmp/'` (depending on the backend Prolog compiler and operating-system). Relative directories must always start with `'./'` due to the lack of a portable solution to check if a path is relative or absolute. The default value set on the *backend Prolog compiler* adapter file can be overridden by defining the `scratch_directory` library alias (see the *logtalk_library_path/2* predicate documentation for details).

report(Option)
Controls the default printing of messages. Possible option values are `on` (by usual default, print all messages that are not intercepted by the user), `warnings` (only print warning and error messages that are not intercepted by the user), and `off` (do not print any messages that are not intercepted by the user).

code_prefix(Character)
Enables the definition of prefix for all functors of Prolog code generated by the Logtalk compiler. The option value must be a single character atom. Its default value is `'$'`. Specifying a code prefix provides a way to solve possible conflicts between Logtalk compiled code and other Prolog code. In addition, some Prolog compilers automatically hide predicates whose functor starts with a specific prefix, such as the character `$`. Although this is not a read-only flag, it should only be changed at startup time and **before** loading any source files. When changing this flag (e.g., from a *settings file*), restart with the *clean* flag turned on to ensure that any compiled files using the old `code_prefix` value will be recompiled.

optimize(Option)
Controls the compiler optimizations. Possible option values are `on` (used by default for deployment) and `off` (used by default for development). Compiler optimizations include the use of static binding whenever possible, the removal of redundant calls to `true/0` from predicate clauses, the removal of redundant unifications when compiling grammar rules, and inlining of predicate definitions with a single clause that links to a local predicate, to a plain Prolog built-in (or foreign) predicate, or to a Prolog module predicate with the same arguments. Care should be taken when developing applications

with this flag turned on as changing and reloading a file may render *static binding* optimizations invalid for code defined in other loaded files. Turning on this flag automatically turns off the *debug* flag.

source_data(Option)
Defines how much information is retained when compiling a source file. Possible option values are `on` (the usual default for development) and `off`. With this flag set to `on`, Logtalk will keep the information represented using documenting directives plus source location data (including source file names and line numbers). This information can be retrieved using the *reflection API* and is useful for documenting, debugging, and integration with third-party development tools. This flag can be turned off in order to generate more compact code.

debug(Option)
Controls the compilation of source files in debug mode (the Logtalk default debugger can only be used with files compiled in this mode). Also controls, by default, printing of `debug>` and `debug(Topic)` messages. Possible option values are `on` and `off` (the usual default). Turning on this flag automatically turns off the *optimize* flag.

reload(Option)
Defines the reloading behavior for source files. Possible option values are `skip` (skip reloading of already loaded files; this value can be used to get similar functionality to the Prolog directive `ensure_loaded/1` but should be used only with fully debugged code), `changed` (the usual default; reload files only when they are changed since last loaded, provided that any explicit flags and the compilation mode are the same as before), and `always` (always reload files).

relative_to(Directory)
Defines a base directory for resolving relative source file paths. The default value is the directory of the source file being compiled.

hook(Object)
Allows the definition of an object (which can be the pseudo-object user) implementing the expanding built-in protocol. The hook object must be compiled and loaded when this option is used. It's also possible to specify a Prolog module instead of a Logtalk object, but the module must be pre-loaded, and its identifier must be different from any object identifier.

clean(Option)
Controls cleaning of the intermediate Prolog files generated when compiling Logtalk source files. Possible option values are `off` and `on` (the usual default). When turned on, intermediate files are deleted after loading, and all source files are recompiled disregarding any existing intermediate files. When turned off, the intermediate files are kept. This is useful when embedding applications, which requires collecting the intermediate code, and when working on large applications to avoid repeated recompilation of stable code. The flag must be turned on when changing compilation modes, changing flags such as *code_prefix*, or when turning on linter flags that are off by default without at the same time making changes to the application source files themselves, as any existing intermediate files would not be recompiled as necessary due to file timestamps not changing.

User-defined flags

Logtalk provides a *create_logtalk_flag/3* predicate that can be used for defining new flags.

1.14.12 Reloading source files

As a general rule, reloading source files should never occur in production code and should be handled with care in development code. Reloading a Logtalk source file usually requires reloading the intermediate Prolog file that is generated by the Logtalk compiler. The problem is that there is no standard behavior for reloading Prolog files. For static predicates, almost all Prolog compilers replace the old definitions with the new ones. However, for dynamic predicates, the behavior depends on the Prolog compiler. Most compilers replace the old definitions, but some of them simply append the new ones, which usually leads to trouble. See the compatibility notes for the backend Prolog compiler you intend to use for more information. There is an additional potential problem when using multi-threading programming. Reloading a threaded object does not recreate from scratch its old message queue, which may still be in use (e.g., threads may be waiting on it).

When using library entities and stable code, you can avoid reloading the corresponding source files (and, therefore, recompiling them) by setting the *reload* compiler flag to `skip`. For code under development, you can turn off the *clean* flag to avoid recompiling files that have not been modified since the last compilation (assuming that backend Prolog compiler that you are using supports retrieving file modification dates). You can disable deleting the intermediate files generated when compiling source files by changing the default flag value in your settings file, by using the corresponding compiler flag with the compiling and loading built-in predicates, or, for the remaining of a working session, by using the call:

```
| ?- set_logtalk_flag(clean, off).
```

Some caveats that you should be aware of. First, some warnings that might be produced when compiling a source file will not show up if the corresponding object file is up-to-date because the source file is not being (re)compiled. Second, if you are using several Prolog compilers with Logtalk, be sure to perform the first compilation of your source files with the `clean` flag turned off: the intermediate Prolog files generated by the Logtalk compiler may not be compatible across Prolog compilers or even for the same Prolog compiler across operating systems (e.g., due to the use of different character encodings or end-of-line characters).

1.14.13 Batch processing

When doing batch processing, you probably want to turn off the *report* flag to suppress all messages of type `banner`, `comment`, `comment(_)`, `warning`, and `warning(_)` that are normally printed. Note that error messages and messages providing information requested by the user will still be printed.

1.14.14 Optimizing performance

The default compiler flag settings are appropriate for the **development** but not necessarily for the **deployment** of applications. To minimize the generated code size, turn the *source_data* flag off. To optimize runtime performance, turn on the *optimize* flag. Your chosen backend Prolog compiler may also provide performance-related flags; check its documentation.

Pay special attention to file compilation/loading order. Whenever possible, compile and load your files by taking into account file dependencies. By default, the compiler will print a warning whenever a file references an entity that is not yet loaded. Solving these warnings is key for optimal performance by enabling *static binding* optimizations. For a clear picture of file dependencies, use the *diagrams* tool to generate a file dependency diagram for your application.

Minimize the use of dynamic predicates. Parametric objects can often be used in alternative. When dynamic predicates cannot be avoided, try to make them private. Declaring a dynamic predicate also as a private predicate allows the compiler to optimize local calls to the database methods (e.g., *assertz/1* and *retract/1*) that modify the predicate.

Sending a *message to self* implies *dynamic binding*, but there are often cases where *(::)/1* is misused to call an imported or inherited predicate that is never going to be redefined in a descendant. In these cases, a *super call*, *(^^)/1*, can be used instead with the benefit of often enabling static binding. Most of the guidelines for writing efficient Prolog code also apply to Logtalk code. In particular, define your predicates to take advantage of first-argument indexing. In the case of recursive predicates, define them as tail-recursive predicates whenever possible.

See the *section on performance* for a detailed discussion on Logtalk performance.

1.14.15 Portable applications

Logtalk is compatible with most modern standards-compliant Prolog compilers. However, this does not necessarily imply that your Logtalk applications will have the same level of portability. If possible, you should only use in your applications Logtalk built-in predicates and ISO Prolog-specified built-in predicates and arithmetic functions. If you need to use built-in predicates (or built-in arithmetic functions) that may not be available in other Prolog compilers, you should try to encapsulate the non-portable code in a small number of objects and provide a portable **interface** for that code through the use of Logtalk protocols. An example will be code that access operating-system specific features. The Logtalk compiler can warn you of the use of non-ISO-specified built-in predicates and arithmetic functions by using the *portability* compiler flag.

1.14.16 Conditional compilation

Logtalk supports conditional compilation within source files using the *if/1*, *elif/1*, *else/0*, and *endif/0* directives. This support is similar to the support found in several Prolog systems such as ECLiPSe, GNU Prolog, SICStus Prolog, SWI-Prolog, XSB, and YAP.

1.14.17 Avoiding common errors

Try to write objects and protocol documentation **before** writing any other code; if you are having trouble documenting a predicate, perhaps you need to go back to the design stage.

Try to avoid lengthy hierarchies. Composition is often a better choice over inheritance for defining new objects (Logtalk supports component-based programming through the use of *categories*). In addition, prototype-based hierarchies are semantically simpler than class-based hierarchies.

Dynamic predicates or dynamic entities are sometimes needed, but we should always try to minimize the use of non-logical features such as asserts and retracts.

Since each Logtalk entity is independently compiled, if an object inherits a dynamic or a meta-predicate predicate, then the respective directives must be repeated to ensure a correct compilation.

In general, Logtalk does not verify if a user predicate call/return arguments comply with the declared modes. On the other hand, Logtalk built-in predicates, built-in methods, and message-sending control structures are fully checked for calling mode errors.

Logtalk error handling strongly depends on the ISO compliance of the chosen Prolog compiler. For instance, the error terms that are generated by some Logtalk built-in predicates assume that the Prolog built-in predicates behave as defined in the ISO standard regarding error conditions. In particular, if your Prolog compiler does not support a `read_term/3` built-in predicate compliant with the ISO Prolog Standard definition, then

the current version of the Logtalk compiler may not be able to detect misspelled variables in your source code.

1.14.18 Coding style guidelines

It is suggested that all code between an entity opening and closing directives be indented by one tab stop. When defining entity code, both directives and predicates, Prolog coding style guidelines may be applied. All Logtalk source files, examples, and standard library entities use tabs (the recommended setting is a tab width equivalent to 4 spaces) for laying out code. Closely related entities can be defined in the same source file. However, for the best performance, is often necessary to have an entity per source file. Entities that might be useful in different contexts (such as library entities) are best defined in their own source files.

A detailed coding style guide is available at the Logtalk official website.

1.15 Printing messages and asking questions

Applications, components, and libraries often print all sorts of messages. These include banners, logging, debugging, and computation results messages. But also, in some cases, user interaction messages. However, the authors of applications, components, and libraries often cannot anticipate the context where their software will be used and thus decide which and when messages should be displayed, suppressed, or diverted. Consider the different components in a Logtalk application development and deployment. At the base level, you have the Logtalk compiler and runtime. The compiler writes messages related to e.g. compiling and loading files, compiling entities, and compilation warnings and errors. The runtime may write banner messages or throw execution errors that may result in printing human-level messages. The development environment can be console-based, or you may be using a GUI tool such as PDT. In the latter case, PDT needs to intercept the Logtalk compiler and runtime messages to present the relevant information using its GUI. Then you have all the other components in a typical application. For example, your own libraries and third-party libraries. The libraries may want to print messages on their own, e.g. banners, debugging information, or logging information. As you assemble all your application components, you want to have the final word on which messages are printed, where, and when. Uncontrolled message printing by libraries could potentially disrupt application flow, expose implementation details, spam the user with irrelevant details, or break user interfaces.

The solution is to decouple the calls to print a message from the actual printing of the output text. The same is true for calls to read user input. By decoupling the call to input some data from the actual read of the data, we can easily switch from, for example, a command-line interface to a GUI input dialog or even automate providing the data (e.g., when automating testing of the user interaction).

Logtalk provides a solution based on the *structured message printing mechanism* that was introduced by Quintus Prolog, where it was apparently implemented by Dave Bowen (thanks to Richard O'Keefe for the historical bits). This mechanism gives the programmer full control of message printing, allowing it to filter, rewrite, or redirect any message. Variations of this mechanism can also be found in some Prolog systems, including SICStus Prolog, SWI-Prolog, and YAP. Based on this mechanism, Logtalk introduces an extension that also allows abstracting asking a user for input. Both mechanisms are implemented by the logtalk built-in object and described in this section. The message printing mechanism is extensively used by the Logtalk compiler itself and by the developer tools. The question-asking mechanism is used e.g. in the *debugger* tool.

1.15.1 Printing messages

The main predicate for printing a message is *logtalk::print_message/3*. A simple example, using the Logtalk runtime, is:

```
| ?- logtalk::print_message(banner, core, banner).

Logtalk 3.23.0
Copyright (c) 1998-2018 Paulo Moura
yes
```

The first argument of the predicate is the kind of message that we want to print. In this case, we use `banner` to indicate that we are printing a product name and copyright banner. An extensive list of message kinds is supported by default:

`banner`
: banner messages (used e.g. when loading tools or main application components; can be suppressed by setting the *report* flag to `warnings` or `off`)

`help`
: messages printed in reply to the user asking for help (mostly for helping port existing Prolog code)

`information` and `information(Group)`
: messages usually printed in reply to a user's request for information

`silent` and `silent(Group)`
: not printed by default (but can be intercepted using the `message_hook/4` predicate)

`comment` and `comment(Group)`
: useful but usually not essential messages (can be suppressed by setting the *report* flag to `warnings` or `off`)

`warning` and `warning(Group)`
: warning messages (generated e.g. by the compiler; can be suppressed by turning off the *report* flag)

`error` and `error(Group)`
: error messages (generated e.g. by the compiler)

`debug`, `debug(Group)`
: debugging messages (by default, only printed when the *debug* flag is turned on; the `print_message/3` goals for these messages are suppressed by the compiler when the *optimize* flag is turned on)

`question`, `question(Group)`
: questions to a user

Using a compound term allows easy partitioning of messages of the same kind in different groups. Note that you can define your own alternative message kind identifiers for your own components, together with suitable definitions for their associated prefixes and output streams.

The second argument of `print_message/3` represents the *component* defining the message being printed. In this context, *component* is a generic term that can designate, e.g., a tool, a library, or some subsystem in a large application. In our example, the component name is `core`, identifying the Logtalk compiler/runtime. This argument was introduced to provide multiple namespaces for message terms and thus simplify programming-in-the-large by allowing easy filtering of all messages from a specific component and also avoiding conflicts when two components happen to define the same message term (e.g., `banner`). Users should choose and use a unique name for a component, which usually is the name of the component itself. For example, all messages from the *lgtunit* tool use `lgtunit` for the component argument. The compiler and runtime are interpreted as a single component designated as `core`.

The third argument of `print_message/3` is the message itself, represented by a term. In the above example, the message term is `banner`. Using a term to represent a message instead of a string with the message text itself has significant advantages. Notably, it allows using a compound term for easy parameterization of the message text and simplifies machine processing, localization of applications, and message interception. For example:

```
| ?- logtalk::print_message(comment, core, redefining_entity(object, foo)).

% Redefining object foo
yes
```

1.15.2 Message tokenization

The use of message terms requires a solution for generating the actual text of the messages. This is supported by defining grammar rules for the *logtalk::message_tokens//2* multifile non-terminal, which translates a message term, for a given component, to a list of tokens. For example:

```
:- multifile(logtalk::message_tokens//2).
:- dynamic(logtalk::message_tokens//2).

logtalk::message_tokens(redefining_entity(Type, Entity), core) -->
    ['Redefining ~w ~q'-[Type, Entity], nl].
```

The following tokens can be used when translating a message:

at_same_line
: Signals a following part to a multi-part message with no line break in between; this token is ignored when it's not the first in the list of tokens

tab(Expression)
: Evaluate the argument as an arithmetic expression and write the resulting number of spaces; this token is ignored when the number of spaces is not positive

nl
: Change line in the output stream

flush
: Flush the output stream (by calling the `flush_output/1` standard predicate)

Format-Arguments
: `Format` must be an atom and `Arguments` must be a list of format arguments (the token arguments are passed to a call to the `format/3` de facto standard predicate)

term(Term, Options)
: `Term` can be any term and `Options` must be a list of valid `write_term/3` output options (the token arguments are passed to a call to the `write_term/3` standard predicate)

ansi(Attributes, Format, Arguments)
: Taken from SWI-Prolog; by default, do nothing; can be used for styled output

begin(Kind, Var)
: Taken from SWI-Prolog; by default, do nothing; can be used together with `end(Var)` to wrap a sequence of message tokens

end(Var)
: Taken from SWI-Prolog; by default, do nothing

The `logtalk` object also defines public predicates for printing a list of tokens, for hooking into printing an individual token, and for setting default output streams and message prefixes. For example, the SWI-Prolog adapter file uses the print message token hook predicate to enable coloring of messages printed on a console.

1.15.3 Meta-messages

Defining tokenization rules for every message is not always necessary, however. Logtalk defines several *meta-messages* that are handy for simple cases and temporary messages only used during application development, notably debugging messages. See the *Debugging messages* section and the logtalk built-in object remarks section for details.

1.15.4 Defining message prefixes and output streams

The *logtalk::message_prefix_stream/4* hook predicate can be used to define a message line prefix and an output stream for printing messages of a given kind and component. For example:

```
:- multifile(logtalk::message_prefix_stream/4).
:- dynamic(logtalk::message_prefix_stream/4).

logtalk::message_prefix_stream(comment, my_app, '% ', user_output).
logtalk::message_prefix_stream(warning, my_app, '* ', user_error).
```

A single clause at most is expected per message kind and component pair. When this predicate is not defined for a given kind and component pair, the following defaults are used:

```
kind_prefix_stream(banner,          '',       user_output).
kind_prefix_stream(help,            '',       user_output).
kind_prefix_stream(question,        '',       user_output).
kind_prefix_stream(question(_),     '',       user_output).
kind_prefix_stream(information,     '% ',     user_output).
kind_prefix_stream(information(_), '% ',      user_output).
kind_prefix_stream(comment,         '% ',     user_output).
kind_prefix_stream(comment(_),      '% ',     user_output).
kind_prefix_stream(warning,         '*     ', user_error).
kind_prefix_stream(warning(_),      '*     ', user_error).
kind_prefix_stream(error,           '!     ', user_error).
kind_prefix_stream(error(_),        '!     ', user_error).
kind_prefix_stream(debug,           '>>> ',   user_error).
kind_prefix_stream(debug(_),        '>>> ',   user_error).
```

When the message kind is unknown, `information` is used instead.

1.15.5 Defining message prefixes and output files

Some applications require copying and saving messages without diverting them from their default stream. For simple cases, this can be accomplished by intercepting the messages using the *logtalk::message_hook/4* multifile hook predicate (see next section). In more complex cases, where messages are already intercepted for a different purpose, it can be tricky to use multiple definitions of the `message_hook/4` predicate as the order of the clauses of a multiple predicate cannot be assumed in general (for all `message_hook/4` predicate definitions to run, all but the last one to be called must fail). Using a single *master* definition is also not ideal as it would result in strong coupling instead of a clean separation of concerns.

The experimental *logtalk::message_prefix_file/6* hook predicate can be used to define a message line prefix and an output file for copying messages of a given kind and component pair. For example:

```
:- multifile(logtalk::message_prefix_file/6).
:- dynamic(logtalk::message_prefix_file/6).

logtalk::message_prefix_file(error,   app, '! ', 'log.txt', append, []).
logtalk::message_prefix_file(warning, app, '! ', 'log.txt', append, []).
```

A single clause at most is expected per message kind and component pair.

This predicate is called by default by the message printing mechanism. Definitions of the `message_hook/4` hook predicate are free to decide if the `logtalk::message_prefix_file/6` predicate should be called and acted upon.

1.15.6 Intercepting messages

Calls to the *logtalk::print_message/3* predicate can be intercepted by defining clauses for the *logtalk::message_hook/4* multifile hook predicate. This predicate can suppress, rewrite, and divert messages.

As a first example, assume that you want to make Logtalk startup less verbose by suppressing printing of the default compiler flag values. This can be easily accomplished by defining the following category in a settings file:

```
:- category(my_terse_logtalk_startup_settings).

    :- multifile(logtalk::message_hook/4).
    :- dynamic(logtalk::message_hook/4).

    logtalk::message_hook(default_flags, comment(settings), core, _).

:- end_category.
```

The printing message mechanism automatically calls the `message_hook/4` hook predicate. When this call succeeds, the mechanism assumes that the message has been successfully handled.

As another example, assume that you want to print all otherwise silent compiler messages:

```
:- category(my_verbose_logtalk_message_settings).

    :- multifile(logtalk::message_hook/4).
    :- dynamic(logtalk::message_hook/4).

    logtalk::message_hook(_Message, silent, core, Tokens) :-
        logtalk::message_prefix_stream(comment, core, Prefix, Stream),
        logtalk::print_message_tokens(Stream, Prefix, Tokens).

    logtalk::message_hook(_Message, silent(Key), core, Tokens) :-
        logtalk::message_prefix_stream(comment(Key), core, Prefix, Stream),
        logtalk::print_message_tokens(Stream, Prefix, Tokens).

:- end_category.
```

1.15.7 Asking questions

Logtalk *structured question-asking* mechanism complements the message printing mechanism. It provides an abstraction for the common task of asking a user a question and reading back its reply. By default, this mechanism writes the question, writes a prompt, and reads the answer using the current user input and output streams but allows all steps to be intercepted, filtered, rewritten, and redirected. Two typical examples are using a GUI dialog for asking questions and automatically providing answers to specific questions.

The question-asking mechanism works in tandem with the message printing mechanism, using it to print the question text and a prompt. It provides an asking predicate and a hook predicate, both declared and defined in the `logtalk` built-in object. The asking predicate, *logtalk::ask_question/5*, is used for asking a question and reading the answer. Assume that we defined the following message tokenization and question prompt and stream:

```
:- category(hitchhikers_guide_to_the_galaxy).

    :- multifile(logtalk::message_tokens//2).
    :- dynamic(logtalk::message_tokens//2).

    % abstract the question text using the atom ultimate_question;
    % the second argument, hitchhikers, is the application component
    logtalk::message_tokens(ultimate_question, hitchhikers) -->
        ['The answer to the ultimate question of life, the universe and everything is?'-[],␣
↪nl].

   :- multifile(logtalk::question_prompt_stream/4).
   :- dynamic(logtalk::question_prompt_stream/4).

   % the prompt is specified here instead of being part of the question text
   % as it will be repeated if the answer doesn't satisfy the question closure
   logtalk::question_prompt_stream(question, hitchhikers, '> ', user_input).

:- end_category.
```

After compiling and loading this category, we can now ask the ultimate question:

```
| ?- logtalk::ask_question(question, hitchhikers, ultimate_question, '=='(42), N).

The answer to the ultimate question of life, the universe and everything is?
> 42.

N = 42
yes
```

Note that the fourth argument, `'=='(42)` in our example, is a *closure* that is used to check the answers provided by the user. The question is repeated until the goal constructed by extending the closure with the user answer succeeds. For example:

```
| ?- logtalk::ask_question(question, hitchhikers, ultimate_question, '=='(42), N).
The answer to the ultimate question of life, the universe and everything is?
> icecream.
> tea.
> 42.
```

(continues on next page)

(continued from previous page)

```
N = 42
yes
```

Practical usage examples of this mechanism can be found, e.g., in the `debugger` tool where it's used to abstract the user interaction when tracing a goal execution in debug mode.

1.15.8 Intercepting questions

Calls to the *logtalk::ask_question/5* predicate can be intercepted by defining clauses for the *logtalk::question_hook/6* multifile hook predicate. This predicate can suppress, rewrite, and divert questions. For example, assume that we want to automate testing and thus cannot rely on someone manually providing answers:

```
:- category(hitchhikers_fixed_answers).

    :- multifile(logtalk::question_hook/6).
    :- dynamic(logtalk::question_hook/6).

    logtalk::question_hook(ultimate_question, question, hitchhikers, _, _, 42).

:- end_category.
```

After compiling and loading this category, trying the question again will now skip asking the user:

```
| ?- logtalk::ask_question(question, hitchhikers, ultimate_question, '=='(42), N).

N = 42
yes
```

In a practical case, the fixed answer would be used for follow-up goals being tested. The question-answer read loop (which calls the question check closure) is not used when a fixed answer is provided using the `logtalk::question_hook/6` predicate thus preventing the creation of endless loops. For example, the following query succeeds:

```
| ?- logtalk::ask_question(question, hitchhikers, ultimate_question, '=='(41), N).

N = 42
yes
```

Note that the `logtalk::question_hook/6` predicate takes as argument the closure specified in the `logtalk::ask_question/5` call, allowing a fixed answer to be checked before being returned.

1.15.9 Multi-threading applications

When writing multi-threading applications, user-defined predicates calling methods such as `print_message/3` or `ask_question/5` may need to be declared synchronized in order to avoid race conditions.

1.16 Term and goal expansion

Logtalk supports a *term and goal expansion mechanism* that can be used to define source-to-source transformations. Two common uses are the definition of language extensions and domain-specific languages.

Logtalk improves upon the term-expansion mechanism found on some Prolog systems by providing the user with fine-grained control on *if*, *when*, and *how* expansions are applied. It allows declaring in a source file itself which expansions, if any, will be used when compiling it. It allows declaring which expansions will be used when compiling a file using compile and loading predicate options. It also allows defining a default expansion for all source files. It defines a concept of *hook objects* that can be used as building blocks to create custom and reusable *expansion workflows* with explicit and well-defined semantics. It prevents the simple act of loading expansion rules affecting subsequent compilation of files. It prevents conflicts between groups of expansion rules of different origins. It avoids a set of buggy expansion rules from breaking other sets of expansion rules.

1.16.1 Defining expansions

Term and goal expansions are defined using, respectively, the predicates *term_expansion/2* and *goal_expansion/2*, which are declared in the expanding built-in protocol. Note that, unlike Prolog systems also providing these two predicates, they are **not** declared as *multifile predicates* in the protocol. This design decision is key for giving the programmer full control of the expansion process and preventing the problems inflicting most of the Prolog systems that provide a term-expansion mechanism.

An example of an object defining expansion rules:

```
:- object(an_object,
    implements(expanding)).

    term_expansion(ping, pong).
    term_expansion(
        colors,
        [white, yellow, blue, green, read, black]
    ).

    goal_expansion(a, b).
    goal_expansion(b, c).
    goal_expansion(X is Expression, true) :-
        catch(X is Expression, _, fail).

:- end_object.
```

These predicates can be explicitly called using the *expand_term/2* and *expand_goal/2* built-in methods or called automatically by the compiler when compiling a source file (see the section below on *hook objects*).

In the case of source files referenced in *include/1* directives, expansions are only applied automatically when the directives are found in source files, not when used as arguments in the *create_object/4*, *create_protocol/3*, and *create_category/4*, predicates. This restriction prevents inconsistent results when, for example, an expansion is defined for a predicate with clauses found in both an included file and as argument in a call to the `create_object/4` predicate.

Clauses for the `term_expansion/2` predicate are called until one of them succeeds. The returned expansion can be a single term or a list of terms (including the empty list). For example:

```
| ?- an_object::expand_term(ping, Term).

Term = pong
yes

| ?- an_object::expand_term(colors, Colors).

Colors = [white, yellow, blue, green, read, black]
yes
```

When no `term_expansion/2` clause applies, the same term that we are trying to expand is returned:

```
| ?- an_object::expand_term(sounds, Sounds).

Sounds = sounds
yes
```

Clauses for the `goal_expansion/2` predicate are recursively called on the expanded goal until a fixed point is reached. For example:

```
| ?- an_object::expand_goal(a, Goal).

Goal = c
yes

| ?- an_object::expand_goal(X is 3+2*5, Goal).

X = 13,
Goal = true
yes
```

When no `goal_expansion/2` clause applies, the same goal that we are trying to expand is returned:

```
| ?- an_object::expand_goal(3 =:= 5, Goal).

Goal = (3=:=5)
yes
```

The goal-expansion mechanism prevents an infinite loop when expanding a goal by checking that a goal to be expanded was not the result from a previous expansion of the same goal. For example, consider the following object:

```
:- object(fixed_point,
    implements(expanding)).

    goal_expansion(a, b).
    goal_expansion(b, c).
    goal_expansion(c, (a -> b; c)).

:- end_object.
```

The expansion of the goal `a` results in the goal `(a -> b; c)` with no attempt to further expand the `a`, `b`, and `c` goals as they have already been expanded.

Goal-expansion applies to goal arguments of control constructs, meta-arguments in built-in or `user` defined

meta-predicates, meta-arguments in local user-defined meta-predicates, meta-arguments in meta-predicate messages when static binding is possible, and `initialization/1`, `if/1`, and `elif/1` directives.

1.16.2 Expanding grammar rules

A common term expansion is the translation of grammar rules into predicate clauses. This transformation is performed automatically by the compiler when a source file entity defines grammar rules. It can also be done explicitly by calling the `expand_term/2` built-in method. For example:

```
| ?- logtalk::expand_term((a --> b, c), Clause).

Clause = (a(A,B) :- b(A,C), c(C,B))
yes
```

Note that the default translation of grammar rules can be overridden by defining clauses for the *term_expansion/2* predicate.

1.16.3 Bypassing expansions

Terms and goals wrapped by the *{}/1* control construct are not expanded. For example:

```
| ?- an_object::expand_term({ping}, Term).

Term = {ping}
yes

| ?- an_object::expand_goal({a}, Goal).

Goal = {a}
yes
```

This also applies to source file terms and source file goals when using hook objects (discussed next).

1.16.4 Hook objects

Term and goal expansion of a source file during its compilation is performed by using *hook objects*. A hook object is simply an object implementing the expanding built-in protocol and defining clauses for the term and goal expansion hook predicates. Hook objects must be compiled and loaded prior to being used to expand a source file.

To compile a source file using a hook object, we can use the *hook* compiler flag in the second argument of the *logtalk_compile/2* and *logtalk_load/2* built-in predicates. For example:

```
| ?- logtalk_load(source_file, [hook(hook_object)]).
...
```

In alternative, we can use a *set_logtalk_flag/2* directive in the source file itself. For example:

```
:- set_logtalk_flag(hook, hook_object).
```

To use multiple hook objects in the same source file, simply write each directive before the block of code that it should handle. For example:

```
:- object(h1,
    implements(expanding)).

    term_expansion((:- public(a/0)), (:- public(b/0))).
    term_expansion(a, b).

:- end_object.
```

```
:- object(h2,
    implements(expanding)).

    term_expansion((:- public(a/0)), (:- public(c/0))).
    term_expansion(a, c).

:- end_object.
```

```
:- set_logtalk_flag(hook, h1).

:- object(s1).

    :- public(a/0).
    a.

:- end_object.


:- set_logtalk_flag(hook, h2).

:- object(s2).

    :- public(a/0).
    a.

:- end_object.
```

```
| ?- {h1, h2, s}.
...

| ?- s1::b.
yes

| ?- s2::c.
yes
```

It is also possible to define a default hook object by defining a global value for the `hook` flag by calling the *set_logtalk_flag/2* predicate. For example:

```
| ?- set_logtalk_flag(hook, hook_object).

yes
```

Note that, due to the `set_logtalk_flag/2` directive being local to a source file, using it to specify a hook object will override any defined default hook object or any hook object specified as a `logtalk_compile/2` or

`logtalk_load/2` predicate compiler option for compiling or loading the source file.

> **Note**
>
> Clauses for the `term_expansion/2` and `goal_expansion/2` predicates defined within an object or a category are never used in the compilation of the object or the category itself.

1.16.5 Virtual source file terms and loading context

When using a hook object to expand the terms of a source file, two virtual file terms are generated: `begin_of_file` and `end_of_file`. These terms allow the user to define term-expansions before and after the actual source file terms.

Logtalk also provides a *logtalk_load_context/2* built-in predicate that can be used to access the compilation/loading context when performing expansions. The *logtalk* built-in object also provides a set of predicates that can be useful, notably when adding Logtalk support for language extensions originally developed for Prolog.

As an example of using the virtual terms and the `logtalk_load_context/2` predicate, assume that you want to convert plain Prolog files to Logtalk by wrapping the Prolog code in each file using an object (named after the file) that implements a given protocol. This could be accomplished by defining the following hook object:

```
:- object(wrapper(_Protocol_),
    implements(expanding)).

    term_expansion(begin_of_file, (:- object(Name,implements(_Protocol_)))) :-
        logtalk_load_context(file, File),
        os::decompose_file_name(File,_ , Name, _).

    term_expansion(end_of_file, (:- end_object)).

:- end_object.
```

Assuming, e.g., `my_car.pl` and `lease_car.pl` files to be wrapped and a `car_protocol` protocol, we could then load them using:

```
| ?- logtalk_load(
        ['my_car.pl', 'lease_car.pl'],
        [hook(wrapper(car_protocol))]
    ).

yes
```

> **Note**
>
> When a source file also contains plain Prolog directives and predicates, these are term-expanded but not goal-expanded (with the exception of the `initialization/1`, `if/1`, and `elif/1` directives, where the goal argument is expanded to improve code portability across backends).

1.16.6 Default compiler expansion workflow

When *compiling a source file*, the compiler will first try, by default, the source file-specific hook object specified using a local `set_logtalk_flag/2` directive, if defined. If that expansion fails, it tries the hook object specified using the `hook/1` compiler option in the `logtalk_compile/2` or `logtalk_load/2` goal that compiles or loads the file, if defined. If that expansion fails, it tries the default hook object, if defined. If that expansion also fails, the compiler tries the Prolog dialect-specific expansion rules found in the *adapter file* (which are used to support non-standard Prolog features).

1.16.7 User defined expansion workflows

Sometimes we have multiple hook objects that we need to combine and use in the compilation of a source file. Logtalk includes a *hook_flows* library that supports two basic expansion workflows: a pipeline of hook objects, where the expansion results from a hook object are fed to the next hook object in the pipeline, and a set of hook objects, where expansions are tried until one of them succeeds. These workflows are implemented as parametric objects, allowing combining them to implement more sophisticated expansion workflows. There is also a *hook_objects* library that provides convenient hook objects for defining custom expansion workflows. This library includes a hook object that can be used to restore the default expansion workflow used by the compiler.

For example, assuming that you want to apply the Prolog backend-specific expansion rules defined in its adapter file, using the backend_adapter_hook library object, passing the resulting terms to your own expansion when compiling a source file, we could use the goal:

```
| ?- logtalk_load(
        source,
        [hook(hook_pipeline([backend_adapter_hook, my_expansion]))]
    ).
```

As a second example, we can prevent expansion of a source file using the library object identity_hook by adding as the first term in a source file the directive:

```
:- set_logtalk_flag(hook, identity_hook).
```

The file will be compiled as-is as any hook object (specified as a compiler option or as a default hook object) and any backend adapter expansion rules are overridden by the directive.

1.16.8 Using Prolog defined expansions

In order to use clauses for the `term_expansion/2` and `goal_expansion/2` predicates defined in plain Prolog, simply specify the pseudo-object `user` as the hook object when compiling source files. When using *backend Prolog compilers* that support a module system, it can also be specified a module containing clauses for the expanding predicates as long as the module name doesn't coincide with an object name. When defining a custom workflow, the library object prolog_module_hook/1 can be used as a workflow step. For example, assuming a module `functions` defining expansion rules that we want to use:

```
| ?- logtalk_load(
        source,
        [hook(hook_set([prolog_module_hook(functions), my_expansion]))]
    ).
```

But note that Prolog module libraries may provide definitions of the expansion predicates that are not compatible with the Logtalk compiler. In particular, when setting the hook object to `user`, be aware of any Prolog

library that is loaded, possibly by default or implicitly by the Prolog system, that may be contributing definitions of the expansion predicates. It is usually safer to define a specific hook object for combining multiple expansions in a fully controlled way.

> **Note**
>
> The `user` object declares `term_expansion/2` and `goal_expansion/2` as multifile and dynamic predicates. This helps in avoiding predicate existence errors when compiling source files with the `hook` flag set to `user` as these predicates are only natively declared by some of the supported backend Prolog compilers.

1.16.9 Debugging expansions

The `term_expansion/2` and `goal_expansion/2` predicates can be *debugged* like any other object predicates. Note that expansions can often be manually tested by sending *expand_term/2* and *expand_goal/2* messages to a hook object with the term or goal whose expansion you want to check as argument. An alternative to the debugging tools is to use a *monitor* for the runtime messages that call the predicates. For example, assume a `expansions_debug.lgt` file with the contents:

```
:- initialization(
    define_events(after, edcg, _, _, expansions_debug)
).


:- object(expansions_debug,
    implements(monitoring)).

    after(edcg, term_expansion(T,E), _) :-
        writeq(term_expansion(T,E)), nl.

:- end_object.
```

We can use this monitor to help debug the expansion rules of the *edcg* library when applied to the `edcgs` example using the queries:

```
| ?- {expansions_debug}.
...

| ?- set_logtalk_flag(events, allow).
yes

| ?- {edcgs(loader)}.
...
term_expansion(begin_of_file,begin_of_file)
term_expansion((:-object(gemini)),[(:-object(gemini)),(:-op(1200,xfx,-->>))])
term_expansion(acc_info(castor,A,B,C,true),[])
term_expansion(pass_info(pollux),[])
term_expansion(pred_info(p,1,[castor,pollux]),[])
term_expansion(pred_info(q,1,[castor,pollux]),[])
term_expansion(pred_info(r,1,[castor,pollux]),[])
term_expansion((p(A)-->>B is A+1,q(B),r(B)),(p(A,C,D,E):-B is A+1,q(B,C,F,E),r(B,F,D,E)))
term_expansion((q(A)-->>[]),(q(A,B,B,C):-true))
term_expansion((r(A)-->>[]),(r(A,B,B,C):-true))
```

(continues on next page)

(continued from previous page)

```
term_expansion(end_of_file,end_of_file)
...
```

This solution does not require compiling the `edcg` hook object in debug mode or access to its source code (e.g., to modify its expansion rules to emit debug messages). We could also simply use the `user` pseudo-object as the monitor object:

```
| ?- assertz((
        after(_, term_expansion(T,E), _) :-
           writeq(term_expansion(T,E)), nl
    )).
yes

| ?- define_events(after, edcg, _, Sender, user).
yes
```

Another alternative is to use a pipeline of hook objects with the library `hook_pipeline/1` and `write_to_stream_hook` objects to write the expansion results to a file. For example, using the `unique.lgt` test file from the `edcgs` library directory:

```
| ?- {hook_flows(loader), hook_objects(loader)}.
...

| ?- open('unique_expanded.lgt', write, Stream),
    logtalk_compile(
        unique,
        [hook(hook_pipeline([edcg,write_to_stream_hook(Stream,[quoted(true)])]))]
    ),
    close(Stream).
...
```

The generated `unique_expanded.lgt` file will contain the clauses resulting from the expansion of the EDCG rules found in the `unique.lgt` file by the `edcg` hook object expansion.

1.17 Documenting

Assuming that the *source_data* flag is turned on, the compiler saves all relevant documenting information collected when compiling a source file. The provided *lgtdoc* tool can access this information by using the *reflection* support and generate a documentation file for each compiled entity (object, protocol, or category) in XML format. Contents of the XML file include the entity name, type, and compilation mode (static or dynamic), the entity relations with other entities, and a description of any declared predicates (name, compilation mode, scope, ...). The XML documentation files can be enriched with arbitrary user-defined information, either about an entity or about its predicates, by using documentation directives. The `lgtdoc` tool includes POSIX and Windows scripts for converting the XML documentation files to several final formats (such as HTML and PDF).

Logtalk supports two documentation directives for providing arbitrary user-defined information about an entity or a predicate. These two directives complement other directives that also provide important documentation information, such as the *mode/2* and *meta_predicate/1* directives.

1.17.1 Entity documenting directives

Arbitrary user-defined entity information can be represented using the *info/1* directive:

```
:- info([
    Key1 is Value1,
    Key2 is Value2,
    ...
]).
```

In this pattern, keys should be atoms and values should be bound terms. The following keys are predefined and may be processed specially by Logtalk tools:

comment
: Comment describing the entity purpose (an atom). End the comment with a period (full stop). As a style guideline, don't use overly long comments. If you need to provide additional details, use the `fails_if` and `remarks` keys.

author
: Entity author(s) (an atom or a compound term `{entity}` where `entity` is the name of an XML entity in a user-defined `custom.ent` file).

version
: Version number (a `Major:Minor:Patch` compound term) Following the Semantic Versioning guidelines is strongly advised.

date
: Date of last modification in ISO 8601 standard format (`Year-Month-Day` where `Year`, `Month`, and `Day` are integers).

parameters
: Parameter names and descriptions for parametric entities (a list of `Name-Description` pairs where both names and descriptions are atoms). End the `Description` with a period (full stop).

parnames
: Parameter names for parametric entities (a list of atoms; a simpler version of the previous key, used when parameter descriptions are deemed unnecessary).

copyright
: Copyright notice for the entity source code (an atom or a compound term `{entity}` where `entity` is the name of an XML entity defined in a user-defined `custom.ent` file).

license
: License terms for the entity source code; usually, just the license name (an atom or a compound term `{entity}` where `entity` is the name of an XML entity in a user-defined `custom.ent` file). License names should, whenever possible, be a license identifier as specified in the SPDX standard.

remarks
: List of general remarks about the entity using `Topic-Text` pairs where both the topic and the text must be atoms. End the `Text` with a period (full stop).

see_also
: List of related entities (using the entity identifiers, which can be atoms or compound terms).

For example:

```
:- info([
    version is 2:1:0,
    author is 'Paulo Moura',
```

(continues on next page)

(continued from previous page)

```
    date is 2000-11-20,
    comment is 'Building representation.',
    diagram is 'UML Class Diagram #312'
]).
```

Use only the keywords that make sense for your application, and remember that you are free to invent your own keywords. All key-value pairs can be retrieved programmatically using the *reflection API* and are visible to the *lgtdoc* tool (which includes them in the generated documentation).

1.17.2 Predicate documenting directives

Arbitrary user-defined predicate information can be represented using the *info/2* directive:

```
:- info(Name/Arity, [
    Key1 is Value1,
    Key2 is Value2,
    ...
]).
```

The first argument can also a grammar rule non-terminal indicator, `Name//Arity`. Keys should be atoms. Values should be bound terms. The following keys are predefined and may be processed specially by Logtalk tools:

comment
: Comment describing the predicate (or non-terminal) purpose (an atom). End the comment with a period (full stop). As a style guideline, don't use overly long comments. If you need to provide additional details, use the `remarks` key.

fails_if
: Comment describing failing conditions for the predicate. As a style guideline, don't use overly long comments. If you need to provide additional details, use the `remarks` key.

arguments
: Names and descriptions of predicate arguments for pretty print output (a list of `Name-Description` pairs where both names and descriptions are atoms). End the `Description` with a period (full stop).

argnames
: Names of predicate arguments for pretty print output (a list of atoms; a simpler version of the previous key, used when argument descriptions are deemed unnecessary).

allocation
: Objects where we should define the predicate. Some possible values are `container`, `descendants`, `instances`, `classes`, `subclasses`, and `any`.

redefinition
: Describes if a predicate is expected to be redefined and, if so, in what way. Some possible values are `never`, `free`, `specialize`, `call_super_first`, `call_super_last`.

exceptions
: List of possible exceptions thrown by the predicate using `Description-Exception` pairs. The description must be an atom. The exception term must be a ground term.

examples
: List of typical predicate call examples using the format `Description-Goal-Bindings`. The description must be an atom with the goal term sharing variables with the bindings. The variable bindings term uses the format `{Variable = Term, ...}`. When there are no variable bindings, the success or failure

of the predicate call should be represented by the terms `{true}` or `{false}`, respectively (you can also use in alternative the terms `{yes}` or `{no}`).

remarks
List of general remarks about the predicate using `Topic-Text` pairs where both the topic and the text must be atoms. End the `Text` with a period (full stop).

since
Version that added the predicate (`Major:Minor:Patch`).

see_also
List of related predicates and non-terminals (using the predicate and non-terminal indicators).

For example:

```
:- info(color/1, [
    comment is 'Table of defined colors.',
    argnames is ['Color'],
    constraint is 'Up to four visible colors allowed.',
    examples is [
       'Check that the color blue is defined' - color(blue) - {true}
    ]
]).
```

As with the `info/1` directive, use only the keywords that make sense for your application and remember that you are free to invent your own keywords. All key-value pairs can also be retrieved programmatically using the *reflection API* and are visible to the *lgtdoc* tool (which includes them in the generated documentation).

1.17.3 Describing predicates

The value of the `comment` key, possibly extended with the `remarks` key, should describe a predicate purpose and, when applicable, the circumstances under which a call may fail. Descriptions should be consistent across library and application APIs. Some guidelines:

1. When starting the description with a verb, use the *third-person singular simple present form*. For example, write `'Runs ...'`, `'Calls ...'`, `'Compares ...'`, `'Parses ...'`, `'Generates ...'`, `'Converts ...'`, `'Creates ...'`, `'Maps ...'`, `'Merges ...'`, `'Finds ...'`, etc.

2. Predicates that are pure logical relations often have descriptions starting with `'True iff ...'` or `'True if ...'`.

3. Predicates with multiple solutions often have descriptions starting with `'Enumerates, by backtracking, all ...'` or `'Enumerates, by backtracking, the ...'`.

4. Predicate call failure conditions often have descriptions with one or more sentences starting with `'Fails when ...'` or `'Fails if ...'`.

If you're not sure how best to describe a predicate, look for examples in the Logtalk libraries and developer tools APIs documentation.

1.17.4 Documenting predicate exceptions

As described above, the `info/2` predicate directive supports an `exceptions` key that allows us to list all exceptions that may occur when calling a predicate. For example:

```
:- info(check_option/1, [
    comment is 'Succeeds if the option is valid. Throws an error otherwise.',
    argnames is ['Option'],
    exceptions is [
        '``Option`` is a variable' - instantiation_error,
        '``Option`` is neither a variable nor a compound term' - type_error(compound, 'Option
↪'),
        '``Option`` is a compound term but not a valid option' - domain_error(option, 'Option
↪')
    ]
]).
```

When possible, only standard exceptions should be used. See e.g. the *error handling methods* section for a full list. The argument names should be the same as those provided in the `arguments` or `argnames` keys. Exceptions are usually listed starting with instantiation and uninstantiation errors, followed by type errors, and then domain errors. These may then be followed by permission, existence, evaluation, representation, or resource errors.

For each exception, use of *controlled language* as found, e.g., in the ISO Prolog Core standard and this Handbook is advised. Some examples:

Instantiation error when one or more arguments cannot be a variable
: `Argument` is a variable

: `Argument1` and `Argument2` are variables

Instantiation error when a closed list with bound elements is required
: `Argument` is a partial list or a list with an element `Element` which is a variable

Uninstantiation error when an argument is not a variable
: `Argument` is not a variable

Type error when an argument is not a variable but also not of the expected type
: `Argument` is neither a variable nor a TYPE

: `Argument` is neither a partial list nor a list

Type error when an element of a list is not a variable but is not of the expected type
: An element `Element` of the `Argument` list is neither a variable nor a TYPE

Domain error when an argument is of the correct type but not in the expected domain
: `Argument` is a TYPE but not a valid DOMAIN

: `Argument` is an integer that is less than zero

Domain error when an element of a list is of the correct type but not in the expected domain
: An element `Element` of the `Argument` list is a TYPE but not a valid DOMAIN

Existence error when an entity of a given kind does not exist
: The KIND `Argument` does not exist

Other classes of errors have a less rigid style. In case of doubt, look for examples in this Handbook, in the APIs documentation, and in standard documents.

1.17.5 Processing and viewing documenting files

The *lgtdoc* tool generates an XML documenting file per entity. It can also generate library, directory, entity, and predicate indexes when documenting libraries and directories. For example, assuming the default file-name extensions, a `trace` object and a `sort(_)` parametric object will result in `trace_0.xml` and `sort_1.xml` XML files.

Each entity XML file contains references to two other files, an XML specification file and a XSLT stylesheet file. The XML specification file can be either a DTD file (`logtalk_entity.dtd`) or an XML Scheme file (`logtalk_entity.xsd`). The XSLT stylesheet file is responsible for converting the XML files to some desired format such as HTML or PDF. The default names for the XML specification file and the XSL stylesheet file are defined by the *lgtdoc* tool but can be overridden by passing a list of options to the tool predicates. The `lgtdoc/xml` sub-directory in the Logtalk installation directory contains the XML specification files described above, along with several sample XSL stylesheet files and sample scripts for converting XML documenting files to several formats (e.g., reStructuredText, Markdown, HTML, and PDF). For example, assume that you want to generate the API documentation for the `types` library:

```
| ?- {types(loader)}.
....

| ?- {lgtdoc(loader)}.
....

| ?- lgtdoc::library(types).
...
```

The above queries will result in the creation of a `xml_docs` in your current directory by default. Assuming that we want to generate Sphinx-based documentation and that we are using a POSIX operating-system, the next steps would be:

```
$ cd xml_docs
$ lgt2rst -s -m
```

The `lgt2rst` script will ask a few questions (project name, author, version, ...). After its completion, the generated HTML files will be found in the `_build/html` directory by default:

```
$ open _build/html/index.html
```

For Windows operating-systems, PowerShell scripts are available. For example, assuming that we want to generate HTML documentation, we could run in a PowerShell window:

```
PS > cd xml_docs
PS > lgt2html.ps1 -p saxon
```

After completion, the generated HTML files will be found in the `xml_docs` directory by default.

See the `NOTES` file in the tool directory for details, specially on the XSLT processor dependencies. You may use the supplied sample files as a starting point for generating the documentation of your Logtalk applications.

The Logtalk DTD file, `logtalk_entity.dtd`, contains a reference to a user-customizable file, `custom.ent`, which declares XML entities for source code author names, license terms, and copyright strings. After editing the `custom.ent` file to reflect your personal data, you may use the XML entities on `info/1` documenting directives. For example, assuming that the XML entities are named *author*, *license*, and *copyright* we may write:

```
:- info([
    version is 1:1:0,
    author is {author},
    license is {license},
    copyright is {copyright}
]).
```

The entity references are replaced by the value of the corresponding XML entity when the XML documenting files are processed (**not** when they are generated; this notation is just a shortcut to take advantage of XML entities).

The *lgtdoc* tool supports a set of options that can be used to control the generation of the XML documentation files. See the tool documentation for details. There is also a *doclet* tool that allows automating the steps required to generate the documentation for an application.

1.17.6 Inline formatting in comments text

Inline formatting in comments text can be accomplished by using Markdown or reStructuredText syntax and converting XML documenting files to Markdown or reStructuredText files (and these, if required, to e.g. HTML, ePub, or PDF formats). Note that Markdown and reStructuredText common syntax elements are enough for most API documentation:

```
Mark *italic text* with one asterisk.
Mark **bold text** with two asterisks.
Mark ``monospaced text`` with two backquotes.
```

Rendering this block as markup gives:

> Mark *italic text* with one asterisk. Mark **bold text** with two asterisks. Mark `monospaced text` with two backquotes.

As single backquotes have different purposes in Markdown (monospaced text) and reStructuredText (domain- or application-dependent meaning), never use them. This also avoids doubts if there's an inline formatting typo in text meant to be rendered as monospaced text (usually inline code fragments).

1.17.7 Diagrams

The *diagrams* tool supports a wide range of diagrams that can also help in documenting an application. The generated diagrams can include URL links to both source code and API documentation. They can also be linked, connecting, for example, high level diagrams to detail diagrams. These features allow diagrams to be an effective solution for navigating and understanding the structure and implementation of an application. This tool uses the same *reflection API* as the `lgtdoc` tool and thus has access to the same source data. See the tool documentation for details.

1.18 Debugging

The Logtalk distribution includes a command-line *debugger* tool implemented as a Logtalk application using the debugging API. It can be loaded at the top-level interpreter by typing:

```
| ?- logtalk_load(debugger(loader)).
```

It can also be loaded automatically at startup time by using a *settings file*.

The *debugger* tool includes the debugging features found in traditional Prolog debuggers. There are some differences, however, between the usual implementation of Prolog debuggers and the current implementation of the Logtalk debugger that you should be aware of. First, unlike most Prolog debuggers, the Logtalk debugger is not a built-in feature but a regular Logtalk application using documented debugging hook predicates. This translates to a different, although similar, set of debugging features when compared with some of the more sophisticated Prolog debuggers. Second, debugging is only possible for entities compiled in debug mode. When compiling an entity in debug mode, Logtalk decorates clauses with source information to allow tracing of the goal execution. Third, the tool provides several types of breakpoints (for pausing and interacting with the debugger) and also log points, while most Prolog systems are limited to traditional predicate spy points.

1.18.1 Compiling source files in debug mode

Compilation of source files in debug mode is controlled by the *debug* compiler flag. The default value for this flag, usually `off`, is defined in the adapter files. Its default value may be changed globally at runtime by calling:

```
| ?- set_logtalk_flag(debug, on).
```

Implicitly, this goal also turns off the `optimize` flag. In alternative, if we want to compile only some source files in debug mode, we may instead write:

```
| ?- logtalk_load([file1, file2, ...], [debug(on)]).
```

The *logtalk_make/1* built-in predicate can also be used to recompile all loaded files (that were compiled without using explicit values for the *debug* and *optimize* compiler flags in a `logtalk_load/2` call or in a *loader file*, if used) in debug mode:

```
| ?- logtalk_make(debug).
```

With most *backend Prolog compilers*, the `{+d}` top-level shortcut can also be used. After debugging, the files can be recompiled in normal or optimized mode using, respectively, the `{+n}` or `{+o}` top-level shortcuts.

> **Warning**
>
> The *clean* compiler flag should be turned on whenever the *debug* flag is turned on at runtime. This is necessary because debug code would not be generated for files previously compiled in normal or optimized mode if there are no changes to the source files.

After loading the debugger, we may check (or enumerate by backtracking), all loaded entities compiled in debug mode as follows:

```
| ?- debugger::debugging(Entity).
```

To compile only a specific entity in debug mode, use the *set_logtalk_flag/2* directive inside the entity. To compile all entities in a source file in debug mode, use the *set_logtalk_flag/2* directive at the beginning of the file.

1.18.2 Procedure box model

Logtalk uses a *procedure box model* similar to those found on most Prolog systems. The traditional Prolog procedure box model defines four ports (*call*, *exit*, *redo*, and *fail*) for describing control flow when calling a predicate:

`call`
: predicate call

`exit`
: success of a predicate call

`redo`
: backtracking into a predicate

`fail`
: failure of a predicate call

Logtalk, as found on some recent Prolog systems, adds a port for dealing with exceptions thrown when calling a predicate:

`exception`
: predicate call throws an exception

In addition to the ports described above, Logtalk adds two more ports, `fact` and `rule`, which show the result of the unification of a goal with, respectively, a fact and a rule head:

`fact`
: unification success between a goal and a fact

`rule`
: unification success between a goal and a rule head

Following Prolog tradition, the user may define for which ports the debugger should pause for user interaction by specifying a list of *leashed* ports. Unleashed ports are just printed with no pause for user interaction when tracing. For example:

```
| ?- debugger::leash([call, exit, fail]).
```

Alternatively, the user may use an atom abbreviation for a pre-defined set of ports. For example:

```
| ?- debugger::leash(loose).
```

The abbreviations defined in Logtalk are similar to those defined on some Prolog compilers:

```
none
     []
loose
     [fact, rule, call]
half
     [fact, rule, call, redo]
tight
     [fact, rule, call, redo, fail, exception]
full
     [fact, rule, call, exit, redo, fail, exception]
```

By default, the debugger pauses at every port for user interaction.

1.18.3 Activating the debugger

The debuggerp::trace/0 and debuggerp::debug/0 predicates implicitly select the `debugger` tool as the active debug handler. If you have additional debug handlers loaded (e.g., the `ports_profiler` tool), those would no longer be active (there can be only one active debug handler at any given time). The debuggerp::nodebug/0 predicate implicitly deselects the `debugger` tool as the active debug handler.

1.18.4 Defining breakpoints

The `debugger` tool provides the following breakpoint types where the debugger pauses at a leashed port for user interaction:

- **Predicate breakpoints**
  Traditional Prolog spy points are defined using a predicate (or a non-terminal) indicator.
- **Clause breakpoints**
  Defined using the location of a clause.
- **Conditional breakpoints**
  Defined using the location of a clause and a condition for pausing.
- **Hit count breakpoints**
  Defined using the location of a clause and an unification count expression as a condition for pausing.

- **Triggered breakpoints**
  Defined using the location of a clause and another breakpoint that must be hit first as a condition for pausing.
- **Context breakpoints**
  Defined using execution context and goal templates as a condition for pausing.

Clause breakpoints are checked when the current goal successfully unifies with a clause head. To simplify their definition, these are specified using the entity identifier instead of the file name (as all entities share a single namespace, an entity can only be defined in a single file) and the first line number of the clause head. But note that only some Prolog backends provide accurate source file term line numbers. Check the *debugger* tool documentation for details.

Defining predicate and clause breakpoints

Predicate and clause breakpoints can be defined using the debugger `spy/1` predicate. The argument can be a predicate indicator (`Name/Arity`), a non-terminal indicator (`Name//Arity`), a clause location (expressed as an `Entity-Line` pair), or a list of breakpoints. For example:

```
| ?- debugger::spy(person-42).

All specified breakpoints added.
yes

| ?- debugger::spy(foo/2).

All specified breakpoints added.
yes

| ?- debugger::spy([foo/4, bar//1, agent-99]).

All specified breakpoints added.
yes
```

Note that setting a clause breakpoint implicitly removes any existing conditional breakpoint, triggered breakpoint, or log point for the same clause.

Unconditional clause and predicate breakpoints can be removed by using the debugger `nospy/1` predicate. The argument can also be a list of breakpoints or a non-instantiated variable, in which case all breakpoints will be removed. For example:

```
| ?- debugger::nospy(_).

All matching predicate and clause breakpoints removed.
yes
```

Defining conditional breakpoints

Conditional clause breakpoints are specified using the debugger `spy/3` predicate. The condition can be a lambda expression, an unification count expression (see next section), or another breakpoint (see next section).

The supported lambda expressions are `[Count, N, Goal]>>Condition` and `[Goal]>>Condition` where `Count` is the unification count, `N` is the goal invocation number, and `Goal` is the goal that unified with the clause head; `Condition` is called in the context of the `user` pseudo-object and must not have any side effects. Some examples:

```
| ?- debugger::spy(planet, 76, [weight(m1,_)]>>true).

Conditional breakpoint added.
yes
```

Note that setting a conditional breakpoint will remove any existing clause breakpoint or log point for the same location.

Conditional breakpoints can be removed by using the debugger `nospy/3` predicate. For example:

```
| ?- debugger::nospy(planet, _, _).

All matching conditional breakpoints removed.
yes
```

Defining hit count breakpoints

Conditional clause breakpoints that depend on the unification count are known as *hit count* clause breakpoints. The debugger pauses at a hit count breakpoint depending on an unification count expression:

- `>(Count)` - break when the unification count is greater than `Count`
- `>=(Count)` - break when the unification count is greater than or equal to `Count`
- `=:=(Count)` - break when the unification count is equal to `Count`
- `=<(Count)` - break when the unification count is less than or equal to `Count`
- `<(Count)` - break when the unification count is less than `Count`
- `mod(M)` - break when the unification count modulo `M` is zero
- `Count` - break when the unification count is greater than or equal to `Count`

For example:

```
| ?- debugger::spy(planet, 41, =<(2)).

Conditional breakpoint added.
yes
```

Defining triggered breakpoints

Conditional clause breakpoints that depend on other clause breakpoint or on a log point are known as *triggered* clause breakpoints. The debugger only pauses at a triggered breakpoint if the clause breakpoint or log point it depends on is hit first. For example:

```
| ?- debugger::spy(mars, 98, planet-76).

Triggered breakpoint added.
yes
```

In this case, the debugger will break for user interaction at the unification port for the clause in the source file defining the `mars` object at line 98 if and only if the debugger paused earlier at the unification port for the clause in the source file defining the `planet` category at line 76.

The debugger prints a `^` character at the beginning of the line for easy recognition of triggered breakpoints.

Defining context breakpoints

A context breakpoint is a tuple describing a message execution context and a goal:

```
(Sender, This, Self, Goal)
```

The debugger pauses for user interaction whenever the breakpoint goal and execution context subsume the goal currently being executed and its execution context. The user may establish any number of context breakpoints as necessary. For example, in order to call the debugger whenever a predicate defined on an object named `foo` is called, we may define the following context breakpoint:

```
| ?- debugger::spy(_, foo, _, _).

Spy point set.
yes
```

For example, we can spy all calls to a `foo/2` predicate with a *bar* atom in the second argument by setting the condition:

```
| ?- debugger::spy(_, _, _, foo(_, bar)).

Spy point set.
yes
```

The debugger `nospy/4` predicate may be used to remove all matching breakpoints. For example, the call:

```
| ?- debugger::nospy(_, _, foo, _).

All matching context breakpoints removed.
yes
```

will remove all context breakpoints where the value of *self* is the atom `foo`.

Removing all breakpoints

We can remove all breakpoints by using the debugger `nospyall/0` predicate:

```
| ?- debugger::nospyall.

All breakpoints removed.
yes
```

There's also a `reset/0` predicate that can be used to reset the debugger to its default settings and delete all defined breakpoints and log points.

1.18.5 Defining log points

Logtalk log points are similar to breakpoints. Therefore, the line number must correspond to the first line of an entity clause. When the debugger reaches a log point, it prints a log message and continues without pausing execution for reading a port command. When the log message is an empty atom, the default port output message is printed. When the log message starts with a `%` character, the default port output message is printed, followed by the log message. In these two cases, the debugger prints a `@` character at the beginning of the line for easy recognition of log points output. When the log message is neither empty nor starts with a `%` character, the log message is printed instead of the default port output message. In this case, the message can contain `$KEYWORD` placeholders that are expanded at runtime. The valid keywords are:

- `PORT`
- `ENTITY`
- `CLAUSE_NUMBER`
- `FILE`
- `LINE`
- `UNIFICATION_COUNT`
- `INVOCATION_NUMBER`
- `GOAL`
- `PREDICATE`
- `EXECUTION_CONTEXT`
- `SENDER`
- `THIS`
- `SELF`
- `METACALL_CONTEXT`
- `COINDUCTION_STACK`
- `THREAD`

Log points are defined using the `log/3` predicate. For example:

```
| ?- debugger::log(agent, 99, '% At the secret headquarters!').
     Log point added.
yes

```

(continues on next page)

(continued from previous page)

```
| ?- debugger::log(loop, 42, 'Message $PREDICATE from $SENDER at thread $THREAD').
     Log point added.
yes
```

The `logging/3` and `nolog/3` predicate can be used to, respectively, query and remove log points. There's also a `nologall/0` predicate that removes all log points.

Note that setting a log point will remove any existing clause breakpoint for the same location.

1.18.6 Tracing program execution

Logtalk allows tracing of execution for all objects compiled in debug mode. To start the debugger in trace mode, write:

```
| ?- debugger::trace.

yes
```

Next, type the query to be debugged. For example, using the `family` example in the Logtalk distribution compiled for debugging:

```
| ?- addams::sister(Sister, Sibling).
     Call: (1) sister(_1082,_1104) ?
     Rule: (1) sister(_1082,_1104) ?
     Call: (2) ::female(_1082) ?
     Call: (3) female(_1082) ?
     Fact: (3) female(morticia) ?
    *Exit: (3) female(morticia) ?
    *Exit: (2) ::female(morticia) ?
    ...
```

While tracing, the debugger will pause for user input at each leashed port, printing an informative message. Each trace line starts with the port, followed by the goal invocation number, followed by the goal. The invocation numbers are unique and allow us to correlate the ports used for a goal. In the output above, you can see, for example, that the goal `::female(_1082)` succeeds with the answer `::female(morticia)`. The debugger also provides determinism information by prefixing the `exit` port with a `*` character when a call succeeds with choice-points pending, thus indicating that there might be alternative solutions for the goal.

Note that breakpoints are ignored when tracing. But when a breakpoint is set for the current predicate or clause, the debugger prints, before the port name and number, a `+` character for predicate breakpoints, a `#` character for clause breakpoints, a `?` character for conditional clause breakpoints, a `^` for triggered breakpoints, and a `*` character for context breakpoints. For example:

```
| ?- debugger::spy(female/2).

yes

| ?- addams::sister(Sister, Sibling).
     Call: (1) sister(_1078,_1100) ?
     Rule: (1) sister(_1078,_1100) ?
     Call: (2) ::female(_1078) ?
  +  Call: (3) female(_1078) ?
```

To stop tracing (but still allowing the debugger to pause at the defined breakpoints), write:

```
| ?- debugger::notrace.

yes
```

1.18.7 Debugging using breakpoints

Tracing a program execution may generate large amounts of debugging data. Debugging using breakpoints allows the user to concentrate on specific points of the code. To start a debugging session using breakpoints, write:

```
| ?- debugger::debug.

yes
```

For example, assuming the predicate breakpoint we set in the previous section on the `female/1` predicate:

```
| ?- addams::sister(Sister, Sibling).
  +  Call: (3) female(_1078) ?
```

To stop the debugger, write:

```
| ?- debugger::nodebug.

yes
```

Note that stopping the debugger does not remove any defined breakpoints or log points.

1.18.8 Debugging commands

The debugger pauses for user interaction at leashed ports when tracing and when hitting a breakpoint. The following commands are available:

c — creep
: go on; you may use the spacebar, return, or enter keys in alternative

l — leap
: continues execution until the next breakpoint is found

s — skip
: skips tracing for the current goal; valid at call, redo, and unification ports

S - Skip
: similar to skip but displaying all intermediate ports unleashed

q — quasi-skip
: skips tracing until returning to the current goal or reaching a breakpoint; valid at call and redo ports

r — retry
: retries the current goal but side-effects are not undone; valid at the fail port

j — jump
: reads invocation number and continues execution until a port is reached for that number

z — zap
: reads either a port name and continues execution until that port is reached or a negated port name and continues execution until a port other than the negated port is reached

`i` — ignore
ignores goal, assumes that it succeeded; valid at call and redo ports

`f` — fail
forces backtracking; may also be used to convert an exception into a failure

`n` — nodebug
turns off debugging

`N` — notrace
turns off tracing

`@` — command; `!` can be used in alternative
reads and executes a query

`b` — break
suspends execution and starts new interpreter; type `end_of_file` to terminate

`a` — abort
returns to top level interpreter

`Q` — quit
quits Logtalk

`p` — print
writes current goal using the `print/1` predicate if available

`d` — display
writes current goal without using operator notation

`w` — write
writes current goal quoting atoms if necessary

`$` — dollar
outputs the compiled form of the current goal (for low-level debugging)

`x` — context
prints execution context

`.` — file
prints file, entity, predicate, and line number information at an unification port

`e` — exception
prints exception term thrown by the current goal

`E` — raise exception
reads and throws an exception term

`=` — debugging
prints debugging information

`<` — write depth
sets the write term depth (set to `0` to reset)

`*` — add
adds a context breakpoint for the current goal

`/` — remove
removes a context breakpoint for the current goal

`+` — add
adds a predicate breakpoint for the current goal

- — remove
removes a predicate breakpoint for the current goal

 # — add
adds a breakpoint for the current clause

| — remove
removes a breakpoint for the current clause

h — condensed help
prints list of command options

? — extended help
prints list of command options

1.18.9 Customizing term writing

Debugging complex applications often requires customizing term writing. The available options are limiting the writing depth of large compound terms and using the `p` command at a leashed port. This command uses the `format/3` de facto standard predicate with the `~p` formatting option to delegate writing the term to the `print/1` predicate. But note that some backends don't support this formatting option.

Term write depth

The terms written by the debugger can be quite large depending on the application being debugged. As described in the previous section, the debugger accepts the `<` command to set the maximum write term depth for compound terms. This command requires that the used *backend Prolog compiler* supports the non-standard but common `max_depth/1` option for the `write_term/3` predicate. When the compound term being written is deeply nested, the sub-terms are only written up to the specified depth with the omitted sub-terms replaced usually by `...`. For example:

```
| ?- write_term([0,1,2,3,4,5,6,7,8,9], [max_depth(5)]).

[0,1,2,3,4|...]
yes
```

The default maximum depth depends on the backend. To print compound terms without a depth limit, set it explicitly to zero if necessary. The debuggerp::write_max_depth/1 and debuggerp::set_write_max_depth/1 predicates can be used to query and set the default maximum depth.

Custom term writing

The implicit use of the traditional `print/1` predicate (using the `p` command) and the `portray/1` user-defined hook predicate requires backend Prolog compiler support for these predicates. See the documentation of the backend you intend to use for details. As an example, assuming the following `portray/1` definition:

```
portray(e(V1,V2)) :-
    format('~q ---> ~q~n', [V1,V2]).
```

Calling the `print/1` predicate with e.g. a `e(x1,x7)` compound term argument will output:

```
| ?- print(e(x1,x7)).

```

(continues on next page)

(continued from previous page)

```
x1 ---> x7
yes
```

1.18.10 Context-switching calls

Logtalk provides a debugging control construct, *(<<)/2*, which allows the execution of a query within the context of an object. Common debugging uses include checking an object local predicates (e.g. predicates representing internal dynamic state) and sending a message from within an object. This control construct may also be used to write unit tests.

Consider the following toy example:

```
:- object(broken).

    :- public(a/1).

    a(A) :- b(A, B), c(B).
    b(1, 2). b(2, 4). b(3, 6).
    c(3).

:- end_object.
```

Something is wrong when we try the object public predicate, `a/1`:

```
| ?- broken::a(A).

no
```

For helping in diagnosing the problem, instead of compiling the object in debug mode and doing a *trace* of the query to check the clauses for the non-public predicates, we can instead simply type:

```
| ?- broken << c(C).

C = 3
yes
```

The `(<<)/2` control construct works by switching the execution context to the object in the first argument and then compiling and executing the second argument within that context:

```
| ?- broken << (self(Self), sender(Sender), this(This)).

Self = broken
Sender = broken
This = broken

yes
```

As exemplified above, the `(<<)/2` control construct allows you to call an object local and private predicates. However, it is important to stress that we are not bypassing or defeating an object predicate scope directives. The calls take place within the context of the specified object, not within the context of the object making the `(<<)/2` call. Thus, the `(<<)/2` control construct implements a form of *execution-context-switching*.

The availability of the `(<<)/2` control construct is controlled by the *context_switching_calls* compiler flag (its default value is defined in the adapter files of the backend Prolog compilers).

1.18.11 Debugging messages

Calls to the *logtalk::print_message/3* predicate where the message kind is either `debug` or `debug(Group)` are only printed, by default, when the *debug* flag is turned on. Moreover, these calls are suppressed by the compiler when the *optimize* flag is turned on. Note that actual printing of debug messages does not require compiling the code in debug mode, only turning on the `debug` flag.

Meta-messages

To avoid having to define *message_tokens//2* grammar rules for translating each and every debug message, Logtalk provides default tokenization for seven *meta-messages* that cover the most common cases:

`@Message`
: By default, the message is printed as passed to the `write/1` predicate followed by a newline.

`Key-Value`
: By default, the message is printed as `Key: Value` followed by a newline. The key is printed as passed to the `write/1` predicate while the value is printed as passed to the `writeq/1` predicate.

`Format+Arguments`
: By default, the message is printed as passed to the `format/2` predicate.

`List`
: By default, the list items are printed indented, one per line. The items are preceded by a dash and can be `@Message`, `Key-Value`, or `Format+Arguments` messages. If that is not the case, the item is printed as passed to the `writeq/1` predicate.

`Title::List`
: By default, the title is printed, followed by a newline and the indented list items, one per line. The items are printed as in the `List` meta message.

`[Stream,Prefix]>>Goal`
: By default, call user-defined printing `Goal` in the context of `user`. The use of a lambda expression allows passing the message stream and prefix. Printing the prefix is delegated to the goal.

`[Stream]>>Goal`
: By default, call user-defined printing `Goal` in the context of `user`. The use of a lambda expression allows passing the message stream.

Some simple examples of using these meta-messages:

```
| ?- logtalk::print_message(debug, core, @'Phase 1 completed').
yes

| ?- logtalk::print_message(debug, core, [Stream]>>write(Stream,foo)).
yes

| ?- set_logtalk_flag(debug, on).
yes

| ?- logtalk::print_message(debug, core, [Stream]>>write(Stream,foo)).
foo
yes
```

(continues on next page)

(continued from previous page)

```
| ?- logtalk::print_message(debug, core, @'Phase 1 completed').
>>> Phase 1 completed
yes

| ?- logtalk::print_message(debug, core, answer-42).
>>> answer: 42
yes

| ?- logtalk::print_message(debug, core, 'Position: <~d,~d>'+[42,23]).
>>> Position: <42,23>
yes

| ?- logtalk::print_message(debug, core, [arthur,ford,marvin]).
>>> - arthur
>>> - ford
>>> - marvin
yes

| ?- logtalk::print_message(debug, core, names::[arthur,ford,marvin]).
>>> names:
>>> - arthur
>>> - ford
>>> - marvin
yes
```

The `>>>` prefix is the default message prefix for `debug` messages. It can be redefined using the *logtalk::message_prefix_stream/4* hook predicate. For example:

```
:- multifile(logtalk::message_prefix_stream/4).
:- dynamic(logtalk::message_prefix_stream/4).

logtalk::message_prefix_stream(debug, core, '(dbg) ', user_error).
```

Selective printing of debug messages

By default, all debug messages are either printed or skipped, depending on the *debug* and *optimize* flags. When the code is not compiled in optimal mode, the *debug_messages* tool allows selective enabling of debug messages per *component* and per debug group. For example, to enable all `debug` and `debug(Group)` messages for the `parser` component:

```
% upon loading the tool, all messages are disabled by default:
| ?- logtalk_load(debug_messages(loader)).
...

% enable both debug and debug(_) messages:
| ?- debug_messages::enable(parser).
yes
```

To enable only `debug(tokenization)` messages for the `parser` component:

```
% first disable any and all enabled messages:
| ?- debug_messages::disable(parser).
yes

% enable only debug(tokenization) messages:
| ?- debug_messages::enable(parser, tokenization).
yes
```

See the tool documentation for more details.

1.18.12 Using the term-expansion mechanism for debugging

Debugging messages only output information by default. These messages can, however, be intercepted to perform other actions. An alternative is to use instead the *term-expansion mechanism* for conditional compilation of debugging goals. For example, the *hook_objects* library provides a print_goal_hook object that simplifies printing entity goals before or after calling them by simply prefixing them with an operator. See the library and hook object documentation for details. You can also define your own specialized hook objects for custom debugging tasks.

1.18.13 Ports profiling

The Logtalk distribution includes a *ports_profiler* tool based on the same procedure box model described above. This tool is specially useful for debugging performance issues (e.g., due to lack of determinism or unexpected backtracking). See the tool documentation for details.

1.18.14 Debug and trace events

The debugging API defines two multifile predicates, logtalk::trace_event/2 and logtalk::debug_handler/3 for handling trace and debug events. It also provides a logtalk::debug_handler/1 multifile predicate that allows an object (or a category) to declare itself as a debug handler provider. The Logtalk `debugger` and `ports_profiler` tools are regular applications that are implemented using this API, which can also be used to implement alternative or new debugging-related tools. See the API documentation for details and the source code of the `debugger` and `ports_profiler` tools for usage examples.

To define a new debug handler provider, add (to an object or category) clauses for the `debug_handler/1` and `debug_handler/3` predicates. For example:

```
% declare my_debug_handler as a debug handler provider
:- multifile(logtalk::debug_handler/1).
logtalk::debug_handler(my_debug_handler).

% handle debug events
:- multifile(logtalk::debug_handler/3).
logtalk::debug_handler(my_debug_handler, Event, ExCtx) :-
    debug_handler(Event, ExCtx).

debug_handler(fact(Entity,Fact,Clause,File,Line), ExCtx) :-
    ...
debug_handler(rule(Entity,Head,Clause,File,Line), ExCtx) :-
    ...
debug_handler(top_goal(Goal, TGoal), ExCtx) :-
```

(continues on next page)

(continued from previous page)

```
    ...
debug_handler(goal(Goal, TGoal), ExCtx) :-
    ...
```

Your debug handler provider should also either automatically call the logtalk::activate_debug_handler/1 and logtalk::deactivate_debug_handler/0 predicate or provide public predicates to simplify calling these predicates. For example:

```
:- public(start/0).
start :-
   logtalk::activate_debug_handler(my_debug_handler).

:- public(stop/0).
stop :-
   logtalk::deactivate_debug_handler.
```

If you only need to define a trace event handler, then simply define clauses for the logtalk::trace_event/2 multifile predicate:

```
:- multifile(logtalk::trace_event/2).
:- dynamic(logtalk::trace_event/2).

% the Logtalk runtime calls all defined logtalk::trace_event/2 hooks using
% a failure-driven loop; thus we don't have to worry about handling all
% events or failing after handling an event to give other hooks a chance
logtalk::trace_event(fact(Entity, Fact, N, _, _), _) :-
    ...
logtalk::trace_event(rule(Entity, Head, N, _, _), _) :-
    ...
```

1.18.15 Source-level debugger

A minimal source-level debugger is provided by the Logtalk for VSCode extension: when debugging in the integrated terminal using the `debugger` tool, the current clause (at leashed unification ports) is shown in the active editor window. The extension can also be used with VSCodium. See its documentation for more details.

1.19 Performance

Logtalk is implemented as a *trans-compiler* to Prolog. When compiling predicates, it preserves in the generated Prolog code all cases of first-argument indexing and tail-recursion. In practice, this means that if you know how to write efficient Prolog predicates, you already know the basics of how to write efficient Logtalk predicates.

The Logtalk compiler appends a single argument to the compiled form of all entity predicate clauses. This hidden argument is used to pass the *execution-context* when proving a query. In the common case where a predicate makes no calls to the *execution-context predicates* and *message-sending control constructs* and is neither a meta-predicate nor a coinductive predicate, the execution-context is simply passed between goals. In this case, with most backend Prolog virtual machines, the cost of this extra argument is null or negligible. When the execution-context needs to be accessed (e.g. to fetch the value of *self* for a *(::)/1* call) there may

be a small inherent overhead due to the access to the individual arguments of the compound term used to represent the execution-context.

1.19.1 Source code compilation modes

Source code can be compiled in *optimal*, *normal*, or *debug* mode, depending on the *optimize* and *debug* compiler flags. Optimal mode is used when deploying an application, while normal and debug modes are used when developing an application. Compiling code in optimal mode enables several optimizations, notably the use of *static binding* whenever enough information is available at compile-time. In debug mode, most optimizations are turned off, and the code is instrumented to generate *debug events* that enable developer tools such as the *command-line debugger* and the *ports profiler*.

1.19.2 Source code compilation order

Static binding optimizations, notably message sending and super calls, require referenced code to be compiled before the calls so that the calls can be resolved at compile time. The compiler prints warnings when the file compilation/loading order is not ideal. See also the section below on circular references.

1.19.3 Local predicate calls

Local calls to object (or category) predicates have zero overhead in terms of the number of inferences, as expected, compared with local Prolog calls.

1.19.4 Calls to imported or inherited predicates

Assuming the *optimize* flag is turned on and a static predicate, *(^ ^)/1* calls have zero overhead in terms of number of inferences.

1.19.5 Calls to module predicates

Local calls from an object (or category) to a module predicate have zero overhead (assuming both the module and the predicate are bound at compile-time).

1.19.6 Messages

Logtalk implements *static binding* and *dynamic binding* for message-sending calls. For dynamic binding, a caching mechanism is used by the runtime. It's useful to measure the performance overhead in *number of logic inferences* compared with plain Prolog and Prolog modules. Note that the number of logic inferences is a metric independent of the chosen backend Prolog compiler. The results for Logtalk 3.17.0 and later versions are:

- Static binding: 0
- Dynamic binding (object bound at compile-time): +1
- Dynamic binding (object bound at runtime): +2

Static binding is the common case with libraries and most application code; it requires compiling code with the *optimize* flag turned on. Dynamic binding numbers are after the first call (i.e. after the generalization of the query is cached). All numbers with the *events* flag set to `deny` (setting this flag to `allow` adds an overhead

of +5 inferences to the results above; note that this flag can be defined on a per-object basis as needed instead of globally and thus minimizing the performance impact).

The dynamic binding caches assume the used *backend Prolog compiler* does indexing of dynamic predicates. This is a common feature of modern Prolog systems, but the actual details vary from system to system and may have an impact on dynamic binding performance.

Note that messages to *self* (*(::)/1* calls) and messages to an object (*(::)/2* calls) from the top-level interpreter always use dynamic binding, as the object that receives the message is only known at runtime.

Messages sent from Prolog modules may use static binding depending on the used backend Prolog compiler native support for goal-expansion. Consult the Prolog compiler documentation and adapter file notes for details.

Warning

Some Prolog systems provide a `time/1` predicate that also reports the number of inferences. But the reported numbers are often misleading when the predicate is called from the top-level. Besides common top-level bookkeeping operations (e.g., keeping track of goal history or applying goal-expansion) that may influence the inference counting, the Logtalk runtime code for a `(::)/2` top-level goal is necessarily different from the code generated for a `(::)/2` goal from a compiled object, as it requires *runtime* compilation of the goal into the same low-level message-sending primitive (assuming dynamic-binding is also required for the compiled object goal).

1.19.7 Automatic expansion of built-in meta-predicates

The compiler always expands calls to the *forall/2*, *once/1*, and *ignore/1* meta-predicates into equivalent definitions using the negation and conditional control constructs. It also expands calls to the *call/1-N*, *phrase/2*, and *phrase/3* meta-predicates when the first argument is bound. These expansions are performed independently of the `optimize` flag value.

1.19.8 Inlining

When the *optimize* flag is turned on, the Logtalk compiler performs *inlining* of predicate calls whenever possible. This includes calls to Prolog predicates that are either built-in, foreign, or defined in a module (including `user`). Inlining notably allows wrapping module or foreign predicates using an object without introducing any overhead. In the specific case of the *execution-context predicates*, calls are inlined independently of the `optimize` flag value.

1.19.9 Generated code simplification and optimizations

When the *optimize* flag is turned on, the Logtalk compiler simplifies and optimizes generated clauses (including those resulting from the compilation of grammar rules), by flattening conjunctions, folding left unifications (e.g. generated as a by-product of the compilation of grammar rules), and removing redundant calls to `true/0`.

When using *lambda expressions* and library meta-predicates, use the *meta_compiler* library to avoid most meta-call overheads.

1.19.10 Size of the generated code

The size of the intermediate Prolog code generated by the compiler is proportional to the size of the source code. Assuming that the *term-expansion mechanism* is not used, each predicate clause in the source code is compiled into a single predicate clause. But the Logtalk compiler also generates internal tables for the defined entities, for the entity relations, and for the declared and defined predicates. These tables enable support for fundamental features such as *inheritance* and *reflection*. The size of these tables is proportional to the number of entities, entity relations, and predicate declarations and definitions. When the *source_data* is turned on (the default when *developing* an application), the generated code also includes additional data about the source code, such as entity and predicate positions in a source file. This data enables advanced developer tool functionality. But it is usually not required when *deploying* an application. Thus, turning this flag off is a common setting for minimizing an application footprint.

1.19.11 Circular references

Circular references, i.e. two objects sending messages to each other, are relatively costly and should be avoided if possible as they prevent using static binding for the messages sent from the first loaded object to the second object. The *logtalk_make(circular)* goal (or its `{@}` top-level abbreviation) can be used to scan for circular entity dependencies. The linter also warns by default about non-ideal file loading order (controlled by the *unknown_entities* flag).

1.19.12 Debug mode overhead

Code compiled in debug mode runs slower, as expected, when compared with normal or optimized mode. The overhead depends on the number of *debug events* generated when running the application. A debug event is simply a pass on a call or unification port of the *procedure box model*. These debug events can be intercepted by defined clauses for the logtalk::trace_event/2 and logtalk::debug_handler/3 multifile predicates. With no application (such as a debugger or a port profiler) loaded defining clauses for these predicates, each goal has an overhead of four extra inferences due to the runtime checking for a definition of the hook predicates and a meta-call of the user goal. The clause head unification events result in one or more inferences per goal (depending on the number of clauses whose head unifies with the goal and backtracking). In practice, this overhead translates to code compiled in debug mode running typically ~2x to ~7x slower than code compiled in normal or optimized mode, depending on the application (the exact overhead is proportional to the number of passes on the call and unification ports; deterministic code often results in a relatively larger overhead when compared with code performing significant backtracking).

1.19.13 Other considerations

One aspect of performance that affects both Logtalk and Prolog code is the characteristics of the Prolog VM. The Logtalk distribution includes two examples, bench and benchmarks, to help evaluate performance with specific backend Prolog systems. A table with benchmark results for a subset of the supported systems is also available at the Logtalk website. But note that multiple factors affect the performance of an application. The benchmark examples and their results only provide a partial assessment.

1.20 Installing Logtalk

This page provides an overview of Logtalk installation requirements and instructions and a description of the files contained in the Logtalk distribution. For detailed, up-to-date installation and configuration instructions, please see the `README.md`, `INSTALL.md`, and `CUSTOMIZE.md` files distributed with Logtalk. The broad compatibility of Logtalk, both with Prolog compilers and operating-systems, together with all the possible user scenarios, means that installation can vary from very simple, by running an installer or a couple of scripts, to the need of patching both Logtalk and Prolog compilers to workaround the lack of strong Prolog standards or to cope with the requirements of less common operating-systems.

The preferred installation scenario is to have Logtalk installed in a system-wide location, thus available for all users, and a local copy of user-modifiable files on each user home directory (even when you are the single user of your computer). This scenario allows each user to independently customize Logtalk and to freely modify the provided libraries and programming examples. Logtalk installers, installation shell scripts, and Prolog integration scripts favor this installation scenario, although alternative installation scenarios are always possible. The installers set two environment variables, `LOGTALKHOME` and `LOGTALKUSER`, pointing, respectively, to the Logtalk installation folder and to the Logtalk user folder.

User applications should preferably be kept outside of the Logtalk user folder created by the installation process, as updating Logtalk often results in updating the contents of this folder. If your applications depend on customizations to the distribution files, backup those changes before updating Logtalk.

1.20.1 Hardware and software requirements

Computer and operating system

Logtalk is compatible with almost any computer/operating-system with a modern, standards-compliant, Prolog compiler available.

Prolog compiler

Logtalk requires a *backend Prolog compiler* supporting official and de facto standards. Capabilities needed by Logtalk that are not defined in the official ISO Prolog Core standard include:

- access to predicate properties
- operating-system access predicates
- de facto standard predicates not (yet) specified in the official standard

Logtalk needs access to the predicate property `built_in` to properly compile objects and categories that contain Prolog built-in predicate calls. In addition, some Logtalk built-ins need to know the dynamic/static status of predicates to ensure correct application. The ISO standard for Prolog modules defines a `predicate_property/2` predicate that is already implemented by most Prolog compilers. Note that if these capabilities are not built-in the user cannot easily define them.

For optimal performance, Logtalk requires that the Prolog compiler supports **first-argument indexing** for both static and dynamic code (most modern compilers support this feature).

Since most Prolog compilers are moving closer to the ISO Prolog standard [ISO95], it is advisable that you try to use the most recent version of your favorite Prolog compiler.

1.20.2 Logtalk installers

Logtalk installers are available for macOS, Linux, and Microsoft Windows. Depending on the chosen installer, some tasks (e.g., setting environment variables or integrating Logtalk with some Prolog compilers) may need to be performed manually.

1.20.3 Source distribution

Logtalk sources are available in a `tar` archive compressed with `bzip2`, `lgt3xxx.tar.bz2`. You may expand the archive by using a decompressing utility or by typing the following commands at the command-line:

```
% tar -jxvf lgt3xxx.tar.bz2
```

This will create a sub-directory named `lgt3xxx` in your current directory. Almost all files in the Logtalk distribution are text files. Different operating-systems use different end-of-line codes for text files. Ensure that your decompressing utility converts the end-of-lines of all text files to match your operating system.

1.20.4 Distribution overview

In the Logtalk installation directory, you will find the following files and directories:

`ACKNOWLEDGMENTS.md` - List of authors, contributors, sponsors, and open source credits

`BIBLIOGRAPHY.bib` – Logtalk bibliography in BibTeX format

`CITATION.cff` - Information on how to cite Logtalk

`CODE_OF_CONDUCT.md` - Code of conduct for contributors and users posting on support forums

`CUSTOMIZE.md` – Logtalk end-user customization instructions

`INSTALL.md` – Logtalk installation instructions

`LICENSE.txt` – Logtalk user license

`NOTICE.txt` – Logtalk copyright notice

`QUICK_START.md` – Quick start instructions for those that do not like to read manuals

`README.md` – several useful pieces of information

`RELEASE_NOTES.md` – release notes for this version

`UPGRADING.md` – instructions on how to upgrade your programs to the current Logtalk version

`VERSION.txt` – file containing the current Logtalk version number (used for compatibility checking when upgrading Logtalk)

`adapters`

`NOTES.md` – notes on the provided adapter files

`template.pl` – template adapter file

`...` – specific adapter files

`coding`

`NOTES.md` – notes on syntax highlighter and text editor support files providing syntax coloring for publishing and editing Logtalk source code

`...` – syntax coloring support files

contributions
NOTES.md – notes on the user-contributed code

... – user-contributed code files

core
NOTES.md – notes on the current status of the compiler and runtime

... – core source files

docs/apis
NOTES.md – notes on the provided documentation for core, library, tools, and contributions entities

index.html – root document for all entities documentation

... – other entity documentation files

docs/handbook
NOTES.md – notes on the provided documentation

bibliography.html – bibliography

glossary.html – glossary

index.html – root document for all documentation

... – other documentation files

docs/man
... – POSIX man pages for the shell scripts

examples
NOTES.md – short description of the provided examples

ack
NOTES.md – example notes and sample queries

loader.lgt – loader utility file for the example objects

... – ack example source files

... – other examples

integration
NOTES.md – notes on scripts for Logtalk integration with Prolog compilers

... – Prolog integration scripts

library
NOTES.md – short description of the library contents

all_loader.lgt – loader utility file for all library entities

... – library source files

paths
NOTES.md – description on how to setup library and example paths

paths.pl – default library and example paths

ports
NOTES.md – description of included ports of third-party software

... – ports

samples
: `loader-sample.lgt` – sample loader file for user applications

`settings-sample.lgt` – sample file for user-defined Logtalk settings

`tester-sample.lgt` – sample file for helping to automate running user application unit tests

`tests-sample.lgt` – sample file for writing user application unit tests

scratch
: `NOTES.md` – notes on the scratch directory

scripts
: `NOTES.md` – notes on scripts for Logtalk user setup, packaging, and installation

`...` – packaging, installation, and setup scripts

tests
: `NOTES.md` – notes on the current status of the unit tests

`...` – unit tests for built-in features

tools
: `NOTES.md` – notes on the provided programming tools

`...` – programming tools

Adapter files

Adapter files provide the glue code between the Logtalk compiler/runtime and a Prolog compiler. Each adapter file contains two sets of predicates: ISO Prolog standard predicates and directives not built-in in the target Prolog compiler and Logtalk specific predicates.

Logtalk already includes ready-to-use adapter files for most academic and commercial Prolog compilers. If an adapter file is not available for the compiler that you intend to use, then you need to build a new one, starting from the included `template.pl` file. Start by making a copy of the template file. Carefully check (or complete if needed) each listed definition. If your Prolog compiler conforms to the ISO standard, this task should only take you a few minutes. In most cases, you can borrow code from the predefined adapter files. If you are unsure that your Prolog compiler provides all the ISO predicates needed by Logtalk, try to run the system by setting the unknown predicate error handler to report as an error any call to a missing predicate. Better yet, switch to a modern, ISO-compliant, Prolog compiler. If you send me your adapter file, with a reference to the target Prolog compiler, maybe I can include it in the next release of Logtalk.

The adapter files specify *default* values for most of the Logtalk *compiler flags*. They also specify values for read-only flags that are used to describe Prolog backend-specific features.

Compiler and runtime

The `core` sub-directory contains the Prolog and Logtalk source files that implement the Logtalk compiler and the Logtalk runtime. The compiler and the runtime may be split in two (or more) separate files or combined into a single file, depending on the Logtalk release that you are installing.

Library

The Logtalk distribution includes a standard library of useful objects, categories, and protocols. Read the corresponding `NOTES.md` file for details about the library contents.

Examples

The Logtalk distribution includes a large number of programming examples. The sources of each one of these examples can be found included in a subdirectory with the same name, inside the directory examples. The majority of these examples include tests and a file named `SCRIPT.txt` with sample calls. Some examples may depend on other examples and library objects to work properly. Read the corresponding `NOTES.md` file for details before running an example.

Logtalk source files

Logtalk source files are text files containing one or more entity definitions (objects, categories, or protocols). The Logtalk source files may also contain plain Prolog code. The extension `.lgt` is normally used. Logtalk compiles these files to plain Prolog by appending to the file name a suffix derived from the extension and by replacing the `.lgt` extension with `.pl` (`.pl` is the default Prolog extension; if your Prolog compiler expects the Prolog source filenames to end with a specific, different extension, you can set it in the corresponding adapter file).

1.21 Prolog integration and migration

This section provides suggestions for integrating and migrating plain Prolog code and Prolog module code to Logtalk. Detailed instructions are provided for encapsulating plain Prolog code in objects, converting Prolog modules into objects, and compiling and reusing Prolog modules as objects from inside Logtalk. An interesting application of the techniques described in this section is a solution for running a Prolog application that uses modules on a Prolog compiler with no module system. The *wrapper* tool can be used to help in migrating Prolog code.

1.21.1 Source files with both Prolog code and Logtalk code

Logtalk source files may contain plain Prolog code intermixed with Logtalk code. The Logtalk compiler simply copies the plain Prolog code as-is to the generated Prolog file. With Prolog modules, it is assumed that the module code starts with a `module/1-2` directive and ends at the end of the file. There is no module ending directive that would allow us to define more than one module per file. In fact, most, if not all, Prolog module systems always define a single module per file. Some of them mandate that the `module/1-2` directive be the first term in a source file. As such, when the Logtalk compiler finds a `module/1-2` directive, it assumes that all code that follows until the end of the file belongs to the module.

1.21.2 Encapsulating plain Prolog code in objects

Most applications consist of several plain Prolog source files, each one defining a few top-level predicates and auxiliary predicates that are not meant to be directly called by the user. Encapsulating plain Prolog code in objects allows us to make clear the different roles of each predicate, to hide implementation details, to prevent auxiliary predicates from being called outside the object, and to take advantage of Logtalk advanced code encapsulating and reusing features. It also simplifies using its developer tools.

Encapsulating Prolog code using Logtalk objects is simple. First, for each source file, add an opening object directive, *object/1-5*, to the beginning of the file and an ending object directive, *end_object/0*, to the end of the file. Choose an object name that reflects the purpose of the source file code (this is a good opportunity for code refactoring if necessary). Second, add *public/1* predicate directives for the top-level predicates that are used directly by the user or called from other source files. Third, we need to be able to call from inside an object predicates defined in other source files/objects. The easiest solution, which has the advantage of not requiring any changes to the predicate definitions, is to use the *uses/2* directive. If your Prolog compiler supports cross-referencing tools, you may use them to help you make sure that all calls to predicates on other source files/objects are listed in the *uses/2* directives. The Logtalk `wrapper` tool can also help in detecting cross-predicate calls. Compiling the resulting objects with the Logtalk *unknown_predicates* and *portability* flags set to `warning` will help you identify calls to predicates defined in other converted source files and possible portability issues.

Prolog multifile predicates

Prolog *multifile* predicates are used when clauses for the same predicate are spread among several source files. When encapsulating plain Prolog code that uses multifile predicates, it's often the case that the clauses of the multifile predicates get spread between different objects and categories, but conversion is straightforward. In the Logtalk object (or category) holding the multifile predicate *primary declaration*, add a *predicate scope directive* and a *multifile/1* directive. In all other objects (or categories) defining clauses for the multifile predicate, add a `multifile/1` directive and predicate clauses using the format:

```
:- multifile(Entity::Name/Arity).

Entity::Functor(...) :-
    ...
```

See the *section* on the `multifile/1` predicate directive for more information. An alternative solution is to simply keep the clauses for the multifile predicates as plain Prolog code and define, if necessary, a parametric object to encapsulate all predicates working with the multifile predicate clauses. For example, assume the following `multifile/1` directive:

```
% city(Name, District, Population, Neighbors)
:- multifile(city/4).
```

We can define a parametric object with `city/4` as its identifier:

```
:- object(city(_Name, _District, _Population, _Neighbors)).

    % predicates for working with city/4 clauses

:- end_object.
```

This solution is preferred when the multifile predicates are used to represent large tables of data. See the section on *parametric objects* for more details.

1.21.3 Converting Prolog modules into objects

Converting Prolog modules into objects may allow an application to run on a wider range of Prolog compilers, overcoming portability problems. Some Prolog compilers don't support a module system. Among those Prolog compilers that support a module system, the lack of standardization leads to several issues, notably with semantics, operators, and meta-predicates. In addition, the conversion allows you to take advantage of Logtalk more powerful abstraction and reuse mechanisms, such as separation between interface and implementation, inheritance, parametric objects, and categories. It also allows you to take full advantage of Logtalk developer tools for improved productivity.

Converting a Prolog module into an object is simplified when the directives used in the module are supported by Logtalk (see the listing in the next section). Assuming that this is the case, apply the following steps:

1. Convert the module `module/1` directive into an *object/1* opening object directive, using the module name as the object name. For `module/2` directives apply the same conversion and convert the list of exported predicates into *public/1* predicate directives. Add a closing object directive, *end_object/0*, at the end of the source code.
2. Convert any `export/1` directives into `public/1` predicate directives.
3. Convert any `use_module/1` directives for modules that will not be converted to objects into `use_module/2` directives (see next section), replacing the file spec in the first argument with the module name.
4. Convert any `use_module/1-2` directives referencing other modules also being converted to objects into Logtalk *uses/2* directives.
5. Convert each `reexport/1` directive into a *uses/2* directive and `public/1` predicate directives (see next section).
6. Convert any `meta_predicate/1` directives into Logtalk *meta_predicate/1* directives by replacing the module meta-argument indicator, `:`, with the Logtalk meta-argument indicator `0` for goal meta-arguments. For closure meta-arguments, use an integer denoting the number of additional arguments that will be appended to construct a goal. Arguments that are not meta-arguments are represented by the `*` character. Do not use argument mode indicators such as `?`, or `+`, or `-` as Logtalk supports *mode directives*.
7. Convert any explicit qualified calls to module predicates to messages by replacing the `(:)/2` operator with the *(::)/2* message-sending operator when the referenced modules are also being converted into objects. Calls in the pseudo-module `user` can be encapsulated using the *{}/1* Logtalk external call control construct. You can also use instead a *uses/2* directive where the first argument would be the atom `user` and the second argument a list of all external predicates. This alternative has the advantages of not requiring changes to the code making the predicate calls and of better visibility for the documenting and diagramming tools.
8. If your module uses the database built-in predicates to implement module-local mutable state using dynamic predicates, add both *private/1* and *dynamic/1* directives for each dynamic predicate.
9. If your module declares or defines clauses for multifile module predicates, replace the `(:)/2` functor by `(::)/2` in the `multifile/1` directives and in the clause heads for all modules defining the multifile predicates that are also being converted into objects; if that is not the case, just keep the `multifile/1` directives and the clause heads as-is.
10. Compile the resulting objects with the Logtalk *unknown_predicates*, and *portability* flags set to `warning` to help you locate possible issues and calls to proprietary Prolog built-in predicates and to predicates defined on other converted modules. In order to improve code portability, check the Logtalk library for possible alternatives to the use of proprietary Prolog built-in predicates.

Before converting your modules to objects, you may try to compile them first as objects (using the *logtalk_compile/1* Logtalk built-in predicates) to help identify any issues that must be dealt with when doing

the conversion to objects. Note that Logtalk supports compiling Prolog files as Logtalk source code without requiring changes to the file name extensions.

1.21.4 Compiling Prolog modules as objects

A possible alternative to porting Prolog code to Logtalk is to compile the Prolog source files using the `logtalk_load/1-2` and `logtalk_compile/1-2` predicates. The Logtalk compiler provides partial support for compiling Prolog modules as Logtalk objects. This support may allow using modules from a backend Prolog system in a different backend Prolog system, although its main purpose is to help in porting existing Prolog code to Logtalk in order to benefit from its extended language features and its developer tools. Why partial support? Although there is an ISO Prolog standard for modules, it is (rightfully) ignored by most implementers and vendors (due to its flaws and deviation from common practice). In addition, there is no de facto standard for module systems, despite otherwise frequent misleading claims. Key system differences include the set of implemented module directives, the directive semantics, the handling of operators, the locality of flags, and the integration of term-expansion mechanisms (when provided). Another potential issue is that, when compiling modules as objects, Logtalk assumes that any referenced module (e.g., using `use_module/1-2` directives) is also being compiled as an object. If that's not the case, the compiled module calls being compiled as message-sending goals will still work for normal predicates but will not work for meta-predicates called using implicit module qualification. The reason is that, unlike in Logtalk, calls to implicitly and explicitly qualified module meta-predicates have different semantics. Follows a discussion of other limitations of this approach that you should be aware of.

Supported module directives

Currently, Logtalk supports the following module directives:

`module/1`
: The module name becomes the object name.

`module/2`
: The module name becomes the object name. The exported predicates become public object predicates. The exported grammar rule non-terminals become public grammar rule non-terminals. The exported operators become public object operators but are not active elsewhere when loading the code.

`use_module/2`
: This directive is compiled as a Logtalk *uses/2* directive in order to ensure correct compilation of the module predicate clauses. The first argument of this directive must be the module **name** (an atom), not a module file specification (the adapter files attempt to use the Prolog dialect level term-expansion mechanism to find the module name from the module file specification). Note that the module is not automatically loaded by Logtalk (as it would be when compiling the directive using Prolog instead of Logtalk; the programmer may also want the specified module to be compiled as an object). The second argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal indicator (`Name//Arity`), a operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations. Predicate aliases can be declared using the notation `Name/Arity as Alias/Arity` or, in alternative, the notation `Name/Arity:Alias/Arity`. Similar for non-terminal aliases.

`export/1`
: Exported predicates are compiled as public object predicates. The argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal indicator (`Name//Arity`), an operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations.

`reexport/2`
: Reexported predicates are compiled as public object predicates. The first argument is the module name. The second argument must be a predicate indicator (`Name/Arity`), a grammar rule non-terminal

indicator (`Name//Arity`), an operator declaration, or a list of predicate indicators, grammar rule non-terminal indicators, and operator declarations. Predicate aliases can be declared using the notation `Name/Arity as Alias/Arity` or, in alternative, the notation `Name/Arity:Alias/Arity`. Similar for non-terminal aliases.

meta_predicate/1
: Module meta-predicates become object meta-predicates. All meta-predicates must be declared using the *meta_predicate/1* directive using Logtalk syntax for normal arguments and meta-arguments. Note that Prolog module meta-predicates and Logtalk meta-predicates don't share the same explicit-qualification calling semantics: in Logtalk, meta-arguments are always called in the context of the *sender*. Moreover, Logtalk is not based on the predicate-prefixing mechanism common to most Prolog module systems.

A common issue when compiling modules as objects is the use of the atoms `dynamic`, `discontiguous`, and `multifile` as operators in directives. For better portability, avoid this usage. For example, write:

```
:- dynamic([foo/1, bar/2]).
```

instead of:

```
:- dynamic foo/1, bar/2.
```

Another common issue is missing `meta_predicate/1`, `dynamic/1`, `discontiguous/1`, and `multifile/1` predicate directives. The Logtalk compiler supports detection of missing directives (by setting its *missing_directives* flag to `warning`).

When compiling modules as objects, you probably don't need event support turned on. You may use the *events* compiler flag to `deny` in the Logtalk compiling and loading built-in methods for a small performance gain for the compiled code.

Unsupported module directives

The `reexport/1` and `use_module/1` directives are not directly supported by the Logtalk compiler. But most Prolog adapter files provide support for compiling these directives using Logtalk's first stage of its *term-expansion mechanism*. Nevertheless, these directives can be converted, respectively, into a sequence of `:- use_module/2` and `export/1` directives and `use_module/2` directives by finding which predicates exported by the specified modules are reexported or imported into the module containing the directive. For `use_module/1` directives, finding the names of the imported predicates that are actually used is easy. First, comment out the directive and compile the file (making sure that the *unknown_predicates* compiler flag is set to `warning`). Logtalk will print a warning with a list of predicates that are called but never defined. Second, use this list to replace the `use_module/1` directives by `use_module/2` directives. You should then be able to compile the modified Prolog module as an object.

Modules using a term-expansion mechanism

Although Logtalk supports *term and goal expansion mechanisms*, the usage semantics are different from similar mechanisms found in some Prolog compilers. In particular, Logtalk does not support defining term and goal expansions clauses in a source file for expanding the source file itself. Logtalk forces a clean separation between expansion clauses and the source files that will be subject to source-to-source expansions by using *hook objects*. But hook objects also provide a working solution here when the expansion code is separated from the code to be expanded. Logtalk supports using a module as a hook object as long as its name doesn't coincide with the name of an object and that the module uses `term_expansion/2` and `goal_expansion/2` predicates. Assuming that's the case, before attempting to compile the modules as objects, set the default hook object to the module containing the expansion code. For example, if the expansions are stored in a `system` module:

```
| ?- set_logtalk_flag(hook, system).
...
```

This, however, may not be enough, as expansions may be stored in multiple modules. A common example is to use a module named `prolog` for system expansions and to store the user-defined expansions in `user`. The Logtalk library provides a solution for these scenarios. Using the `hook_flows` library we can select multiple hook objects or hook modules. For example, assuming expansions stored on both `user` and `system` modules:

```
| ?- logtalk_load(hook_flows(loader)).
...

| ?- set_logtalk_flag(hook, hook_set([user, system])).
...
```

After these queries, we can try to compile the modules and look for other porting or portability issues. A well-know issue is Prolog module term-expansions calling predicates such as `prolog_load_context/2`, which will always fail when it's the Logtalk compiler instead of the Prolog compiler loading a source file. In some of these cases, it may be possible to rewrite the expansion rules to use the *logtalk_load_context/2* predicate instead.

File search paths

Some Prolog systems provide a mechanism for defining file search paths (this mechanism works differently from Logtalk own support for defining library path aliases). When porting Prolog code that defines file search paths, e.g. for finding module libraries, it often helps to load the pristine Prolog application before attempting to compile its source files as Logtalk source files. Depending on the Prolog backend, this may allow the file search paths to be used when compiling modules as objects that use file directives such as `use_module/2`.

1.21.5 Dealing with proprietary Prolog directives and predicates

Most Prolog compilers define proprietary, non-standard directives and predicates that may be used in both plain code and module code. Non-standard Prolog built-in predicates are usually not problematic, as Logtalk is usually able to identify and compile them correctly (but see the notes on built-in meta-predicates for possible caveats). However, Logtalk will generate compilation errors on source files containing proprietary directives unless you first specify how the directives should be handled. Several actions are possible on a per-directive basis: ignoring the directive (i.e., do not copy the directive, although a goal can be proved as a consequence), rewriting and copying the directive to the generated Prolog files, or rewriting and re-compiling the resulting directive. To specify these actions, the adapter files contain clauses for the internal `'$lgt_prolog_term_expansion'/2` predicate. For example, assume that a given Prolog compiler defines a `comment/2` directive for predicates using the format:

```
:- comment(foo/2, "Brief description of the predicate").
```

We can rewrite this predicate into a Logtalk `info/2` directive by defining a suitable clause for the `'$lgt_prolog_term_expansion'/2` predicate:

```
'$lgt_prolog_term_expansion'(
        (:- comment(F/A, String)),
        (:- info(F/A, [comment is Atom]))
) :-
    atom_codes(Atom, String).
```

This Logtalk feature can be used to allow compilation of legacy Prolog code without the need of changing the sources. When used, it is advisable to set the *portability* compiler flag to `warning` in order to more easily identify source files that are likely non-portable across Prolog compilers.

A second example, where a proprietary Prolog directive is discarded after triggering a side effect:

```
'$lgt_prolog_term_expansion'(
        (:- load_foreign_files(Files,Libs,InitRoutine)),
        []
) :-
    load_foreign_files(Files,Libs,InitRoutine).
```

In this case, although the directive is not copied to the generated Prolog file, the foreign library files are loaded as a side effect of the Logtalk compiler calling the `'$lgt_prolog_term_expansion'/2` hook predicate.

1.21.6 Calling Prolog module predicates

Prolog module predicates can be called from within objects or categories by simply using explicit module qualification, i.e. by writing `Module:Goal` or `Goal@Module` (depending on the module system). Logtalk also supports the use of `use_module/2` directives in objects and categories (with the restriction that the first argument of the directive must be the actual module name and not the module file name or the module file path). In this case, these directives are parsed in a similar way to Logtalk *uses/2* directives, with calls to the specified module predicates being automatically translated to `Module:Goal` calls.

As a general rule, the Prolog modules should be loaded (e.g., in the auxiliary Logtalk loader files) *before* compiling objects that make use of module predicates. Moreover, the Logtalk compiler does not generate code for the automatic loading of modules referenced in `use_module/1-2` directives. This is a consequence of the lack of standardization of these directives, whose first argument can be a module name, a straight file name, or a file name using some kind of library notation, depending on the *backend Prolog compiler*. Worse, modules are sometimes defined in files with names different from the module names, requiring finding, opening, and reading the file in order to find the actual module name.

Logtalk allows you to send a message to a module in order to call one of its predicates. This is usually not advised as it implies a performance penalty when compared to just using the `Module:Call` notation. Moreover, this works only if there is no object with the same name as the module you are targeting. This feature is necessary, however, in order to properly support the compilation of modules containing `use_module/2` directives as objects. If the modules specified in the `use_module/2` directives are not compiled as objects but are instead loaded as-is by Prolog, the exported predicates would need to be called using the `Module:Call` notation but the converted module will be calling them through message-sending. Thus, this feature ensures that, on a module compiled as an object, any predicate calling other module predicates will work as expected, either these other modules are loaded as-is or also compiled as objects.

For more details, see the *Calling Prolog predicates* section.

1.21.7 Loading converted Prolog applications

Logtalk strongly favors and advises users to provide a main *loader file* for applications that explicitly load any required libraries and the application source files. In contrast, Prolog applications often either scatter loading of source files from multiple files or use implicit loading of source files via `use_module/1-2` directives. Due to this frequent ad-hoc approach, it's common to find Prolog applications with duplicated loading directives, and where loading order ignores the dependencies between source files. These issues are easily exposed by the Logtalk linter when compiling Prolog files as Logtalk files. Also common are Prolog files with multiple circular dependencies. While this should not affect the *semantics* of the ported code, it may cause some performance penalties as it prevents the Logtalk compiler from optimizing the message sending goals using

static-binding. It also makes the application architecture more difficult to understand. The definition of explicit loader files provides a good opportunity for sorting out loading order and circular dependencies, with the linter warnings providing hints for possible code refactoring to eliminate these issues. The *diagrams* tool supports directory and file loading and dependency diagrams that are also useful in understanding applications architecture.

CHAPTER

TWO

REFERENCE MANUAL

2.1 Grammar

The Logtalk grammar is here described using W3C-style Extended Backus-Naur Form syntax. Non-terminal symbols not defined here can be found in the ISO Prolog Core standard. Terminal symbols are represented between double-quotes.

2.1.1 Entities

```
entity ::=
   object
   | category
   | protocol
```

2.1.2 Object definition

```
object ::=
   begin_object_directive ( object_term )* end_object_directive

begin_object_directive ::=
   ":- object(" object_identifier ( "," object_relations )? ")."

end_object_directive ::=
   ":- end_object."

object_relations ::=
   prototype_relations
   | non_prototype_relations

prototype_relations ::=
   prototype_relation
   | prototype_relation "," prototype_relations

prototype_relation ::=
   implements_protocols
   | imports_categories
   | extends_objects
```

(continues on next page)

(continued from previous page)

```
non_prototype_relations ::=
   non_prototype_relation
   | non_prototype_relation "," non_prototype_relations

non_prototype_relation ::=
   implements_protocols
   | imports_categories
   | instantiates_classes
   | specializes_classes
```

2.1.3 Category definition

```
category ::=
   begin_category_directive  ( category_term )* end_category_directive

begin_category_directive ::=
   ":- category(" category_identifier ( "," category_relations )? ")."

end_category_directive ::=
   ":- end_category."

category_relations ::=
   category_relation
   | category_relation "," category_relations

category_relation ::=
   implements_protocols
   | extends_categories
   | complements_objects
```

2.1.4 Protocol definition

```
protocol ::=
   begin_protocol_directive  ( protocol_directive )* end_protocol_directive

begin_protocol_directive ::=
   ":- protocol(" protocol_identifier ( "," extends_protocols)? ")."

end_protocol_directive ::=
   ":- end_protocol."
```

2.1.5 Entity relations

```
extends_protocols ::=
   "extends(" extended_protocols ")"

extends_objects ::=
   "extends(" extended_objects ")"

extends_categories ::=
   "extends(" extended_categories ")"

implements_protocols ::=
   "implements(" implemented_protocols ")"

imports_categories ::=
   "imports(" imported_categories ")"

instantiates_classes ::=
   "instantiates(" instantiated_objects ")"

specializes_classes ::=
   "specializes(" specialized_objects ")"

complements_objects ::=
   "complements(" complemented_objects ")"
```

Implemented protocols

```
implemented_protocols ::=
   implemented_protocol
   | implemented_protocol_sequence
   | implemented_protocol_list

implemented_protocol ::=
   protocol_identifier
   | scope "::" protocol_identifier

implemented_protocol_sequence ::=
   implemented_protocol
   | implemented_protocol "," implemented_protocol_sequence

implemented_protocol_list ::=
   "[" implemented_protocol_sequence "]"
```

Extended protocols

```
extended_protocols ::=
   extended_protocol
   | extended_protocol_sequence
   | extended_protocol_list

extended_protocol ::=
   protocol_identifier
   | scope "::" protocol_identifier

extended_protocol_sequence ::=
   extended_protocol
   |extended_protocol "," extended_protocol_sequence

extended_protocol_list ::=
   "[" extended_protocol_sequence "]"
```

Imported categories

```
imported_categories ::=
   imported_category
   | imported_category_sequence
   | imported_category_list

imported_category ::=
   category_identifier
   | scope "::" category_identifier

imported_category_sequence ::=
   imported_category
   | imported_category "," imported_category_sequence

imported_category_list ::=
   "[" imported_category_sequence "]"
```

Extended objects

```
extended_objects ::=
   extended_object
   | extended_object_sequence
   | extended_object_list

extended_object ::=
   object_identifier
   | scope "::" object_identifier

extended_object_sequence ::=
   extended_object
   | extended_object "," extended_object_sequence
```

(continues on next page)

(continued from previous page)

```
extended_object_list ::=
   "[" extended_object_sequence "]"
```

Extended categories

```
extended_categories ::=
   extended_category
   | extended_category_sequence
   | extended_category_list

extended_category ::=
   category_identifier
   | scope "::" category_identifier

extended_category_sequence ::=
   extended_category
   | extended_category "," extended_category_sequence

extended_category_list ::=
   "[" extended_category_sequence "]"
```

Instantiated objects

```
instantiated_objects ::=
   instantiated_object
   | instantiated_object_sequence
   | instantiated_object_list

instantiated_object ::=
   object_identifier
   | scope "::" object_identifier

instantiated_object_sequence ::=
   instantiated_object
   | instantiated_object "," instantiated_object_sequence

instantiated_object_list ::=
   "[" instantiated_object_sequence "]"
```

Specialized objects

```
specialized_objects ::=
   specialized_object
   | specialized_object_sequence
   | specialized_object_list

specialized_object ::=
   object_identifier
   | scope "::" object_identifier

specialized_object_sequence ::=
   specialized_object
   | specialized_object "," specialized_object_sequence

specialized_object_list ::=
   "[" specialized_object_sequence "]"
```

Complemented objects

```
complemented_objects ::=
   object_identifier
   | complemented_object_sequence
   | complemented_object_list

complemented_object_sequence ::=
   object_identifier
   | object_identifier "," complemented_object_sequence

complemented_object_list ::=
   "[" complemented_object_sequence "]"
```

Entity and predicate scope

```
scope ::=
   "public"
   | "protected"
   | "private"
```

2.1.6 Entity identifiers

```
entity_identifier ::=
   object_identifier
   | protocol_identifier
   | category_identifier
```

Object identifiers

```
object_identifier ::=
   atom
   | compound
```

Category identifiers

```
category_identifier ::=
   atom
   | compound
```

Protocol identifiers

```
protocol_identifier ::=
   atom
```

Module identifiers

```
module_identifier ::=
   atom
```

2.1.7 Source files

```
source_file ::=
   ( source_file_content )*

source_file_content ::=
   source_file_directive
   | clause
   | grammar_rule
   | entity
```

2.1.8 Source file names

```
source_file_name ::=
   atom
   | library_source_file_name

library_source_file_name ::=
   library_name "(" atom ")"

library_name ::=
   atom
```

2.1.9 Terms

Object terms

```
object_term ::=
   object_directive
   | clause
   | grammar_rule
```

Category terms

```
category_term ::=
   category_directive
   | clause
   | grammar_rule
```

2.1.10 Directives

Source file directives

```
source_file_directive ::=
   ":- encoding(" atom ")."
   | ":- set_logtalk_flag(" atom "," nonvar ")."
   | ":- include(" source_file_name ")."
   | prolog_directive
```

Conditional compilation directives

```
conditional_compilation_directive ::=
   ":- if(" callable ")."
   | ":- elif(" callable ")."
   | ":- else."
   | ":- endif."
```

Object directives

```
object_directive ::=
   ":- initialization(" callable ")."
   | ":- built_in."
   | ":- threaded."
   | ":- dynamic."
   | ":- info(" entity_info_list ")."
   | ":- set_logtalk_flag(" atom "," nonvar ")."
   | ":- include(" source_file_name ")."
   | ":- uses(" object_alias_list ")."
   | ":- use_module(" module_alias_list ")."
```

(continues on next page)

(continued from previous page)

```
   | conditional_compilation_directive
   | predicate_directive
```

Category directives

```
category_directive ::=
   ":- built_in."
   | ":- dynamic."
   | ":- info(" entity_info_list ")."
   | ":- set_logtalk_flag(" atom "," nonvar ")."
   | ":- include(" source_file_name ")."
   | ":- uses(" object_alias_list ")."
   | ":- use_module(" module_alias_list ")."
   | conditional_compilation_directive
   | predicate_directive
```

Protocol directives

```
protocol_directive ::=
   ":- built_in."
   | ":- dynamic."
   | ":- info(" entity_info_list ")."
   | ":- set_logtalk_flag(" atom "," nonvar ")."
   | ":- include(" source_file_name ")."
   | conditional_compilation_directive
   | predicate_directive
```

Predicate directives

```
predicate_directive ::=
   alias_directive
   | synchronized_directive
   | uses_directive
   | use_module_directive
   | scope_directive
   | mode_directive
   | meta_predicate_directive
   | meta_non_terminal_directive
   | info_directive
   | dynamic_directive
   | discontiguous_directive
   | multifile_directive
   | coinductive_directive
   | operator_directive

alias_directive ::=
   ":- alias(" entity_identifier "," alias_directive_resource_list ")."
```

(continues on next page)

(continued from previous page)

```
synchronized_directive ::=
   ":- synchronized(" synchronized_directive_resource_term ")."

uses_directive ::=
   ":- uses(" ( object_identifier | parameter_variable ) "," uses_directive_resource_list ").
↪"

use_module_directive ::=
   ":- use_module(" ( module_identifier | parameter_variable ) "," use_module_directive_
↪resource_list ")."

scope_directive ::=
   ":- public(" scope_directive_resource_term ")."
   | ":- protected(" scope_directive_resource_term ")."
   | ":- private(" scope_directive_resource_term ")."

mode_directive ::=
   ":- mode(" ( predicate_mode_term | non_terminal_mode_term ) "," number_of_proofs ")."

meta_predicate_directive ::=
   ":- meta_predicate(" meta_predicate_template_term ")."

meta_non_terminal_directive ::=
   ":- meta_non_terminal(" meta_non_terminal_template_term ")."

info_directive ::=
   ":- info(" ( predicate_indicator | non_terminal_indicator ) "," predicate_info_list ")."

dynamic_directive ::=
   ":- dynamic(" qualified_directive_resource_term ")."

discontiguous_directive ::=
   ":- discontiguous(" qualified_directive_resource_term ")."

multifile_directive ::=
   ":- multifile(" qualified_directive_resource_term ")."

coinductive_directive ::=
   ":- coinductive(" ( predicate_indicator_term | coinductive_predicate_template_term ) ")."

parameter_variable ::=
   _variable_

scope_directive_resource_term ::=
   scope_directive_resource
   | scope_directive_resource_sequence
   | scope_directive_resource_list

scope_directive_resource ::=
   predicate_indicator
   | non_terminal_indicator
```

(continues on next page)

(continued from previous page)

```
   | operator

scope_directive_resource_sequence ::=
   scope_directive_resource
   | scope_directive_resource "," scope_directive_resource_sequence

scope_directive_resource_list ::=
   "[" scope_directive_resource_sequence "]"

entity_resources_list ::=
   predicate_indicator_list
   | operator_list

predicate_indicator_term ::=
   predicate_indicator
   | predicate_indicator_sequence
   | predicate_indicator_list

predicate_indicator_sequence ::=
   predicate_indicator
   | predicate_indicator "," predicate_indicator_sequence

predicate_indicator_list ::=
   "[" predicate_indicator_sequence "]"

alias_directive_resource_list ::=
   "[" alias_directive_resource_sequence "]"

alias_directive_resource_sequence ::=
   alias_directive_resource
   | alias_directive_resource "," alias_directive_resource_sequence

alias_directive_resource ::=
   predicate_indicator_alias
   | non_terminal_indicator_alias

synchronized_directive_resource_term ::=
   synchronized_directive_resource
   | synchronized_directive_resource_sequence
   | synchronized_directive_resource_list

synchronized_directive_resource ::=
   predicate_indicator
   | non_terminal_indicator

synchronized_directive_resource_sequence ::=
   synchronized_directive_resource
   | synchronized_directive_resource "," synchronized_directive_resource_sequence

synchronized_directive_resource_list ::=
   "[" synchronized_directive_resource_sequence "]"
```

(continues on next page)

(continued from previous page)

```
uses_directive_resource_list ::=
   "[" uses_directive_resource_sequence "]"

uses_directive_resource_sequence ::=
   uses_directive_resource
   | uses_directive_resource "," uses_directive_resource_sequence

uses_directive_resource ::=
   predicate_indicator
   | non_terminal_indicator
   | predicate_template_alias
   | operator

use_module_directive_resource_list ::=
   "[" use_module_directive_resource_sequence "]"

use_module_directive_resource_sequence ::=
   use_module_directive_resource
   | use_module_directive_resource "," use_module_directive_resource_sequence

use_module_directive_resource ::=
   predicate_indicator
   | non_terminal_indicator
   | predicate_template_alias
   | operator

qualified_directive_resource_term ::=
   qualified_directive_resource
   | qualified_directive_resource_sequence
   | qualified_directive_resource_list

qualified_directive_resource_sequence ::=
   qualified_directive_resource
   | qualified_directive_resource "," qualified_directive_resource_sequence

qualified_directive_resource_list ::=
   "[" qualified_directive_resource_sequence "]"

qualified_directive_resource ::=
   predicate_indicator
   | non_terminal_indicator
   | object_identifier "::" ( predicate_indicator | non_terminal_indicator)
   | category_identifier "::" ( predicate_indicator | non_terminal_indicator)
   | module_identifier ":" ( predicate_indicator | non_terminal_indicator)

predicate_indicator_alias ::=
   predicate_indicator "as" predicate_indicator

predicate_template_alias ::=
   callable "as" callable

non_terminal_indicator ::=
```

(continues on next page)

(continued from previous page)

```
   functor "//" arity

non_terminal_indicator_alias ::=
   non_terminal_indicator "as" non_terminal_indicator

operator_sequence ::=
   operator specification
   | operator specification "," operator_sequence

operator_list ::=
   "[" operator_sequence "]"

coinductive_predicate_template_term ::=
   coinductive_predicate_template
   | coinductive_predicate_template_sequence
   | coinductive_predicate_template_list

coinductive_predicate_template_sequence ::=
   coinductive_predicate_template
   | coinductive_predicate_template "," coinductive_predicate_template_sequence

coinductive_predicate_template_list ::=
   "[" coinductive_predicate_template_sequence "]"

coinductive_predicate_template ::=
   atom "(" coinductive_mode_terms ")"

coinductive_mode_terms ::=
   coinductive_mode_term
   | coinductive_mode_terms "," coinductive_mode_terms

coinductive_mode_term ::=
   "+"
   | "-"

predicate_mode_term ::=
   atom "(" mode_terms ")"

non_terminal_mode_term ::=
   atom "(" mode_terms ")"

mode_terms ::=
   mode_term
   |mode_term "," mode_terms

mode_term ::=
   "@"  type?
   | "+" type?
   | "-" type?
   | "?" type?
   | "++" type?
   | "--" type?
```

(continues on next page)

(continued from previous page)

```
type ::=
   prolog_type | logtalk_type | user_defined_type

prolog_type ::=
   "term"
   | "nonvar"
   | "var"
   | "compound"
   | "ground"
   | "callable"
   | "list"
   | "atomic"
   | "atom"
   | "number"
   | "integer"
   | "float"

logtalk_type ::=
   "object"
   | "category"
   | "protocol"
   | "event"

user_defined_type ::=
   atom
   | compound

number_of_proofs ::=
   "zero"
   | "zero_or_one"
   | "zero_or_more"
   | "one"
   | "one_or_more"
   | "zero_or_error"
   | "one_or_error"
   | "zero_or_one_or_error"
   | "zero_or_more_or_error"
   | "one_or_more_or_error"
   | "error"

meta_predicate_template_term ::=
   meta_predicate_template
   | meta_predicate_template_sequence
   | meta_predicate_template_list

meta_predicate_template_sequence ::=
   meta_predicate_template
   | meta_predicate_template "," meta_predicate_template_sequence

meta_predicate_template_list ::=
   "[" meta_predicate_template_sequence "]"
```

(continues on next page)

(continued from previous page)

```
meta_predicate_template ::=
   object_identifier "::" atom "(" meta_predicate_specifiers ")"
   | category_identifier "::" atom "(" meta_predicate_specifiers ")"
   | module_identifier ":" atom "(" meta_predicate_specifiers ")"
   | atom "(" meta_predicate_specifiers ")"

meta_predicate_specifiers ::=
   meta_predicate_specifier
   | meta_predicate_specifier "," meta_predicate_specifiers

meta_predicate_specifier ::=
   non_negative_integer
   | "::"
   | "^"
   | "*"

meta_non_terminal_template_term ::=
   meta_predicate_template_term

entity_info_list ::=
   "[" entity_info_sequence? "]"

entity_info_sequence ::=
   entity_info_item "is" nonvar
   | entity_info_item "is" nonvar "," entity_info_sequence

entity_info_item ::=
   "comment"
   | "remarks"
   | "author"
   | "version"
   | "date"
   | "copyright"
   | "license"
   | "parameters"
   | "parnames"
   | "see_also"
   | atom

predicate_info_list ::=
   "[" predicate_info_sequence? "]"

predicate_info_sequence ::=
   predicate_info_item "is" nonvar
   | predicate_info_item "is" nonvar "," predicate_info_sequence

predicate_info_item ::=
   "comment"
   | "remarks"
   | "arguments"
   | "argnames"
```

(continues on next page)

(continued from previous page)

```
   | "redefinition"
   | "allocation"
   | "examples"
   | "exceptions"
   | "see_also"
   | atom

object_alias_list ::=
   "[" object_alias_sequence "]"

object_alias_sequence ::=
   object_alias
   | object_alias "," object_alias_sequence

object_alias ::=
   object_identifier "as" object_identifier

module_alias_list ::=
   "[" module_alias_sequence "]"

module_alias_sequence ::=
   module_alias
   | module_alias "," module_alias_sequence

module_alias ::=
   module_identifier "as" module_identifier
```

2.1.11 Clauses and goals

```
clause ::=
   object_identifier "::" head ":-" body
   | module_identifier ":" head ":-" body
   | head ":-" body
   | object_identifier "::" fact
   | module_identifier ":" fact
   | fact

goal ::=
   message_sending
   | super_call
   | external_call
   | context_switching_call
   | callable

message_sending ::=
   message_to_object
   | message_delegation
   | message_to_self

message_to_object ::=
```

(continues on next page)

(continued from previous page)

```
   receiver "::" messages

message_delegation ::=
   "[" message_to_object "]"

message_to_self ::=
   "::" messages

super_call ::=
   "^^" message

messages ::=
   message
   | "(" message "," messages ")"
   | "(" message ";" messages ")"
   | "(" message "->" messages ")"

message ::=
   callable
   | variable

receiver ::=
   "{" callable "}"
   | object_identifier
   | variable

external_call ::=
   "{" callable "}"

context_switching_call ::=
   object_identifier "<<" callable
```

2.1.12 Lambda expressions

```
lambda_expression ::=
   lambda_free_variables "/" lambda_parameters ">>" callable
   | lambda_free_variables "/" callable
   | lambda_parameters ">>" callable

lambda_free_variables ::=
   "{" variables? "}"

lambda_parameters ::=
   "[" terms? "]"

variables ::=
   variable
   | variable "," variables

terms ::=
```

(continues on next page)

(continued from previous page)

```
   term
   | term "," terms
```

2.1.13 Entity properties

```
category_property ::=
   "static"
   | "dynamic"
   | "built_in"
   | "file(" atom ")"
   | "file(" atom "," atom ")"
   | "lines(" integer "," integer ")"
   | "directive(" integer "," integer ")"
   | "events"
   | "source_data"
   | "public(" entity_resources_list ")"
   | "protected(" entity_resources_list ")"
   | "private(" entity_resources_list ")"
   | "declares(" predicate_indicator "," predicate_declaration_property_list ")"
   | "defines(" predicate_indicator "," predicate_definition_property_list ")"
   | "includes(" predicate_indicator "," ( object_identifier | category_identifier ) ","␣
↪predicate_definition_property_list ")"
   | "provides(" predicate_indicator "," ( object_identifier | category_identifier ) ","␣
↪predicate_definition_property_list ")"
   | "alias(" ( object_identifier | module_identifier ) "," entity_alias_property_list ")"
   | "alias(" predicate_indicator "," predicate_alias_property_list ")"
   | "calls(" predicate "," predicate_call_update_property_list ")"
   | "updates(" predicate "," predicate_call_update_property_list ")"
   | "number_of_clauses(" integer ")"
   | "number_of_rules(" integer ")"
   | "number_of_user_clauses(" integer ")"
   | "number_of_user_rules(" integer ")"
   | "debugging"

object_property ::=
   "static"
   | "dynamic"
   | "built_in"
   | "threaded"
   | "file(" atom ")"
   | "file(" atom "," atom ")"
   | "lines(" integer "," integer ")"
   | "directive(" integer "," integer ")"
   | "context_switching_calls"
   | "dynamic_declarations"
   | "events"
   | "source_data"
   | "complements(" ( "allow" | "restrict" ) ")"
   | "complements"
   | "public(" entity_resources_list ")"
```

(continues on next page)

(continued from previous page)

```
   | "protected(" entity_resources_list ")"
   | "private(" entity_resources_list ")"
   | "declares(" predicate_indicator "," predicate_declaration_property_list ")"
   | "defines(" predicate_indicator "," predicate_definition_property_list ")"
   | "includes(" predicate_indicator "," ( object_identifier | category_identifier ) ","␣
↪predicate_definition_property_list ")"
   | "provides(" predicate_indicator "," ( object_identifier | category_identifier ) ","␣
↪predicate_definition_property_list ")"
   | "alias(" ( object_identifier | module_identifier ) "," entity_alias_property_list ")"
   | "alias(" predicate_indicator "," predicate_alias_property_list ")"
   | "calls(" predicate "," predicate_call_update_property_list ")"
   | "updates(" predicate "," predicate_call_update_property_list ")"
   | "number_of_clauses(" integer ")"
   | "number_of_rules(" integer ")"
   | "number_of_user_clauses(" integer ")"
   | "number_of_user_rules(" integer ")"
   | "module"
   | "debugging"

protocol_property ::=
   "static"
   | "dynamic"
   | "built_in"
   | "source_data"
   | "file(" atom ")"
   | "file(" atom "," atom ")"
   | "lines(" integer "," integer ")"
   | "directive(" integer "," integer ")"
   | "public(" entity_resources_list ")"
   | "protected(" entity_resources_list ")"
   | "private(" entity_resources_list ")"
   | "declares(" predicate_indicator "," predicate_declaration_property_list ")"
   | "alias(" predicate_indicator "," predicate_alias_property_list ")"
   | "debugging"

predicate_declaration_property_list ::=
   "[" predicate_declaration_property_sequence "]"

predicate_declaration_property_sequence ::=
   predicate_declaration_property
   | predicate_declaration_property "," predicate_declaration_property_sequence

predicate_declaration_property ::=
   "static"
   | "dynamic"
   | "scope(" scope ")"
   | "private"
   | "protected"
   | "public"
   | "coinductive"
   | "multifile"
   | "synchronized"
```

(continues on next page)

(continued from previous page)

```
   | "meta_predicate(" meta_predicate_template ")"
   | "coinductive(" coinductive_predicate_template ")"
   | "non_terminal(" non_terminal_indicator ")"
   | "include(" atom ")"
   | "lines(" integer  "," integer ")"
   | "line_count(" integer ")"
   | "mode(" ( predicate_mode_term | non_terminal_mode_term ) "," number_of_proofs ")"
   | "info(" list ")"

predicate_definition_property_list ::=
   "[" predicate_definition_property_sequence "]"

predicate_definition_property_sequence ::=
   predicate_definition_property
   | predicate_definition_property "," predicate_definition_property_sequence

predicate_definition_property ::=
   "inline"
   | "auxiliary"
   | "non_terminal(" non_terminal_indicator ")"
   | "include(" atom ")"
   | "lines(" integer  "," integer ")"
   | "line_count(" integer ")"
   | "number_of_clauses(" integer ")"
   | "number_of_rules(" integer ")"

entity_alias_property_list ::=
   "[" entity_alias_property_sequence "]"

entity_alias_property_sequence ::=
   entity_alias_property
   | entity_alias_property "," entity_alias_property_sequence

entity_alias_property ::=
   "object"
   | "module"
   | "for(" ( object_identifier | module_identifier ) ")"
   | "include(" atom ")"
   | "lines(" integer  "," integer ")"
   | "line_count(" integer ")"

predicate_alias_property_list ::=
   "[" predicate_alias_property_sequence "]"

predicate_alias_property_sequence ::=
   predicate_alias_property
   | predicate_alias_property "," predicate_alias_property_sequence

predicate_alias_property ::=
   "predicate"
   | "for(" predicate_indicator ")"
   | "from(" entity_identifier ")"
```

(continues on next page)

(continued from previous page)

```
   | "non_terminal(" non_terminal_indicator ")"
   | "include(" atom ")"
   | "lines(" integer  "," integer ")"
   | "line_count(" integer ")"

predicate ::=
   predicate_indicator
   | "^^" predicate_indicator
   | "::" predicate_indicator
   | ( variable | object_identifier ) "::" predicate_indicator
   | ( variable | module_identifier ) ":" predicate_indicator

predicate_call_update_property_list ::=
   "[" predicate_call_update_property_sequence "]"

predicate_call_update_property_sequence ::=
   predicate_call_update_property
   | predicate_call_update_property "," predicate_call_update_property_sequence

predicate_call_update_property ::=
   "caller(" predicate_indicator ")"
   | "include(" atom ")"
   | "lines(" integer  "," integer ")"
   | "line_count(" integer ")"
   | "alias(" predicate_indicator ")"
   | "non_terminal(" non_terminal_indicator ")"
```

2.1.14 Predicate properties

```
predicate_property ::=
   "static"
   | "dynamic"
   | "scope(" scope ")"
   | "private"
   | "protected"
   | "public"
   | "logtalk"
   | "prolog"
   | "foreign"
   | "coinductive(" coinductive_predicate_template ")"
   | "multifile"
   | "synchronized"
   | "built_in"
   | "inline"
   | "recursive"
   | "declared_in(" entity_identifier ")"
   | "defined_in(" ( object_identifier | category_identifier ) ")"
   | "redefined_from(" ( object_identifier | category_identifier ) ")"
   | "meta_predicate(" meta_predicate_template ")"
   | "alias_of(" callable ")"
```

(continues on next page)

(continued from previous page)

```
   | "alias_declared_in(" entity_identifier ")"
   | "non_terminal(" non_terminal_indicator ")"
   | "mode(" ( predicate_mode_term | non_terminal_mode_term ) "," number_of_proofs ")"
   | "info(" list ")"
   | "number_of_clauses(" integer ")"
   | "number_of_rules(" integer ")"
   | "declared_in(" entity_identifier "," line_count ")"
   | "defined_in(" ( object_identifier | category_identifier ) "," line_count ")"
   | "redefined_from(" ( object_identifier | category_identifier ) "," line_count ")"
   | "alias_declared_in(" entity_identifier "," line_count ")"

line_count ::=
   integer
```

2.1.15 Compiler flags

```
compiler_flag ::=
   flag "(" flag_value ")"
```

2.2 Control constructs

2.2.1 Message sending

control construct

`(::)/2`

Description

```
Object::Message
{Proxy}::Message
```

Sends a message to an object. The message argument must match a *public* predicate of the receiver object. When the message corresponds to a *protected* or *private* predicate, the call is only valid if the *sender* matches the *predicate scope container*. When the predicate is declared but not defined, the message simply fails (as per the *closed-world assumption*).

When the predicate used to answer the message is a *meta-predicate*, the *calling context* for the predicate meta-arguments is the object sending the message.

The `{Proxy}::Message` syntax allows simplified access to *parametric object proxies*. Its operational semantics is equivalent to the conjunction `(call(Proxy), Proxy::Message)`. I.e. `Proxy` is proved within the context of the pseudo-object *user* and, if successful, the `Proxy` term is used as an object identifier. Exceptions thrown when proving `Proxy` are handled by the `(::)/2` control construct. This control construct supports backtracking over the `{Proxy}` goal.

The lookups for the message declaration and the corresponding method are performed using a depth-first strategy. Depending on the value of the *optimize* flag, these lookups are performed at compile-time whenever

sufficient information is available. When the lookups are performed at runtime, a caching mechanism is used to improve performance for subsequent messages. See the User Manual section on *performance* for details.

Modes and number of proofs

```
+object_identifier::+callable - zero_or_more
{+object_identifier}::+callable - zero_or_more
```

Errors

Either `Object` or `Message` is a variable:
`instantiation_error`
`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
`Message` is neither a variable nor a callable term:
`type_error(callable, Message)`
`Message`, with predicate indicator `Name/Arity`, is declared private:
`permission_error(access, private_predicate, Name/Arity)`
`Message`, with predicate indicator `Name/Arity`, is declared protected:
`permission_error(access, protected_predicate, Name/Arity)`
`Message`, with predicate indicator `Name/Arity`, is not declared:
`existence_error(predicate_declaration, Name/Arity)`
`Object` does not exist:
`existence_error(object, Object)`

`Proxy` is a variable:
`instantiation_error`
`Proxy` is neither a variable nor a callable term:
`type_error(callable, Proxy)`
`Proxy`, with predicate indicator `Name/Arity`, does not exist in the `user` pseudo-object:
`existence_error(procedure, Name/Arity)`

Examples

```
| ?- list::member(X, [1, 2, 3]).

X = 1 ;
X = 2 ;
X = 3
yes
```

See also

(::)/1, *(^ ^)/1*, *[]/1*

control construct

`(::)/1`

Description

```
::Message
```

Sends a message to *self*. Can only be used in the body of a predicate definition. The argument should match a *public* or *protected* predicate of *self*. It may also match a *private* predicate if the predicate is within the scope of the object where the method making the call is defined, if imported from a category, if used from within a category, or when using private inheritance. When the predicate is declared but not defined, the message simply fails (as per the *closed-world assumption*).

When the predicate used to answer the message is a *meta-predicate*, the *calling context* for the predicate meta-arguments is the object sending the message.

The lookups for the message declaration and the corresponding method are performed using a depth-first strategy. A message to *self* necessarily implies the use of *dynamic binding* but a caching mechanism is used to improve performance in subsequent messages. See the User Manual section on *performance* for details.

Modes and number of proofs

```
::+callable - zero_or_more
```

Errors

`Message` is a variable:
`instantiation_error`
`Message` is neither a variable nor a callable term:
`type_error(callable, Message)`
`Message`, with predicate indicator `Name/Arity`, is declared private:
`permission_error(access, private_predicate, Name/Arity)`
`Message`, with predicate indicator `Name/Arity`, is not declared:
`existence_error(predicate_declaration, Name/Arity)`

Examples

```
area(Area) :-
    ::width(Width),
    ::height(Height),
    Area is Width * Height.
```

See also

(::)/2, *(^ ^)/1*, *[]/1*

2.2.2 Message delegation

control construct

`[]/1`

Description

```
[Object::Message]
[{Proxy}::Message]
```

This control construct allows the programmer to send a message to an object while preserving the original sender and meta-call context. It is mainly used in the definition of object handlers for unknown messages. This functionality is usually known as *delegation* but be aware that this is an overloaded word that can mean different things in different object-oriented programming languages.

To prevent the use of this control construct to break object encapsulation, an attempt to delegate a message to the original sender results in an error. The remaining error conditions are the same as the *(::)/2* control construct.

Note that, despite the correct functor for this control construct being (traditionally) `'.'/2`, we refer to it as `[]/1` simply to emphasize that the syntax is a list with a single element.

Modes and number of proofs

```
[+object_identifier::+callable] - zero_or_more
[{+object_identifier}::+callable] - zero_or_more
```

Errors

`Object` is a variable:
`instantiation_error`
Object is neither a variable nor an object identifier:
`type_error(object_identifier, Object)`
`Object` does not exist:
`existence_error(object, Object)`
`Object` and the original *sender* are the same object:
`permission_error(access, object, Sender)`

`Proxy` is a variable:
`instantiation_error`
`Proxy` is neither a variable nor an object identifier:
`type_error(object_identifier, Proxy)`
`Proxy`, with predicate indicator `Name/Arity`, does not exist in the `user` pseudo-object:
`existence_error(procedure, Name/Arity)`

`Message` is a variable:
`instantiation_error`

`Message` is neither a variable nor a callable term:
`type_error(callable, Message)`
`Message`, with predicate indicator `Name/Arity`, is declared private:
`permission_error(access, private_predicate, Name/Arity)`
`Message`, with predicate indicator `Name/Arity`, is declared protected:
`permission_error(access, protected_predicate, Name/Arity)`
`Message`, with predicate indicator `Name/Arity`, is not declared:
`existence_error(predicate_declaration, Name/Arity)`

Examples

```
% delegate unknown messages to the "backup" object:
forward(Message) :-
    [backup::Message].
```

See also

(::)/2, *(::)/1*, *(^ ^)/1*, *forward/1*

2.2.3 Calling imported and inherited predicates

control construct

(^^)/1

Description

```
^^Predicate
```

Calls an imported or inherited predicate definition. The call fails if the predicate is declared but there is no imported or inherited predicate definition (as per the *closed-world assumption*). This control construct may be used within objects or categories in the body of a predicate definition.

This control construct preserves the implicit execution context *self* and *sender* arguments (plus the meta-call context and coinduction stack when applicable) when calling the inherited (or imported) predicate definition.

The lookups for the predicate declaration and the predicate definition are performed using a depth-first strategy. Depending on the value of the *optimize* flag, these lookups are performed at compile-time when the predicate is static and sufficient information is available. When the lookups are performed at runtime, a caching mechanism is used to improve performance in subsequent calls. See the User Manual section on *performance* for details.

When the call is made from within an object, the lookup for the predicate definition starts at the imported categories, if any. If an imported predicate definition is not found, the lookup proceeds to the ancestor objects. Calls from predicates defined in complementing categories lookup inherited definitions as if the calls were made from the complemented object, thus allowing more comprehensive object patching. For other categories, the predicate definition lookup is restricted to the extended categories.

The called predicate should be declared *public* or *protected*. It may also be declared *private* if within the scope of the entity where the method making the call is defined.

This control construct is a generalization of the Smalltalk *super* keyword to take into account Logtalk support for prototypes and categories besides classes.

Modes and number of proofs

```
^^+callable - zero_or_more
```

Errors

`Predicate` is a variable:
    `instantiation_error`
`Predicate` is neither a variable nor a callable term:
    `type_error(callable, Predicate)`
`Predicate`, with predicate indicator `Name/Arity`, is declared private:
    `permission_error(access, private_predicate, Name/Arity)`
`Predicate`, with predicate indicator `Name/Arity`, is not declared:
    `existence_error(predicate_declaration, Name/Arity)`

Examples

```
% specialize the inherited definition
% of the init/0 predicate:
init :-
    assertz(counter(0)),
    ^^init.
```

See also

(::)/2, *(::)/1*, *[]/1*

2.2.4 Calling predicates in *this*

control construct

(@)/1

Description

```
@Predicate
```

Calls a predicate definition in *this*. The argument must be a callable term at compile-time. The predicate must be declared (by a scope directive). This control construct provides access to predicate definitions in *this* from categories. For example, it allows overriding a predicate definition from a complementing category with a new definition that calls goals before and after calling the overridden definition (the overriding definition is sometimes described in other programming languages as an *around method*). When used within an object, it's the same as calling its argument.

Modes and number of proofs

```
@ +callable - zero_or_more
```

Errors

`Predicate`, with predicate indicator `Name/Arity`, is not declared:

```
existence_error(predicate_declaration, Name/Arity)
```

Examples

Assuming an object declaring a `make_sound/0` predicate, define an *around method* in a complementing category:

```
make_sound :-
    write('Started making sound...'), nl,
    @make_sound,
    write('... finished making sound.'), nl.
```

See also

(::)/2, *(::)/1*, *[]/1*

2.2.5 Calling external predicates

control construct

{}/1

Description

```
{Goal}
{Closure}
{Term}
```

This control construct allows the programmer to bypass the Logtalk compiler (including its linter but not its optimizer) in multiple contexts:

- Calling a goal as-is (from within an object or category) in the context of the *user* pseudo-object.
- Extending a closure as-is with the remaining arguments of a *call/2-N* call in order to construct a goal that will be called within the context of the `user` pseudo-object.
- Wrapping a source file term (either a clause or a directive) or a source file goal to bypass the *term-expansion mechanism*.
- Using it in place of an object identifier when sending a message. In this case, its argument is proved as a goal within the context of the `user` pseudo-object with the resulting term being used as an object identifier in the message-sending goal. This feature is mainly used with *parametric objects* when their identifiers correspond to predicates defined in `user`.
- Using it as a message to an object. This is mainly useful when the message is a *conjunction of messages* where some of goals are calls to Prolog built-in predicates.

> **Note**
>
> This control construct is opaque to cuts when used to wrap a goal (thus ensuring the same semantics independently of the argument being bound at compile-time or at runtime).

Modes and number of proofs

```
{+callable} - zero_or_more
```

Errors

`Goal` is a variable:

```
instantiation_error
```

`Goal` is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

`Closure` is a variable:

```
instantiation_error
```

`Closure` is neither a variable nor a callable term:

```
type_error(callable, Closure)
```

`Term` is a variable:

```
instantiation_error
```

`Term` is neither a variable nor a callable term:

```
type_error(callable, Term)
```

Examples

```
% overload the standard (<)/2 operator by
% calling its standard built-in definition:
N1/D1 < N2/D2 :-
    {N1*D2 < N2*D1}.

% call a closure in the context of "user":
call_in_user(F, X, Y, Z) :-
    call({F}, X, Y, Z).

% bypass the compiler for a proprietary backend directive:
{:- load_foreign_resource(file)}.

% use parametric object proxies:
| ?- {circle(Id, Radius, Color)}::area(Area).
...

% use Prolog built-in predicates as messages:
| ?- logtalk::{write('hello world!'), nl}.
hello world!
yes
```

2.2.6 Context switching calls

control construct

(<<)/2

Description

```
Object<<Goal
{Proxy}<<Goal
```

Debugging control construct. Calls a goal within the context of the specified object. The goal is called with the following execution context:

- *sender*, *this*, and *self* values set to the object
- empty meta-call context
- empty coinduction stack

The goal may need to be written between parenthesis to avoid parsing errors due to operator conflicts. This control construct should only be used for debugging or for writing unit tests. This control construct can only be used for objects compiled with the *context_switching_calls* compiler flag set to `allow`. Set this compiler flag to `deny` to disable this control construct and thus prevent using it to break encapsulation when deploying applications.

The `{Proxy}<<Goal` syntax allows simplified access to *parametric object proxies*. Its operational semantics is equivalent to the goal conjunction `(call(Proxy), Proxy<<Goal)`. I.e. `Proxy` is proved within the context of the pseudo-object *user* and, if successful, the goal term is used as a parametric object identifier. Exceptions thrown when proving `Proxy` are handled by the `(<<)/2` control construct. This syntax construct supports backtracking over the `{Proxy}` goal.

Caveat: although the goal argument is fully compiled before calling, some necessary information for the second compiler pass may not be available at runtime.

Modes and number of proofs

```
+object_identifier<<+callable - zero_or_more
{+object_identifier}<<+callable - zero_or_more
```

Errors

`Object` is a variable:

```
instantiation_error
```

`Object` is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Object)
```

`Object` does not contain a local definition for the Goal predicate:

```
existence_error(procedure, Goal)
```

`Object` does not exist:

```
existence_error(object, Object)
```

`Object` was created/compiled with support for context-switching calls turned off:

```
permission_error(access, database, Goal)
```

`Proxy` is a variable:

```
instantiation_error
```

`Proxy` is neither a variable nor an object identifier:

```
type_error(object_identifier, Proxy)
```

The predicate `Proxy` does not exist in the `user` pseudo-object:

```
existence_error(procedure, ProxyFunctor/ProxyArity)
```

`Goal` is a variable:

```
instantiation_error
```

`Goal` is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

Examples

```
% call the member/2 predicate in the
% context of the "list" object:
test(member) :-
    list << member(1, [1]).
```

2.3 Directives

2.3.1 Source file directives

directive

encoding/1

Description

```
encoding(Encoding)
```

Declares the source file text encoding. Requires a *backend Prolog compiler* supporting the chosen encoding. When used, this directive must be the first term in the source file in the first line. This directive is also supported in files included in a main file or in a dynamically created entity using *include/1* directives.

The encoding used in a source file (and, in the case of a Unicode encoding, any BOM present) will be used for the intermediate Prolog file generated by the compiler. Logtalk uses the encoding names specified by IANA. In those cases where a preferred MIME name alias is specified, the alias is used instead. Examples includes `'US-ASCII'`, `'ISO-8859-1'`, `'ISO-8859-2'`, `'ISO-8859-15'`, `'UCS-2'`, `'UCS-2LE'`, `'UCS-2BE'`, `'UTF-8'`, `'UTF-16'`, `'UTF-16LE'`, `'UTF-16BE'`, `'UTF-32'`, `'UTF-32LE'`, `'UTF-32BE'`, `'Shift_JIS'`, and `'EUC-JP'`. When writing portable code that cannot be expressed using ASCII, `'UTF-8'` is the most commonly supported Unicode encoding.

The backend Prolog compiler adapter files define a table that translates between the Logtalk and Prolog specific atoms that represent each supported encoding. The *encoding_directive* read-only flag can be used to find if a backend supports this directive and how.

Template and modes

```
encoding(+atom)
```

Examples

```
:- encoding('UTF-8').
```

directive

include/1

Description

```
include(File)
```

Includes a file contents, which must be valid terms, at the place of occurrence of the directive. The file can be specified as a relative path, an absolute path, or using *library notation* and is expanded as a source file name. Relative paths are interpreted as relative to the path of the file containing the directive. The file extension is optional (the recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*).

When using the *reflection API*, predicates from an included file can be distinguished from predicates from the main file by looking for the `include/1` predicate declaration property or the `include/1` predicate definition property. For the included predicates, the `line_count/1` property stores the term line number in the included file.

This directive can be used as either a source file directive or an entity directive. As an entity directive, it can be used both in entities defined in source files and with the entity creation built-in predicates. In the latter case, the file should be specified using an absolute path or using library notation (which expands to a full path) to avoid a fragile dependency on the current working directory.

Included files may contain an *encoding/1* directive, which may specify the same encoding of the main file or a different encoding.

> **Warning**
>
> When using this directive as an argument in calls to the *create_object/4*, *create_category/4*, and *create_protocol/3* built-in predicates, the objects, categories, and protocols will not be recreated or redefined when the included file(s) are modified and the *logtalk_make/0* predicate or the *logtalk_make/1* (with target `all`) predicates are called.

Template and modes

```
include(@source_file_name)
```

Examples

```
% include the "raw_1.txt" text file found
% on the "data" library directory:
:- include(data('raw_1.txt')).

% include a "factbase.pl" file in the same directory
% of the source file containing the directive:
:- include('factbase.pl').

% include a file given its absolute path:
:- include('/home/me/databases/countries.pl').

% create a wrapper object for a Prolog file using
% library notation to define the file path:
| ?- create_object(cities, [], [public(city/4), include(geo('cities.pl'))], []).
```

directive

initialization/1

Description

```
initialization(Goal)
```

When used within an object, this directive defines a goal to be called after the object has been successfully loaded into memory. When used at a global level within a source file, this directive defines a goal to be called after the source file is successfully compiled and loaded into memory.

The loading context can be accessed from this directive by calling the *logtalk_load_context/2* predicate. Note that the usable loading context keys depend on the directive scope (file or object).

Multiple initialization directives can be used in a source file or in an object. Their goals will be called in the same order as the directives at loading time.

Note

Arbitrary goals cannot be used as directives in source files. Any goal that should be automatically called when a source file is loaded must be wrapped using this directive.

Categories and protocols cannot contain `initialization/1` directives as the initialization goals would lack a complete *execution context* that is only available for objects.

Although technically a global `initialization/1` directive in a source file is a Prolog directive, calls to Logtalk built-in predicates from it are usually compiled to improve portability, improve performance, and provide better support for embedded applications.

Warning

Some backend Prolog compilers declare the atom `initialization` as an operator for a lighter syntax. But this makes the code non-portable and is therefore a practice best avoided.

Template and modes

```
initialization(@callable)
```

Examples

```
% call the init/0 predicate after loading the
% source file containing the directive

:- initialization(init).
```

```
% print a debug message after loading a
% source file defining an object

:- object(log).

    :- initialization(start_date).

    start_date :-
        os::date_time(Year, Month, Day, _, _, _, _),
        logtalk::print_message(debug, my_app, 'Starting date: ~d-~d-~d~n'+[Year,Month,Day]).

:- end_object.
```

See also

logtalk_load_context/2

directive

op/3

Description

```
op(Precedence, Associativity, Operator)
op(Precedence, Associativity, [Operator, ...])
```

Declares operators. Global operators can be declared inside a source file by writing the respective directives before the entity opening directives. Operators declared inside entities have local scope. Calls to the standard term input and output predicates take into account any locally defined operators.

Template and modes

```
op(+integer, +associativity, +atom_or_atom_list)
```

Examples

Some of the predicate argument instantiation mode operators used by Logtalk:

```
:- op(200, fy, +).
:- op(200, fy, ?).
:- op(200, fy, @).
:- op(200, fy, -).
```

An example of using entity local operators. Consider the following `ops.lgt` file:

```
:- initialization((write(<=>(1,2)), nl)).

:- object(ops).

    :- op(700, xfx, <=>).

    :- public(w/1).
    w(Term) :-
        write(Term), nl.

    :- public(r/1).
    r(Term) :-
        read(Term).

:- end_object.
```

Loading the file automatically calls the initialization goal. Compare its output with the output of the `ops::w/1` predicate. Compare also reading a term from within the `ops` object versus reading from `user`.

```
| ?- {ops}.
<=>(1,2)
true.

| ?- ops::w(<=>(1,2)).
1<=>2
true.

| ?- ops::r(T).
|: 3<=>4.

T = <=>(3, 4).

| ?- read(T).
|: 5<=>6.

SYNTAX ERROR: operator expected
```

See also

current_op/3

directive

set_logtalk_flag/2

Description

```
set_logtalk_flag(Flag, Value)
```

Sets local flag values. The scope of this directive is the entity or the source file containing it. For global scope, use the corresponding *set_logtalk_flag/2* built-in predicate called from an *initialization/1* directive. For a description of the predefined compiler flags, consult the *Compiler flags* section in the User Manual. The directive affects the compilation of all terms that follow it within the scope of the directive.

Template and modes

```
set_logtalk_flag(+atom, +nonvar)
```

Errors

Flag is a variable:
    `instantiation_error`
Value is a variable:
    `instantiation_error`
Flag is not an atom:
    `type_error(atom, Flag)`
Flag is neither a variable nor a valid flag:
    `domain_error(flag, Flag)`
Value is not a valid value for flag Flag:
    `domain_error(flag_value, Flag + Value)`
Flag is a read-only flag:
    `permission_error(modify, flag, Flag)`

Examples

```
% turn off the compiler unknown entity warnings
% during the compilation of this source file:
:- set_logtalk_flag(unknown_entities, silent).


:- object(...).

```

(continues on next page)

(continued from previous page)

```
    % generate events for messages sent from this object:
    :- set_logtalk_flag(events, allow).
    ...

    % turn off suspicious call lint checks for the next predicate:
    :- set_logtalk_flag(suspicious_calls, silent).
    foo :-
        ...
    :- set_logtalk_flag(suspicious_calls, warning).
    ...
```

2.3.2 Conditional compilation directives

directive

`if/1`

Description

```
if(Goal)
```

Starts an *if-then* branch when performing conditional compilation. The code following the directive is compiled iff `Goal` is true. If `Goal` throws an error instead of either succeeding or failing, the error is reported by the compiler and compilation of the enclosing source file or entity is aborted. The goal is *expanded* when the directive occurs in a source file. Conditional compilation directives can be nested.

> **Warning**
>
> Conditional compilation goals **cannot** depend on predicate definitions contained in the same source file that contains the conditional compilation directives (as those predicates only become available after the file is successfully compiled and loaded).

Template and modes

```
if(@callable)
```

Examples

A common example is checking if a built-in predicate exists and providing a definition when the predicate is absent:

```
:- if(\+ predicate_property(length(_,_), built_in)).

    length(List, Length) :-
        ...
```

(continues on next page)

(continued from previous page)

```
:- endif.
```

Another common example is conditionally including code for a specific backend Prolog compiler:

```
:- if(current_logtalk_flag(prolog_dialect, swi)).

    % SWI-Prolog specific code
    :- set_prolog_flag(double_quotes, codes).

:- endif.
```

If necessary, test goal errors can be converted into failures using the standard `catch/3` control construct. For example:

```
:- if(catch(\+ log(7,_), _, fail)).

   % define the legacy log/2 predicate
   log(X, Y) :- Y is log(X).

:- endif.
```

See also

elif/1, *else/0*, *endif/0*

directive

elif/1

Description

```
elif(Goal)
```

Supports embedded conditionals when performing conditional compilation. The code following the directive is compiled iff `Goal` is true. If `Goal` throws an error instead of either succeeding or failing, the error is reported by the compiler and compilation of the enclosing source file or entity is aborted. The goal is *expanded* when the directive occurs in a source file.

Warning

Conditional compilation goals **cannot** depend on predicate definitions contained in the same source file that contains the conditional compilation directives (as those predicates only become available after the file is successfully compiled and loaded).

Template and modes

```
elif(@callable)
```

Examples

```
:- if(current_prolog_flag(double_quotes, codes)).

    ...

:- elif(current_prolog_flag(double_quotes, chars)).

    ...

:- elif(current_prolog_flag(double_quotes, atom)).

    ...

:- endif.
```

See also

else/0, *endif/0*, *if/1*

directive

else/0

Description

```
else
```

Starts an *else* branch when performing conditional compilation. The code following this directive is compiled iff the goal in the matching *if/1* or *elif/1* directive is false.

Template and modes

```
else
```

Examples

An example where a hypothetical application would have some limitations that the user should be made aware of when running on a backend Prolog compiler with bounded arithmetic:

```
:- if(current_prolog_flag(bounded, true)).

    :- initialization(
        logtalk::print_message(warning,app,bounded_arithmetic)
    ).

:- else.

    :- initialization(
        logtalk::print_message(comment,app,unbounded_arithmetic)
    ).

:- endif.
```

See also

elif/1, *endif/0*, *if/1*

directive

endif/0

Description

```
endif
```

Ends conditional compilation for the matching *if/1* directive.

Template and modes

```
endif
```

Examples

```
:- if(date::today(_,5,25)).

    :- initialization(write('Happy Towel Day!\n')).

:- endif.
```

See also

elif/1, *else/0*, *if/1*

2.3.3 Entity directives

directive

`built_in/0`

Description

```
built_in
```

Declares an entity as built-in. Built-in entities must be static and cannot be redefined once loaded. This directive is used in the pre-defined protocols, categories, and objects that are automatically loaded at startup.

Template and modes

```
built_in
```

Examples

```
:- built_in.
```

directive

`category/1-4`

Description

```
category(Category)

category(Category,
    implements(Protocols))

category(Category,
    extends(Categories))

category(Category,
    complements(Objects))

category(Category,
    implements(Protocols),
    extends(Categories))
```

(continues on next page)

(continued from previous page)

```
category(Category,
    implements(Protocols),
    complements(Objects))

category(Category,
    extends(Categories),
    complements(Objects))

category(Category,
    implements(Protocols),
    extends(Categories),
    complements(Objects))
```

Starting category directive.

Template and modes

```
category(+category_identifier)

category(+category_identifier,
    implements(+implemented_protocols))

category(+category_identifier,
    extends(+extended_categories))

category(+category_identifier,
    complements(+complemented_objects))

category(+category_identifier,
    implements(+implemented_protocols),
    extends(+extended_categories))

category(+category_identifier,
    implements(+implemented_protocols),
    complements(+complemented_objects))

category(+category_identifier,
    extends(+extended_categories),
    complements(+complemented_objects))

category(+category_identifier,
    implements(+implemented_protocols),
    extends(+extended_categories),
    complements(+complemented_objects))
```

Examples

```
:- category(monitoring).

:- category(monitoring,
    implements(monitoringp)).

:- category(attributes,
    implements(protected::variables)).

:- category(extended,
    extends(minimal)).

:- category(logging,
    implements(monitoring),
    complements(employee)).
```

See also

end_category/0

directive

dynamic/0

Description

```
dynamic
```

Declares an entity and its contents as dynamic. Dynamic entities can be abolished at runtime.

As entities created at runtime (using the *create_object/4*, *create_protocol/3*, and *create_category/4* built-in predicates) are always dynamic, this directive is only necessary in the rare cases where we want to define dynamic entities in source files.

Warning

Some backend Prolog compilers declare the atom `dynamic` as an operator for a lighter syntax, forcing writing this atom between parenthesis when using this directive.

Template and modes

```
dynamic
```

Examples

```
:- dynamic.
```

See also

dynamic/1, *object_property/2*, *protocol_property/2*, *category_property/2*

directive

end_category/0

Description

```
end_category
```

Ending category directive.

Template and modes

```
end_category
```

Examples

```
:- end_category.
```

See also

category/1-4

directive

end_object/0

Description

```
end_object
```

Ending object directive.

Template and modes

```
end_object
```

Examples

```
:- end_object.
```

See also

object/1-5

directive

end_protocol/0

Description

```
end_protocol
```

Ending protocol directive.

Template and modes

```
end_protocol
```

Examples

```
:- end_protocol.
```

See also

protocol/1-2

directive

info/1

Description

```
info([Key is Value, ...])
```

Documentation directive for objects, protocols, and categories. The directive argument is a list of pairs using the format *Key is Value*. See the *Entity documenting directives* section for a description of the default keys.

Template and modes

```
info(+entity_info_list)
```

Examples

```
:- info([
    version is 1:0:0,
    author is 'Paulo Moura',
    date is 2000-11-20,
    comment is 'List protocol.'
]).
```

See also

info/2, *object_property/2*, *protocol_property/2*, *category_property/2*

directive

object/1-5

Description

Stand-alone objects (prototypes)

```
object(Object)

object(Object,
    implements(Protocols))

object(Object,
    imports(Categories))

object(Object,
    implements(Protocols),
    imports(Categories))
```

Prototype extensions

```
object(Object,
    extends(Objects))

object(Object,
    implements(Protocols),
    extends(Objects))

object(Object,
    imports(Categories),
    extends(Objects))

object(Object,
    implements(Protocols),
    imports(Categories),
    extends(Objects))
```

Class instances

```
object(Object,
    instantiates(Classes))

object(Object,
    implements(Protocols),
    instantiates(Classes))

object(Object,
    imports(Categories),
    instantiates(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    instantiates(Classes))
```

Classes

```
object(Object,
    specializes(Classes))

object(Object,
    implements(Protocols),
    specializes(Classes))

object(Object,
    imports(Categories),
    specializes(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    specializes(Classes))
```

Classes with metaclasses

```
object(Object,
    instantiates(Classes),
    specializes(Classes))

object(Object,
    implements(Protocols),
    instantiates(Classes),
    specializes(Classes))

object(Object,
    imports(Categories),
    instantiates(Classes),
    specializes(Classes))

object(Object,
    implements(Protocols),
    imports(Categories),
    instantiates(Classes),
    specializes(Classes))
```

Starting object directive.

Template and modes

Stand-alone objects (prototypes)

```
object(+object_identifier)

object(+object_identifier,
    implements(+implemented_protocols))

object(+object_identifier,
    imports(+imported_categories))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories))
```

Prototype extensions

```
object(+object_identifier,
    extends(+extended_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    extends(+extended_objects))

object(+object_identifier,
    imports(+imported_categories),
    extends(+extended_objects))

object(+object_identifier,
```

(continues on next page)

(continued from previous page)

```
    implements(+implemented_protocols),
    imports(+imported_categories),
    extends(+extended_objects))
```

Class instances

```
object(+object_identifier,
    instantiates(+instantiated_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    instantiates(+instantiated_objects))

object(+object_identifier,
    imports(+imported_categories),
    instantiates(+instantiated_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    instantiates(+instantiated_objects))
```

Classes

```
object(+object_identifier,
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    specializes(+specialized_objects))

object(+object_identifier,
    imports(+imported_categories),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    specializes(+specialized_objects))
```

Class with metaclasses

```
object(+object_identifier,
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))

object(+object_identifier,
    imports(+imported_categories),
```

(continues on next page)

(continued from previous page)

```
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))

object(+object_identifier,
    implements(+implemented_protocols),
    imports(+imported_categories),
    instantiates(+instantiated_objects),
    specializes(+specialized_objects))
```

Examples

```
:- object(list).

:- object(list,
    implements(listp)).

:- object(list,
    extends(compound)).

:- object(list,
    implements(listp),
    extends(compound)).

:- object(object,
    imports(initialization),
    instantiates(class)).

:- object(abstract_class,
    instantiates(class),
    specializes(object)).

:- object(agent,
    imports(private::attributes)).
```

See also

end_object/0

directive

protocol/1-2

Description

```
protocol(Protocol)

protocol(Protocol,
    extends(Protocols))
```

Starting protocol directive.

Template and modes

```
protocol(+protocol_identifier)

protocol(+protocol_identifier,
    extends(+extended_protocols))
```

Examples

```
:- protocol(listp).

:- protocol(listp,
    extends(compoundp)).

:- protocol(queuep,
    extends(protected::listp)).
```

See also

end_protocol/0

directive

threaded/0

Description

```
threaded
```

Declares that an object supports threaded engines, concurrent calls, and asynchronous messages. Any object containing calls to the built-in multi-threading predicates (or importing a category that contains such calls) must include this directive.

This directive results in the automatic creation and set up of an object message queue when the object is loaded or created at runtime. Object message queues are used for exchanging thread notifications and for storing concurrent goal solutions and replies to the multi-threading calls made within the object. The

message queue for the *user* pseudo-object is automatically created at Logtalk startup (provided that multi-threading programming is supported and enabled for the chosen *backend Prolog compiler*).

Note

This directive requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Template and modes

```
threaded
```

Examples

```
:- threaded.
```

See also

synchronized/1, *object_property/2*

directive

uses/1

Description

```
uses([Object as Alias, ...])
```

Declares object aliases. Typically used to shorten long object names, to simplify and consistently send messages to parameterized objects, and to simplify using or experimenting with different object implementations of the same protocol when using explicit message-sending. Object aliases are local to the object (or category) where they are defined.

The objects being aliased can be *parameter variables* or parametric objects where one or more parameters are parameter variables when using the directive in a parametric object or a parametric category defined in a source file (the common case).

Declaring multiple aliases for the same object is allowed. But repeated declarations of the same alias, declaring an alias for an object alias, and redefining an alias to reference a different object are reported as compilation errors.

To enable the use of *static binding*, and thus optimal message-sending performance, the objects should be loaded before compiling the entities that call their predicates.

Template and modes

```
uses(+object_alias_list)
```

Examples

```
:- object(foo(_HeapType_, _OptionsObject_)).

    :- uses([
        fast_random as rnd,
        time(utc) as time,
        heap(_HeapType_) as heap,
        _OptionsObject_ as options
    ]).

    bar :-
        ...,
        % the same as fast_random::permutation(L, P)
        rnd::permutation(L, P),
        % the same as heap(_HeapType_)::as_heap(L, H)
        heap::as_heap(L, H),
        % the same as _OptionsObject_::get(foo, X)
        options::get(foo, X),
        % the same as time(utc)::now(T)
        time::now(T),
        ...
```

See also

uses/2, *use_module/1*, *use_module/2*

directive

use_module/1

Description

```
use_module([Module as Alias, ...])
```

Declares module aliases. Typically used to shorten long module names and to simplify using or experimenting with different module implementations of the same predicates when using explicitly-qualified calls. Module aliases are local to the object (or category) where they are defined.

The modules being aliased can be *parameter variables* when using the directive in a parametric object or a parametric category defined in a source file (the common case).

Declaring multiple aliases for the same module is allowed. But repeated declarations of the same alias, declaring an alias for a module alias, and redefining an alias to reference a different module are reported as compilation errors.

To enable the use of *static binding*, and thus optimal predicate call performance, the modules should be loaded before compiling the entities that call their predicates.

Note that this directive semantics differs from the directive with the same name found on some Prolog implementations where it is used to load a module and import all its exported predicates.

Template and modes

```
use_module(+module_alias_list)
```

Examples

```
:- object(foo(_DataModule_)).

    :- use_module([
        data_noise_handler as cleaner,
        _DataModule_ as data
    ]).

    bar :-
        ...,
        % the same as _DataModule_:xy(X, Y)
        data:xy(X, Y),
        % the same as data_noise_handler:filter(X, Y)
        cleaner:filter(X, Y, Z),
        ...
```

See also

uses/1, *uses/2*, *use_module/2*

2.3.4 Predicate directives

directive

alias/2

Description

```
alias(Entity, [Name/Arity as Alias/Arity, ...])
alias(Entity, [Name//Arity as Alias//Arity, ...])
```

Declares predicate and grammar rule non-terminal *aliases*. A predicate (non-terminal) alias is an alternative name for a predicate (non-terminal) declared or defined in an extended protocol, an implemented protocol, an extended category, an imported category, an extended prototype, an instantiated class, or a specialized class. Predicate aliases may be used to solve conflicts between imported or inherited predicates. It may also be used to give a predicate (non-terminal) a name more appropriate in its usage context. This directive may be used in objects, protocols, and categories.

Predicate (and non-terminal) aliases are specified using (preferably) the notation `Name/Arity as Alias/Arity` or, in alternative, the notation `Name/Arity::Alias/Arity`.

It is also possible to declare predicate and grammar rule non-terminal aliases in implicit qualification directives for sending messages to objects and calling module predicates.

Template and modes

```
alias(@entity_identifier, +list(predicate_indicator_alias))
alias(@entity_identifier, +list(non_terminal_indicator_alias))
```

Examples

```
% resolve a predicate name conflict:
:- alias(list, [member/2 as list_member/2]).
:- alias(set,  [member/2 as set_member/2]).

% define an alternative name for a non-terminal:
:- alias(words, [singular//0 as peculiar//0]).
```

See also

uses/2, *use_module/2*, *uses/1*

directive

coinductive/1

Description

```
coinductive(Name/Arity)
coinductive((Name/Arity, ...))
coinductive([Name/Arity, ...])

coinductive(Name//Arity)
coinductive((Name//Arity, ...))
coinductive([Name//Arity, ...])

coinductive(Template)
coinductive((Template1, ...))
coinductive([Template1, ...])
```

This is an **experimental** directive, used for declaring *coinductive* predicates. Requires a *backend Prolog compiler* with minimal support for cyclic terms. The current implementation of coinduction allows the generation of only the *basic cycles* but all valid solutions should be recognized. Use a predicate indicator or a non-terminal indicator as argument when all the coinductive predicate arguments are relevant for coinductive success. Use a template when only some coinductive predicate arguments (represented by a "+") should be considered when testing for coinductive success (represent the arguments that should be disregarded by

a "-"). It's possible to define local *coinductive_success_hook/1-2* predicates that are automatically called with the coinductive predicate term resulting from a successful unification with an ancestor goal as first argument. The second argument, when present, is the coinductive hypothesis (i.e., the ancestor goal) used. These hook predicates can provide an alternative to the use of tabling when defining some coinductive predicates. There is no overhead when these hook predicates are not defined.

This directive must precede any calls to the declared coinductive predicates.

Template and modes

```
coinductive(+predicate_indicator_term)
coinductive(+non_terminal_indicator_term)
coinductive(+coinductive_predicate_template_term)
```

Examples

```
:- coinductive(comember/2).
:- coinductive(ones_and_zeros//0).
:- coinductive(controller(+,+,+,-,-)).
```

See also

coinductive_success_hook/1-2, *predicate_property/2*

directive

discontiguous/1

Description

```
discontiguous(Name/Arity)
discontiguous((Name/Arity, ...))
discontiguous([Name/Arity, ...])

discontiguous(Name//Arity)
discontiguous((Name//Arity, ...))
discontiguous([Name//Arity, ...])
```

Declares *discontiguous* predicates and discontiguous grammar rule non-terminals. The use of this directive should be avoided as not all *backend Prolog compilers* support discontiguous predicates.

Warning

Some backend Prolog compilers declare the atom `discontiguous` as an operator for a lighter syntax. But this makes the code non-portable and is therefore a practice best avoided.

Template and modes

```
discontiguous(+predicate_indicator_term)
discontiguous(+non_terminal_indicator_term)
```

Examples

```
:- discontiguous(counter/1).

:- discontiguous((lives/2, works/2)).

:- discontiguous([db/4, key/2, file/3]).
```

directive

dynamic/1

Description

```
dynamic(Name/Arity)
dynamic((Name/Arity, ...))
dynamic([Name/Arity, ...])

dynamic(Entity::Name/Arity)
dynamic((Entity::Name/Arity, ...))
dynamic([Entity::Name/Arity, ...])

dynamic(Module:Name/Arity)
dynamic((Module:Name/Arity, ...))
dynamic([Module:Name/Arity, ...])

dynamic(Name//Arity)
dynamic((Name//Arity, ...))
dynamic([Name//Arity, ...])

dynamic(Entity::Name//Arity)
dynamic((Entity::Name//Arity, ...))
dynamic([Entity::Name//Arity, ...])

dynamic(Module:Name//Arity)
dynamic((Module:Name//Arity, ...))
dynamic([Module:Name//Arity, ...])
```

Declares *dynamic* predicates and dynamic grammar rule non-terminals. Note that an object can be static and have both static and dynamic predicates/non-terminals. When the dynamic predicates are local to an object, declaring them also as *private predicates* allows the Logtalk compiler to generate optimized code for asserting and retracting predicate clauses. Categories can also contain dynamic predicate directives but cannot contain clauses for dynamic predicates.

The predicate indicators (or non-terminal indicators) can be explicitly qualified with an object, category, or module identifier when the predicates (or non-terminals) are also declared *multifile*.

Note that dynamic predicates cannot be declared synchronized (when necessary, declare the predicates updating the dynamic predicates as *synchronized*).

Warning

Some backend Prolog compilers declare the atom `dynamic` as an operator for a lighter syntax. But this makes the code non-portable and is therefore a practice best avoided.

Template and modes

```
dynamic(+qualified_predicate_indicator_term)
dynamic(+qualified_non_terminal_indicator_term)
```

Examples

```
:- dynamic(counter/1).

:- dynamic((lives/2, works/2)).

:- dynamic([db/4, key/2, file/3]).
```

See also

dynamic/0, *predicate_property/2*

directive

info/2

Description

```
info(Name/Arity, [Key is Value, ...])
info(Name//Arity, [Key is Value, ...])
```

Documentation directive for predicates and grammar rule non-terminals. The first argument is either a predicate indicator or a grammar rule non-terminal indicator. The second argument is a list of pairs using the format *Key is Value*. See the *Predicate documenting directives* section for a description of the default keys.

Template and modes

```
info(+predicate_indicator, +predicate_info_list)
info(+non_terminal_indicator, +predicate_info_list)
```

Examples

```
:- info(empty/1, [
    comment is 'True if the argument is an empty list.',
    argnames is ['List']
]).

:- info(sentence//0, [
    comment is 'Rewrites a sentence into a noun phrase and a verb phrase.'
]).
```

See also

info/1, *mode/2*, *predicate_property/2*

directive

meta_predicate/1

Description

```
meta_predicate(Template)
meta_predicate((Template, ...))
meta_predicate([Template, ...])

meta_predicate(Entity::Template)
meta_predicate((Entity::Template, ...))
meta_predicate([Entity::Template, ...])

meta_predicate(Module:Template)
meta_predicate((Module:Template, ...))
meta_predicate([Module:Template, ...])
```

Declares *meta-predicates*, i.e., predicates that have arguments interpreted as goals, arguments interpreted as *closures* used to construct goals, or arguments with sub-terms that will be interpreted as goals or closures. Meta-arguments are always called in the meta-predicate *calling context*, not in the meta-predicate *definition context*.

Meta-arguments which are goals are represented by the integer `0`. Meta-arguments which are closures are represented by a positive integer, `N`, representing the number of additional arguments that will be appended to the closure in order to construct the corresponding goal (typically by calling the *call/1-N* built-in method). Meta-arguments with sub-terms that will be interpreted as goals or closures are represented by `::`. Meta-arguments that will be called using the `bagof/3` or `setof/3` predicates and that can thus be existentially-qualified are represented by the atom `^`. Normal arguments are represented by the atom `*`.

Logtalk allows the use of this directive to override the original meta-predicate directive. This is sometimes necessary when calling Prolog built-in meta-predicates or Prolog module meta-predicates due to the lack of standardization of the syntax of the meta-predicate templates. Another case is when a meta-predicate directive is missing. The compiler requires access to correct and non-ambiguous meta-predicate templates to correctly compile calls to Prolog meta-predicates.

Warning

Some backend Prolog compilers declare the atom `meta_predicate` as an operator for a lighter syntax. But this makes the code non-portable and is therefore a practice best avoided.

Template and modes

```
meta_predicate(+meta_predicate_template_term)

meta_predicate(+object_identifier::+meta_predicate_template_term)
meta_predicate(+category_identifier::+meta_predicate_template_term)

meta_predicate(+module_identifier:+meta_predicate_template_term)
```

Examples

```
% findall/3 second argument is interpreted as a goal:
:- meta_predicate(findall(*, 0, *)).

% both forall/2 arguments are interpreted as goals:
:- meta_predicate(forall(0, 0)).

% maplist/3 first argument is interpreted as a closure
% that will be expanded to a goal by appending two
% arguments:
:- meta_predicate(maplist(2, *, *)).
```

See also

meta_non_terminal/1, *predicate_property/2*

directive

meta_non_terminal/1

Description

```
meta_non_terminal(Template)
meta_non_terminal((Template, ...))
meta_non_terminal([Template, ...])

meta_non_terminal(Entity::Template)
meta_non_terminal((Entity::Template, ...))
meta_non_terminal([Entity::Template, ...])

meta_non_terminal(Module:Template)
meta_non_terminal((Module:Template, ...))
meta_non_terminal([Module:Template, ...])
```

Declares meta-non-terminals, i.e., non-terminals that have arguments that will be called as non-terminals (or grammar rule bodies). An argument may also be a *closure* instead of a goal if the non-terminal uses the *call//1-N* built-in methods to construct and call the actual non-terminal from the closure and the additional arguments or the *phrase//1* built-in method.

Meta-arguments which are non-terminals are represented by the integer `0`. Meta-arguments which are closures are represented by a positive integer, `N`, representing the number of additional arguments that will be appended to the closure in order to construct the corresponding meta-call. Normal arguments are represented by the atom `*`. Meta-arguments are always called in the meta-non-terminal calling context, not in the meta-non-terminal definition context.

Template and modes

```
meta_non_terminal(+meta_non_terminal_template_term)

meta_non_terminal(+object_identifier::+meta_non_terminal_template_term)
meta_non_terminal(+category_identifier::+meta_non_terminal_template_term)

meta_non_terminal(+module_identifier:+meta_non_terminal_template_term)
```

Examples

```
:- meta_non_terminal(phrase(1, *)).
phrase(X, T) --> call(X, T).
```

See also

meta_predicate/1, *predicate_property/2*

directive

mode/2

Description

```
mode(Mode, NumberOfProofs)
```

Most predicates can be used with several instantiations modes. This directive enables the specification of each *instantiation mode* and the corresponding *number of proofs* (but not necessarily distinct solutions). The instantiation mode of each argument can include type information. You may also use this directive for documenting grammar rule non-terminals.

Template and modes

```
mode(+predicate_mode_term, +number_of_proofs)
mode(+non_terminal_mode_term, +number_of_proofs)
```

Examples

```
:- mode(atom_concat(-atom, -atom, +atom), one_or_more).
:- mode(atom_concat(+atom, +atom, -atom), one).

:- mode(var(@term), zero_or_one).

:- mode(solve(+callable, -list(atom)), zero_or_one).
```

See also

info/2, *predicate_property/2*

directive

multifile/1

Description

```
multifile(Name/Arity)
multifile((Name/Arity, ...))
multifile([Name/Arity, ...])

multifile(Entity::Name/Arity)
multifile((Entity::Name/Arity, ...))
multifile([Entity::Name/Arity, ...])

multifile(Module:Name/Arity)
multifile((Module:Name/Arity, ...))
multifile([Module:Name/Arity, ...])
```

(continues on next page)

(continued from previous page)

```
multifile(Name//Arity)
multifile((Name//Arity, ...))
multifile([Name//Arity, ...])

multifile(Entity::Name//Arity)
multifile((Entity::Name//Arity, ...))
multifile([Entity::Name//Arity, ...])

multifile(Module:Name//Arity)
multifile((Module:Name//Arity, ...))
multifile([Module:Name//Arity, ...])
```

Declares *multifile* predicates and multifile grammar rule non-terminals. In the case of object or category multifile predicates, the predicate (or non-terminal) must also have a *scope directive* in the object or category holding its *primary declaration* (i.e., the declaration without the `Entity::` prefix). Entities holding multifile predicate primary declarations must be compiled and loaded prior to any entities contributing with clauses for the multifile predicates (to prevent using multifile predicates to break entity encapsulation).

Protocols cannot declare or define multifile predicates as protocols cannot contain predicate definitions.

Warning

Some backend Prolog compilers declare the atom `multifile` as an operator for a lighter syntax. But this makes the code non-portable and is therefore a practice best avoided.

Template and modes

```
multifile(+qualified_predicate_indicator_term)
multifile(+qualified_non_terminal_indicator_term)
```

Examples

```
:- multifile(table/3).
:- multifile(user::hook/2).
```

See also

public/1, *protected/1*, *private/1*, *predicate_property/2*

directive

private/1

Description

```
private(Name/Arity)
private((Name/Arity, ...))
private([Name/Arity, ...])

private(Name//Arity)
private((Name//Arity, ...))
private([Name//Arity, ...])


private(op(Precedence,Associativity,Operator))
private((op(Precedence,Associativity,Operator), ...))
private([op(Precedence,Associativity,Operator), ...])
```

Declares *private* predicates, private grammar rule non-terminals, and private operators. A private predicate can only be called from the object containing the private directive. A private non-terminal can only be used in a call of the *phrase/2* and *phrase/3* methods from the object containing the private directive.

Template and modes

```
private(+predicate_indicator_term)
private(+non_terminal_indicator_term)
private(+operator_declaration)
```

Examples

```
:- private(counter/1).

:- private((init/1, free/1)).

:- private([data/3, key/1, keys/1]).
```

See also

protected/1, *public/1*, *predicate_property/2*

directive

protected/1

Description

```
protected(Name/Arity)
protected((Name/Arity, ...))
protected([Name/Arity, ...])

protected(Name//Arity)
protected((Name//Arity, ...))
protected([Name//Arity, ...])

protected(op(Precedence,Associativity,Operator))
protected((op(Precedence,Associativity,Operator), ...))
protected([op(Precedence,Associativity,Operator), ...])
```

Declares *protected* predicates, protected grammar rule non-terminals, and protected operators. A protected predicate can only be called from the object containing the directive or from an object that inherits the directive. A protected non-terminal can only be used as an argument in a *phrase/2* and *phrase/3* calls from the object containing the directive or from an object that inherits the directive.

Note

Protected operators are not inherited but declaring them provides a reusable specification for using them in descendant objects (or categories).

Template and modes

```
protected(+predicate_indicator_term)
protected(+non_terminal_indicator_term)
protected(+operator_declaration)
```

Examples

```
:- protected(init/1).

:- protected((print/2, convert/4)).

:- protected([load/1, save/3]).
```

See also

private/1, *public/1*, *predicate_property/2*

directive

public/1

Description

```
public(Name/Arity)
public((Name/Arity, ...))
public([Name/Arity, ...])

public(Name//Arity)
public((Name//Arity, ...))
public([Name//Arity, ...])

public(op(Precedence,Associativity,Operator))
public((op(Precedence,Associativity,Operator), ...))
public([op(Precedence,Associativity,Operator), ...])
```

Declares *public* predicates, public grammar rule non-terminals, and public operators. A public predicate can be called from any object. A public non-terminal can be used as an argument in *phrase/2* and *phrase/3* calls from any object.

Note

Declaring a public operator does not make it global when the entity holding the scope directive is compiled and loaded. But declaring public operators provides a reusable specification for using them in the entity clients (e.g., in *uses/2* directives).

Template and modes

```
public(+predicate_indicator_term)
public(+non_terminal_indicator_term)
public(+operator_declaration)
```

Examples

```
:- public(ancestor/1).

:- public((instance/1, instances/1)).

:- public([leaf/1, leaves/1]).
```

See also

private/1, *protected/1*, *predicate_property/2*

directive

synchronized/1

Description

```
synchronized(Name/Arity)
synchronized((Name/Arity, ...))
synchronized([Name/Arity, ...])

synchronized(Name//Arity)
synchronized((Name//Arity, ...))
synchronized([Name//Arity, ...])
```

Declares *synchronized* predicates and synchronized grammar rule non-terminals. The most common use is for predicates that have side effects (e.g., asserting or retracting clauses for a dynamic predicate) in multi-threaded applications. A synchronized predicate (or synchronized non-terminal) is protected by a mutex in order to allow for thread synchronization when proving a call to the predicate (or non-terminal).

All predicates (and non-terminals) declared in the same synchronized directive share the same mutex. In order to use a separate mutex for each predicate (non-terminal) so that they are independently synchronized, a per-predicate synchronized directive must be used.

Warning

Declaring a predicate synchronized implicitly makes it **deterministic**. When using a single-threaded *backend Prolog compiler*, calls to synchronized predicates behave as wrapped by the standard *once/1* meta-predicate.

Note that synchronized predicates cannot be declared *dynamic* (when necessary, declare the predicates updating the dynamic predicates as synchronized).

Template and modes

```
synchronized(+predicate_indicator_term)
synchronized(+non_terminal_indicator_term)
```

Examples

```
:- synchronized(db_update/1).

:- synchronized((write_stream/2, read_stream/2)).

:- synchronized([add_to_queue/2, remove_from_queue/2]).
```

See also

predicate_property/2

directive

uses/2

Description

```
uses(Object, [Name/Arity, ...])
uses(Object, [Name/Arity as Alias/Arity, ...])

uses(Object, [Predicate as Alias, ...])

uses(Object, [Name//Arity, ...])
uses(Object, [Name//Arity as Alias//Arity, ...])

uses(Object, [op(Precedence, Associativity, Operator), ...])
```

Declares that all calls made from predicates (or non-terminals) defined in the category or object containing the directive to the specified predicates (or non-terminals) are to be interpreted as messages to the specified object. Thus, this directive may be used to simplify writing of predicate definitions by allowing the programmer to omit the `Object::` prefix when using the predicates listed in the directive (as long as the calls do not occur as arguments for non-standard Prolog meta-predicates not declared in the adapter files). It is also possible to include operator declarations in the second argument.

This directive is also taken into account when compiling calls to the *database* and *reflection* built-in methods by looking into these methods predicate arguments if bound at compile-time.

It is possible to specify a predicate alias using the notation `Name/Arity as Alias/Arity` or, in alternative, the notation `Name/Arity::Alias/Arity`. Aliases may be used to avoid conflicts between predicates specified in `use_module/2` and `uses/2` directives, for giving more meaningful names considering the calling context of the predicates, and to change the order of the predicate arguments when calling the predicate. For predicates, it is also possible to define alias shorthands using the notation `Predicate as Alias` or, in alternative, the notation `Predicate::Alias`, where `Predicate` and `Alias` are callable terms where some or all arguments may be instantiated.

To enable the use of *static binding*, and thus optimal message-sending performance, the objects should be loaded before compiling the entities that call their predicates.

The object identifier argument can also be a *parameter variable* when using the directive in a parametric object or a parametric category defined in a source file (the common case). In this case, *dynamic binding* will be used for all listed predicates (and non-terminals). The parameter variable must be instantiated at runtime when the messages are sent.

Template and modes

```
uses(+object_identifier, +predicate_indicator_list)
uses(+object_identifier, +predicate_indicator_alias_list)

uses(+object_identifier, +predicate_template_alias_list)

uses(+object_identifier, +non_terminal_indicator_list)
uses(+object_identifier, +non_terminal_indicator_alias_list)

uses(+object_identifier, +operator_list)
```

Examples

```
:- uses(list,  [append/3, member/2]).
:- uses(store, [data/2]).
:- uses(user,  [table/4]).

foo :-
    ...,
    % the same as findall(X, list::member(X, L), A)
    findall(X, member(X, L), A),
    % the same as list::append(A, B, C)
    append(A, B, C),
    % the same as store::assertz(data(X, C))
    assertz(data(X, C)),
    % call the table/4 predicate in "user"
    table(X, Y, Z, T),
    ...
```

Another example, using the extended notation that allows us to define predicate aliases:

```
:- uses(btrees, [new/1 as new_btree/1]).
:- uses(queues, [new/1 as new_queue/1]).

btree_to_queue :-
    ...,
    % the same as btrees::new(Tree)
    new_btree(Tree),
    % the same as queues::new(Queue)
    new_queue(Queue),
    ...
```

An example of defining a predicate alias that is also a shorthand:

```
:- uses(logtalk, [
    print_message(debug, my_app, Message) as dbg(Message)
]).
```

Predicate aliases can also be used to change argument order:

```
:- uses(meta, [
    fold_left(Closure,Accumulator,List,Result) as foldl(Closure,List,Accumulator,Result)
]).
```

An example of using a *parameter variable* in place of the object identifier to allow using the same test set for checking multiple implementations of the same protocol:

```
:- object(tests(_HeapObject_),
    extends(lgtunit)).

    :- uses(_HeapObject_, [
        as_heap/2, as_list/2, valid/1, new/1,
        insert/4, insert_all/3, delete/4, merge/3,
        empty/1, size/2, top/3, top_next/5
    ]).
```

See also

uses/1, *use_module/1*, *use_module/2*, *alias/2*

directive

use_module/2

Description

```
use_module(Module, [Name/Arity, ...])
use_module(Module, [Name/Arity as Alias/Arity, ...])

use_module(Module, [Predicate as Alias, ...])

use_module(Module, [Name//Arity, ...])
use_module(Module, [Name//Arity as Alias//Arity, ...])

use_module(Module, [op(Precedence,Associativity,Operator), ...])
```

This directive declares that all calls (made from predicates defined in the category or object containing the directive) to the specified predicates (or non-terminals) are to be interpreted as calls to explicitly-qualified module predicates (or non-terminals). Thus, this directive may be used to simplify the writing of predicate definitions by allowing the programmer to omit the `Module:` prefix when using the predicates listed in the directive (as long as the predicate calls do not occur as arguments for non-standard Prolog meta-predicates not declared on the adapter files). It is also possible to include operator declarations in the second argument.

This directive is also taken into account when compiling calls to the *database* and *reflection* built-in methods by looking into these methods predicate arguments if bound at compile-time.

It is possible to specify a predicate alias using the notation `Name/Arity as Alias/Arity` or, in alternative, the notation `Name/Arity:Alias/Arity`. Aliases may be used either for avoiding conflicts between predicates specified in `use_module/2` and *uses/2* directives or for giving more meaningful names considering the calling context of the predicates. For predicates, it is also possible to define alias shorthands using the notation `Predicate as Alias` or, in alternative, the notation `Predicate::Alias`, where `Predicate` and `Alias` are callable terms where some or all arguments may be instantiated.

Note that this directive differs from the directive with the same name found on some Prolog implementations by requiring the first argument to be a module name (an atom) instead of a file specification. In Logtalk, there's no mixing between *loading* a resource and (declaring the) *using* (of) a resource. As a consequence, this directive doesn't automatically load the module. Loading the module file is dependent on the used *back-end Prolog compiler* and must be done separately (usually, using a source file directive such as `use_module/1` or `use_module/2` in the entity file or preferably in the application *loader file* file). Also, note that the name of the module may differ from the name of the module file.

Warning

The modules **must** be loaded prior to the compilation of entities that call the module predicates. This is required in general to allow the compiler to check if the called module predicate is a meta-predicate and retrieve its meta-predicate template to ensure proper call compilation.

The module identifier argument can also be a *parameter variable* when using the directive in a parametric object or a parametric category defined in a source file (the common case). In this case, *dynamic binding* will be used for all listed predicates (and non-terminals). The parameter variable must be instantiated at runtime when the calls are made.

Template and modes

```
use_module(+module_identifier, +predicate_indicator_list)
use_module(+module_identifier, +module_predicate_indicator_alias_list)

use_module(+module_identifier, +predicate_template_alias_list)

use_module(+module_identifier, +non_terminal_indicator_list)
use_module(+module_identifier, +module_non_terminal_indicator_alias_list)

use_module(+module_identifier, +operator_list)
```

Examples

```
:- use_module(lists, [append/3, member/2]).
:- use_module(store, [data/2]).
:- use_module(user,  [foo/1 as bar/1]).

foo :-
    ...,
    % same as findall(X, lists:member(X, L), A)
    findall(X, member(X, L), A),
    % same as lists:append(A, B, C)
    append(A, B, C),
    % same as assertz(store:data(X, C))
    assertz(data(X, C)),
    % same as retractall(user:foo(_))
    retractall(bar(_)),
    ...
```

Another example, using the extended notation that allows us to define predicate aliases:

```
:- use_module(ugraphs, [transpose_ugraph/2 as transpose/2]).

convert_graph :-
    ...,
    % the same as ugraphs:transpose_ugraph(Graph0, Graph)
    transpose(Graph0, Graph),
    ...
```

An example of defining a predicate alias that is also a shorthand:

```
:- use_module(pairs, [
    map_list_to_pairs(length, Lists, Pairs) as length_pairs(Lists, Pairs)
]).
```

An example of using a *parameter variable* in place of the module identifier to delay to runtime the actual module to use:

```
:- object(bar(_OptionsModule_)).

    :- use_module(_OptionsModule_, [
        set/2, get/2, reset/0
    ]).
```

See also

use_module/1, *uses/2*, *uses/1*, *alias/2*

2.4 Built-in predicates

2.4.1 Enumerating objects, categories and protocols

built-in predicate

`current_category/1`

Description

```
current_category(Category)
```

Enumerates, by backtracking, all currently defined categories. All categories are found, either static, dynamic, or built-in.

Modes and number of proofs

```
current_category(?category_identifier) - zero_or_more
```

Errors

`Category` is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`

Examples

```
% enumerate the defined categories:
| ?- current_category(Category).

Category = core_messages ;
...
```

See also

abolish_category/1, *category_property/2*, *create_category/4*, *complements_object/2*, *extends_category/2-3*, *imports_category/2-3*

built-in predicate

current_object/1

Description

```
current_object(Object)
```

Enumerates, by backtracking, all currently defined objects. All objects are found, either static, dynamic or built-in.

Modes and number of proofs

```
current_object(?object_identifier) - zero_or_more
```

Errors

`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`

Examples

```
% enumerate the defined objects:
| ?- current_object(Object).

Object = user ;
Object = logtalk ;
...
```

See also

abolish_object/1, *create_object/4*, *object_property/2*, *extends_object/2-3*, *instantiates_class/2-3*, *specializes_class/2-3*, *complements_object/2*

built-in predicate

current_protocol/1

Description

```
current_protocol(Protocol)
```

Enumerates, by backtracking, all currently defined protocols. All protocols are found, either static, dynamic, or built-in.

Modes and number of proofs

```
current_protocol(?protocol_identifier) - zero_or_more
```

Errors

`Protocol` is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`

Examples

```
% enumerate the defined protocols:
| ?- current_protocol(Protocol).

Protocol = expanding ;
Protocol = monitoring ;
Protocol = forwarding ;
...
```

See also

abolish_protocol/1, *create_protocol/3*, *protocol_property/2*, *conforms_to_protocol/2-3*, *extends_protocol/2-3*, *implements_protocol/2-3*

2.4.2 Enumerating objects, categories and protocols properties

built-in predicate

category_property/2

Description

```
category_property(Category, Property)
```

Enumerates, by backtracking, the properties associated with the defined categories. The valid properties are listed in the language grammar section on *entity properties* and described in the User Manual section on *category properties*.

Modes and number of proofs

```
category_property(?category_identifier, ?category_property) - zero_or_more
```

Errors

`Category` is neither a variable nor a valid category identifier:
`type_error(category_identifier, Category)`

`Property` is neither a variable nor a callable term:
`type_error(callable, Property)`
`Property` is a callable term but not a valid category property:
`domain_error(category_property, Property)`

Examples

```
% enumerate the properties of the core_messages built-in category:
| ?- category_property(core_messages, Property).

Property = source_data ;
Property = static ;
Property = built_in ;
...
```

See also

abolish_category/1, *create_category/4*, *current_category/1*, *complements_object/2*, *extends_category/2-3*, *imports_category/2-3*

built-in predicate

object_property/2

Description

```
object_property(Object, Property)
```

Enumerates, by backtracking, the properties associated with the defined objects. The valid properties are listed in the language grammar section on *entity properties* and described in the User Manual section on *object properties*.

Modes and number of proofs

```
object_property(?object_identifier, ?object_property) - zero_or_more
```

Errors

`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
`Property` is neither a variable nor a callable term:
`type_error(callable, Property)`
`Property` is a callable term but not a valid object property:
`domain_error(object_property, Property)`

Examples

```
% enumerate the properties of the logtalk built-in object:
| ?- object_property(logtalk, Property).

Property = context_switching_calls ;
Property = source_data ;
Property = threaded ;
Property = static ;
Property = built_in ;
...
```

See also

abolish_object/1, *create_object/4*, *current_object/1*, *extends_object/2-3*, *instantiates_class/2-3*, *specializes_class/2-3*, *complements_object/2*

built-in predicate

protocol_property/2

Description

```
protocol_property(Protocol, Property)
```

Enumerates, by backtracking, the properties associated with the currently defined protocols. The valid properties are listed in the language grammar section on *entity properties* and described in the User Manual section on *protocol properties*.

Modes and number of proofs

```
protocol_property(?protocol_identifier, ?protocol_property) - zero_or_more
```

Errors

`Protocol` is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
`Property` is neither a variable nor a callable term:
`type_error(callable, Property)`
`Property` is a callable term but not a valid protocol property:
`domain_error(protocol_property, Property)`

Examples

```
% enumerate the properties of the monitoring built-in protocol:
| ?- protocol_property(monitoring, Property).

Property = source_data ;
Property = static ;
Property = built_in ;
...
```

See also

abolish_protocol/1, *create_protocol/3*, *current_protocol/1*, *conforms_to_protocol/2-3*, *extends_protocol/2-3*, *implements_protocol/2-3*

2.4.3 Creating new objects, categories and protocols

built-in predicate

create_category/4

Description

```
create_category(Identifier, Relations, Directives, Clauses)
```

Creates a new, dynamic category. This predicate is often used as a primitive to implement high-level category creation methods.

Note that, when opting for runtime generated category identifiers, it's possible to run out of identifiers when using a *backend Prolog compiler* with bounded integer support. The portable solution, when creating a large number of dynamic categories in long-running applications, is to recycle, whenever possible, the identifiers.

When creating a new dynamic parametric category, access to the object parameters must use the *parameter/2* built-in execution context method.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

Modes and number of proofs

```
create_category(?category_identifier, @list(category_relation), @list(category_directive),␣
↪@list(clause)) - one
```

Errors

`Relations`, `Directives`, or `Clauses` is a variable:
    `instantiation_error`
`Identifier` is neither a variable nor a valid category identifier:
    `type_error(category_identifier, Identifier)`
`Identifier` is already in use:
    `permission_error(modify, category, Identifier)`
    `permission_error(modify, object, Identifier)`
    `permission_error(modify, protocol, Identifier)`
`Relations` is neither a variable nor a proper list:
    `type_error(list, Relations)`
Repeated entity relation clause:
    `permission_error(repeat, entity_relation, implements/1)`
    `permission_error(repeat, entity_relation, extends/1)`
    `permission_error(repeat, entity_relation, complements/1)`
`Directives` is neither a variable nor a proper list:
    `type_error(list, Directives)`
`Clauses` is neither a variable nor a proper list:
    `type_error(list, Clauses)`

Examples

```
| ?- create_category(
        tolerances,
        [implements(comparing)],
        [],
        [epsilon(1e-15), (equal(X, Y) :- epsilon(E), abs(X-Y) =< E)]
    ).
```

See also

abolish_category/1, *category_property/2*, *current_category/1*, *complements_object/2*, *extends_category/2-3*, *imports_category/2-3*

built-in predicate

create_object/4

Description

```
create_object(Identifier, Relations, Directives, Clauses)
```

Creates a new, dynamic object. The word *object* is used here as a generic term. This predicate can be used to create new prototypes, instances, and classes. This predicate is often used as a primitive to implement high-level object creation methods.

Note that, when opting for runtime generated object identifiers, it's possible to run out of identifiers when using a *backend Prolog compiler* with bounded integer support. The portable solution, when creating a large number of dynamic objects in long-running applications, is to recycle, whenever possible, the identifiers.

When creating a new dynamic parametric object, access to the object parameters must use the *parameter/2* built-in execution context method.

Declared predicates (using scope clauses in the `Directives` argument) are implicitly declared also as dynamic.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

Modes and number of proofs

```
create_object(?object_identifier, @list(object_relation), @list(object_directive),␣
↪@list(clause)) - one
```

Errors

`Relations`, `Directives`, or `Clauses` is a variable:
`instantiation_error`
`Identifier` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Identifier)`
`Identifier` is already in use:
`permission_error(modify, category, Identifier)`
`permission_error(modify, object, Identifier)`
`permission_error(modify, protocol, Identifier)`
`Relations` is neither a variable nor a proper list:
`type_error(list, Relations)`
Repeated entity relation clause:
`permission_error(repeat, entity_relation, implements/1)`
`permission_error(repeat, entity_relation, imports/1)`
`permission_error(repeat, entity_relation, extends/1)`
`permission_error(repeat, entity_relation, instantiates/1)`
`permission_error(repeat, entity_relation, specializes/1)`
`Directives` is neither a variable nor a proper list:
`type_error(list, Directives)`
`Clauses` is neither a variable nor a proper list:
`type_error(list, Clauses)`

Examples

```
% create a stand-alone object (a prototype):
| ?- create_object(
        translator,
        [],
        [public(int/2)],
        [int(0, zero)]
    ).

% create a prototype derived from a parent prototype:
| ?- create_object(
        mickey,
        [extends(mouse)],
        [public(alias/1)],
        [alias(mortimer)]
    ).

% create a class instance:
| ?- create_object(
        p1,
        [instantiates(person)],
        [],
        [name('Paulo Moura'), age(42)]
    ).
```

(continues on next page)

(continued from previous page)

```
% create a subclass:
| ?- create_object(
        hovercraft,
        [specializes(vehicle)],
        [public([propeller/2, fan/2])],
        []
    ).

% create an object with an initialization goal:
| ?- create_object(
        runner,
        [instantiates(runners)],
        [initialization(::start)],
        [length(22), time(60)]
    ).

% create an object supporting dynamic predicate declarations:
| ?- create_object(
        database,
        [],
        [set_logtalk_flag(dynamic_declarations, allow)],
        []
    ).
```

See also

abolish_object/1, *current_object/1*, *object_property/2*, *extends_object/2-3*, *instantiates_class/2-3*, *specializes_class/2-3*, *complements_object/2*

built-in predicate

create_protocol/3

Description

```
create_protocol(Identifier, Relations, Directives)
```

Creates a new, dynamic, protocol. This predicate is often used as a primitive to implement high-level protocol creation methods.

Note that, when opting for runtime generated protocol identifiers, it's possible to run out of identifiers when using a *backend Prolog compiler* with bounded integer support. The portable solution, when creating a large number of dynamic protocols in long-running applications, is to recycle, whenever possible, the identifiers.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

Modes and number of proofs

```
create_protocol(?protocol_identifier, @list(protocol_relation), @list(protocol_directive)) -␣
↪one
```

Errors

Either `Relations` or `Directives` is a variable:
`instantiation_error`
`Identifier` is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Identifier)`
`Identifier` is already in use:
`permission_error(modify, category, Identifier)`
`permission_error(modify, object, Identifier)`
`permission_error(modify, protocol, Identifier)`
`Relations` is neither a variable nor a proper list:
`type_error(list, Relations)`
Repeated entity relation clause:
`permission_error(repeat, entity_relation, extends/1)`
`Directives` is neither a variable nor a proper list:
`type_error(list, Directives)`

Examples

```
| ?- create_protocol(
       logging,
       [extends(monitoring)],
       [public([log_file/1, log_on/0, log_off/0])]
    ).
```

See also

abolish_protocol/1, *current_protocol/1*, *protocol_property/2*, *conforms_to_protocol/2-3*, *extends_protocol/2-3*, *implements_protocol/2-3*

2.4.4 Abolishing objects, categories and protocols

built-in predicate

abolish_category/1

Description

```
abolish_category(Category)
```

Abolishes a dynamic category. The category identifier can then be reused when creating a new category.

Modes and number of proofs

```
abolish_category(+category_identifier) - one
```

Errors

`Category` is a variable:
    `instantiation_error`
`Category` is neither a variable nor a valid category identifier:
    `type_error(category_identifier, Category)`
`Category` is an identifier of a static category:
    `permission_error(modify, static_category, Category)`
`Category` does not exist:
    `existence_error(category, Category)`

Examples

```
| ?- abolish_category(monitoring).
```

See also

category_property/2, *create_category/4*, *current_category/1 complements_object/2*, *extends_category/2-3*, *imports_category/2-3*

built-in predicate

abolish_object/1

Description

```
abolish_object(Object)
```

Abolishes a dynamic object. The object identifier can then be reused when creating a new object.

Modes and number of proofs

```
abolish_object(+object_identifier) - one
```

Errors

`Object` is a variable:
    `instantiation_error`
`Object` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Object)`
`Object` is an identifier of a static object:
    `permission_error(modify, static_object, Object)`
`Object` does not exist:
    `existence_error(object, Object)`

Examples

```
| ?- abolish_object(list).
```

See also

create_object/4, *current_object/1*, *object_property/2*, *extends_object/2-3*, *instantiates_class/2-3*, *specializes_class/2-3*, *complements_object/2*

built-in predicate

abolish_protocol/1

Description

```
abolish_protocol(Protocol)
```

Abolishes a dynamic protocol. The protocol identifier can then be reused when creating a new protocol.

Modes and number of proofs

```
abolish_protocol(@protocol_identifier) - one
```

Errors

`Protocol` is a variable:
    `instantiation_error`
`Protocol` is neither a variable nor a valid protocol identifier:
    `type_error(protocol_identifier, Protocol)`
`Protocol` is an identifier of a static protocol:
    `permission_error(modify, static_protocol, Protocol)`
`Protocol` does not exist:
    `existence_error(protocol, Protocol)`

Examples

```
| ?- abolish_protocol(listp).
```

See also

create_protocol/3, *current_protocol/1*, *protocol_property/2*, *conforms_to_protocol/2-3*, *extends_protocol/2-3*, *implements_protocol/2-3*

2.4.5 Objects, categories, and protocols relations

built-in predicate

extends_object/2-3

Description

```
extends_object(Prototype, Parent)
extends_object(Prototype, Parent, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one extends the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
extends_object(?object_identifier, ?object_identifier) - zero_or_more
extends_object(?object_identifier, ?object_identifier, ?scope) - zero_or_more
```

Errors

`Prototype` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Prototype)`
`Parent` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Parent)`
`Scope` is neither a variable nor an atom:
    `type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
    `domain_error(scope, Scope)`

Examples

```
% enumerate objects derived from the state_space prototype:
| ?- extends_object(Object, state_space).

% enumerate objects publicly derived from the list prototype:
| ?- extends_object(Object, list, public).
```

See also

current_object/1, *instantiates_class/2-3*, *specializes_class/2-3*

built-in predicate

extends_protocol/2-3

Description

```
extends_protocol(Protocol, ParentProtocol)
extends_protocol(Protocol, ParentProtocol, Scope)
```

Enumerates, by backtracking, all pairs of protocols such that the first one extends the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
extends_protocol(?protocol_identifier, ?protocol_identifier) - zero_or_more
extends_protocol(?protocol_identifier, ?protocol_identifier, ?scope) - zero_or_more
```

Errors

`Protocol` is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, Protocol)`
`ParentProtocol` is neither a variable nor a valid protocol identifier:
`type_error(protocol_identifier, ParentProtocol)`
`Scope` is neither a variable nor an atom:
`type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
`domain_error(scope, Scope)`

Examples

```
% enumerate the protocols extended by the listp protocol:
| ?- extends_protocol(listp, Protocol).

% enumerate protocols that privately extend the termp protocol:
| ?- extends_protocol(Protocol, termp, private).
```

See also

current_protocol/1, *implements_protocol/2-3*, *conforms_to_protocol/2-3*

built-in predicate

extends_category/2-3

Description

```
extends_category(Category, ParentCategory)
extends_category(Category, ParentCategory, Scope)
```

Enumerates, by backtracking, all pairs of categories such that the first one extends the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
extends_category(?category_identifier, ?category_identifier) - zero_or_more
extends_category(?category_identifier, ?category_identifier, ?scope) - zero_or_more
```

Errors

`Category` is neither a variable nor a valid protocol identifier:
`type_error(category_identifier, Category)`
`ParentCategory` is neither a variable nor a valid protocol identifier:
`type_error(category_identifier, ParentCategory)`
`Scope` is neither a variable nor an atom:
`type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
`domain_error(scope, Scope)`

Examples

```
% enumerate the categories extended by the derailleur category:
| ?- extends_category(derailleur, Category).

% enumerate categories that privately extend the basics category:
| ?- extends_category(Category, basics, private).
```

See also

current_category/1, *complements_object/2*, *imports_category/2-3*

built-in predicate

implements_protocol/2-3

Description

```
implements_protocol(Object, Protocol)
implements_protocol(Category, Protocol)

implements_protocol(Object, Protocol, Scope)
implements_protocol(Category, Protocol, Scope)
```

Enumerates, by backtracking, all pairs of entities such that an object or a category implements a protocol. The relation scope is represented by the atoms `public`, `protected`, and `private`. This predicate only returns direct implementation relations. For a transitive closure, see the *conforms_to_protocol/2-3* predicate.

Modes and number of proofs

```
implements_protocol(?object_identifier, ?protocol_identifier) - zero_or_more
implements_protocol(?category_identifier, ?protocol_identifier) - zero_or_more

implements_protocol(?object_identifier, ?protocol_identifier, ?scope) - zero_or_more
implements_protocol(?category_identifier, ?protocol_identifier, ?scope) - zero_or_more
```

Errors

`Object` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Object)`
`Category` is neither a variable nor a valid category identifier:
    `type_error(category_identifier, Category)`
`Protocol` is neither a variable nor a valid protocol identifier:
    `type_error(protocol_identifier, Protocol)`
`Scope` is neither a variable nor an atom:
    `type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
    `domain_error(scope, Scope)`

Examples

```
% check that the list object implements the listp protocol:
| ?- implements_protocol(list, listp).

% check that the list object publicly implements the listp protocol:
| ?- implements_protocol(list, listp, public).

% enumerate only objects that implement the listp protocol:
| ?- current_object(Object), implements_protocol(Object, listp).

% enumerate only categories that implement the serialization protocol:
| ?- current_category(Category), implements_protocol(Category, serialization).
```

See also

current_object/1, *current_protocol/1*, *current_category/1*, *conforms_to_protocol/2-3*

built-in predicate

conforms_to_protocol/2-3

Description

```
conforms_to_protocol(Object, Protocol)
conforms_to_protocol(Category, Protocol)

conforms_to_protocol(Object, Protocol, Scope)
conforms_to_protocol(Category, Protocol, Scope)
```

Enumerates, by backtracking, all pairs of entities such that an object or a category conforms to a protocol. The relation scope is represented by the atoms `public`, `protected`, and `private`. This predicate implements a transitive closure for the protocol implementation relation.

Modes and number of proofs

```
conforms_to_protocol(?object_identifier, ?protocol_identifier) - zero_or_more
conforms_to_protocol(?category_identifier, ?protocol_identifier) - zero_or_more

conforms_to_protocol(?object_identifier, ?protocol_identifier, ?scope) - zero_or_more
conforms_to_protocol(?category_identifier, ?protocol_identifier, ?scope) - zero_or_more
```

Errors

`Object` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Object)`
`Category` is neither a variable nor a valid category identifier:
    `type_error(category_identifier, Category)`
`Protocol` is neither a variable nor a valid protocol identifier:
    `type_error(protocol_identifier, Protocol)`
`Scope` is neither a variable nor an atom:
    `type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
    `domain_error(scope, Scope)`

Examples

```
% enumerate objects and categories that conform to the listp protocol:
| ?- conforms_to_protocol(Object, listp).

% enumerate objects and categories that privately conform to the listp protocol:
| ?- conforms_to_protocol(Object, listp, private).

% enumerate only objects that conform to the listp protocol:
| ?- current_object(Object), conforms_to_protocol(Object, listp).
```

(continues on next page)

(continued from previous page)

```
% enumerate only categories that conform to the serialization protocol:
| ?- current_category(Category), conforms_to_protocol(Category, serialization).
```

See also

current_object/1, *current_protocol/1*, *current_category/1*, *implements_protocol/2-3*

built-in predicate

complements_object/2

Description

```
complements_object(Category, Object)
```

Enumerates, by backtracking, all category–object pairs such that the category explicitly complements the object.

Modes and number of proofs

```
complements_object(?category_identifier, ?object_identifier) - zero_or_more
```

Errors

`Category` is neither a variable nor a valid category identifier:
`type_error(category_identifier, Prototype)`
`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Parent)`

Examples

```
% check that the logging category complements the employee object:
| ?- complements_object(logging, employee).
```

See also

current_category/1, *imports_category/2-3*

built-in predicate

imports_category/2-3

Description

```
imports_category(Object, Category)

imports_category(Object, Category, Scope)
```

Enumerates, by backtracking, importation relations between objects and categories. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
imports_category(?object_identifier, ?category_identifier) - zero_or_more
imports_category(?object_identifier, ?category_identifier, ?scope) - zero_or_more
```

Errors

`Object` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Object)`
`Category` is neither a variable nor a valid category identifier:
    `type_error(category_identifier, Category)`
`Scope` is neither a variable nor an atom:
    `type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
    `domain_error(scope, Scope)`

Examples

```
% check that the xref_diagram object imports the diagram category:
| ?- imports_category(xref_diagram, diagram).

% enumerate the objects that privately import the diagram category:
| ?- imports_category(Object, diagram, private).
```

See also

current_category/1, *complements_object/2*

built-in predicate

instantiates_class/2-3

Description

```
instantiates_class(Instance, Class)
instantiates_class(Instance, Class, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one instantiates the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
instantiates_class(?object_identifier, ?object_identifier) - zero_or_more
instantiates_class(?object_identifier, ?object_identifier, ?scope) - zero_or_more
```

Errors

`Instance` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Instance)`
`Class` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Class)`
`Scope` is neither a variable nor an atom:
    `type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
    `domain_error(scope, Scope)`

Examples

```
% check that the water_jug is an instante of state_space:
| ?- instantiates_class(water_jug, state_space).

% enumerate the state_space instances where the
% instantiation relation is public:
| ?- instantiates_class(Space, state_space, public).
```

See also

current_object/1, *extends_object/2-3*, *specializes_class/2-3*

built-in predicate

specializes_class/2-3

Description

```
specializes_class(Class, Superclass)
specializes_class(Class, Superclass, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one specializes the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Modes and number of proofs

```
specializes_class(?object_identifier, ?object_identifier) - zero_or_more
specializes_class(?object_identifier, ?object_identifier, ?scope) - zero_or_more
```

Errors

`Class` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Class)`
`Superclass` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Superclass)`
`Scope` is neither a variable nor an atom:
`type_error(atom, Scope)`
`Scope` is an atom but an invalid entity scope:
`domain_error(scope, Scope)`

Examples

```
% enumerate the state_space subclasses:
| ?- specializes_class(Subclass, state_space).

% enumerate the state_space subclasses where the
% specialization relation is public:
| ?- specializes_class(Subclass, state_space, public).
```

See also

current_object/1, *extends_object/2-3*, *instantiates_class/2-3*

2.4.6 Event handling

built-in predicate

abolish_events/5

Description

```
abolish_events(Event, Object, Message, Sender, Monitor)
```

Abolishes all matching events. The two types of events are represented by the atoms `before` and `after`. When the predicate is called with the first argument unbound, both types of events are abolished.

Modes and number of proofs

```
abolish_events(@term, @term, @term, @term, @term) - one
```

Errors

`Event` is neither a variable nor a valid event identifier:
`type_error(event, Event)`
`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
`Message` is neither a variable nor a callable term:
`type_error(callable, Message)`
`Sender` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
`Monitor` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
% abolish all events for messages sent to the "list"
% object being monitored by the "debugger" object:
| ?- abolish_events(_, list, _, _, debugger).
```

See also

current_event/5, *define_events/5*, *before/3*, *after/3*

built-in predicate

current_event/5

Description

```
current_event(Event, Object, Message, Sender, Monitor)
```

Enumerates, by backtracking, all defined events. The two types of events are represented by the atoms `before` and `after`.

Modes and number of proofs

```
current_event(?event, ?term, ?term, ?term, ?object_identifier) - zero_or_more
```

Errors

`Event` is neither a variable nor a valid event identifier:
`type_error(event, Event)`
`Object` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Object)`
`Message` is neither a variable nor a callable term:
`type_error(callable, Message)`
`Sender` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Sender)`
`Monitor` is neither a variable nor a valid object identifier:
`type_error(object_identifier, Monitor)`

Examples

```
% enumerate all events monitored by the "debugger" object:
| ?- current_event(Event, Object, Message, Sender, debugger).
```

See also

abolish_events/5, *define_events/5*, *before/3*, *after/3*

built-in predicate

define_events/5

Description

```
define_events(Event, Object, Message, Sender, Monitor)
```

Defines a new set of events. The two types of events are represented by the atoms `before` and `after`. When the predicate is called with the first argument unbound, both types of events are defined. The object `Monitor` must define the event handler methods required by the `Event` argument.

Modes and number of proofs

```
define_events(@term, @term, @term, @term, +object_identifier) - one
```

Errors

`Event` is neither a variable nor a valid event identifier:
    `type_error(event, Event)`
`Object` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Object)`
`Message` is neither a variable nor a callable term:
    `type_error(callable, Message)`
`Sender` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Sender)`
`Monitor` is a variable:
    `instantiation_error`
`Monitor` is neither a variable nor a valid object identifier:
    `type_error(object_identifier, Monitor)`
`Monitor` does not define the required `before/3` method:
    `existence_error(procedure, before/3)`
`Monitor` does not define the required `after/3` method:
    `existence_error(procedure, after/3)`

Examples

```
% define "debugger" as a monitor for member/2 messages
% sent to the "list" object:
| ?- define_events(_, list, member(_, _), _ , debugger).
```

See also

abolish_events/5, *current_event/5*, *before/3*, *after/3*

2.4.7 Multi-threading

built-in predicate

threaded/1

Description

```
threaded(Conjunction)
threaded(Disjunction)
```

Proves each goal in a conjunction or a disjunction of goals in its own thread. This meta-predicate is deterministic and opaque to cuts. The predicate argument is **not** flattened.

When the argument is a conjunction of goals, a call to this predicate blocks until either all goals succeed, one of the goals fail, or one of the goals generate an exception; the failure of one of the goals or an exception on the execution of one of the goals results in the termination of the remaining threads. The predicate call is true *iff* all goals are true. The predicate call fails if all goals fail. When one of the goals throws an exception, the predicate call re-throws that exception.

When the argument is a disjunction of goals, a call to this predicate blocks until either one of the goals succeeds or all the goals fail or throw exceptions; the success of one of the goals results in the termination of the remaining threads. The predicate call is true *iff* one of the goals is true. The predicate call fails if all goals fails. When no goal succeeds and one of the goals throws an exception, the predicate call re-throws that exception.

When the predicate argument is neither a conjunction nor a disjunction of goals, no threads are used. In this case, the predicate call is equivalent to a `once/1` predicate call.

A dedicated message queue is used per call of this predicate to collect the individual goal results.

> **Note**
>
> This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Meta-predicate template

```
threaded(0)
```

Modes and number of proofs

```
threaded(+callable) - zero_or_one
```

Errors

`Goals` is a variable:
```
instantiation_error
```
A goal in `Goals` is a variable:
```
instantiation_error
```
`Goals` is neither a variable nor a callable term:
```
type_error(callable, Goals)
```
A goal `Goal` in `Goals` is neither a variable nor a callable term:
```
type_error(callable, Goal)
```

Examples

Prove a conjunction of goals, each one in its own thread:
```
threaded((Goal, Goals))
```
Prove a disjunction of goals, each one in its own thread:
```
threaded((Goal; Goals))
```

See also

threaded_call/1-2, *threaded_once/1-2*, *threaded_ignore/1*, *synchronized/1*

built-in predicate

threaded_call/1-2

Description

```
threaded_call(Goal)
threaded_call(Goal, Tag)
```

Proves `Goal` asynchronously using a new thread. The argument can be a message-sending goal. Calls to this predicate always succeed and return immediately. The results (success, failure, or exception) are sent back to the message queue of the object containing the call (*this*) and can be retrieved by calling the *threaded_exit/1* predicate.

The `threaded_call/2` variant returns a threaded call identifier tag that can be used with the *threaded_exit/2* and *threaded_cancel/1* predicates. Tags shall be regarded as opaque terms; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Meta-predicate template

```
threaded_call(0)
threaded_call(0, *)
```

Modes and number of proofs

```
threaded_call(@callable) - one
threaded_call(@callable, --nonvar) - one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
`Tag` is not a variable:
    `uninstantiation_error(Tag)`

Examples

Prove `Goal` asynchronously in a new thread:
    `threaded_call(Goal)`
Prove `::Message` asynchronously in a new thread:
    `threaded_call(::Message)`
Prove `Object::Message` asynchronously in a new thread:
    `threaded_call(Object::Message)`

See also

threaded_exit/1-2, *threaded_ignore/1*, *threaded_once/1-2*, *threaded_peek/1-2*, *threaded_cancel/1*, *threaded/1*, *synchronized/1*

built-in predicate

threaded_once/1-2

Description

```
threaded_once(Goal)
threaded_once(Goal, Tag)
```

Proves `Goal` asynchronously using a new thread. Only the first goal solution is found. The argument can be a message-sending goal. This call always succeeds. The result (success, failure, or exception) is sent back to the message queue of the object containing the call (*this*).

The `threaded_once/2` variant returns a threaded call identifier tag that can be used with the *threaded_exit/2* and *threaded_cancel/1* predicates. Tags shall be regarded as opaque terms; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Meta-predicate template

```
threaded_once(0)
threaded_once(0, *)
```

Modes and number of proofs

```
threaded_once(@callable) - one
threaded_once(@callable, --nonvar) - one
```

Errors

Goal is a variable:
    `instantiation_error`
Goal is neither a variable nor a callable term:
    `type_error(callable, Goal)`
Tag is not a variable:
    `uninstantiation_error(Tag)`

Examples

Prove `Goal` asynchronously in a new thread:

```
threaded_once(Goal)
```

Prove `::Message` asynchronously in a new thread:

```
threaded_once(::Message)
```

Prove `Object::Message` asynchronously in a new thread:

```
threaded_once(Object::Message)
```

See also

threaded_call/1-2, *threaded_exit/1-2*, *threaded_ignore/1*, *threaded_peek/1-2*, *threaded_cancel/1*, *threaded/1*, *synchronized/1*

built-in predicate

threaded_ignore/1

Description

```
threaded_ignore(Goal)
```

Proves `Goal` asynchronously using a new thread. Only the first goal solution is found. The argument can be a message-sending goal. This call always succeeds, independently of the result (success, failure, or exception), which is simply discarded instead of being sent back to the message queue of the object containing the call (*this*).

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Meta-predicate template

```
threaded_ignore(0)
```

Modes and number of proofs

```
threaded_ignore(@callable) - one
```

Errors

`Goal` is a variable:
`instantiation_error`
`Goal` is neither a variable nor a callable term:
`type_error(callable, Goal)`

Examples

Prove `Goal` asynchronously in a new thread:
`threaded_ignore(Goal)`
Prove `::Message` asynchronously in a new thread:
`threaded_ignore(::Message)`
Prove `Object::Message` asynchronously in a new thread:
`threaded_ignore(Object::Message)`

See also

threaded_call/1-2, *threaded_exit/1-2*, *threaded_once/1-2*, *threaded_peek/1-2*, *threaded/1*, *synchronized/1*

built-in predicate

threaded_exit/1-2

Description

```
threaded_exit(Goal)
threaded_exit(Goal, Tag)
```

Retrieves the result of proving `Goal` in a new thread. This predicate blocks execution until the reply is sent to the *this* message queue by the thread executing the goal. When there is no thread proving the goal, the predicate generates an exception. This predicate is non-deterministic, providing access to any alternative solutions of its argument.

The argument of this predicate should be a *variant* of the argument of the corresponding *threaded_call/1* or *threaded_once/1* call. When the predicate argument is subsumed by the `threaded_call/1` or `threaded_once/1` call argument, the `threaded_exit/1` call will succeed iff its argument is a solution of the (more general) goal.

The `threaded_exit/2` variant accepts a threaded call identifier tag generated by the calls to the *threaded_call/2* and *threaded_once/2* predicates. Tags shall be regarded as an opaque term; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_exit(+callable) - zero_or_more
threaded_exit(+callable, +nonvar) - zero_or_more
```

Errors

`Goal` is a variable:
`instantiation_error`
`Goal` is neither a variable nor a callable term:
`type_error(callable, Goal)`
No thread is running for proving `Goal` in the `Object` calling context:
`existence_error(thread, Object)`
`Tag` is a variable:
`instantiation_error`

Examples

To retrieve an asynchronous goal proof result:
`threaded_exit(Goal)`
To retrieve an asynchronous message to *self* result:
`threaded_exit(::Goal)`
To retrieve an asynchronous message result:
`threaded_exit(Object::Goal)`

See also

threaded_call/1-2, *threaded_ignore/1*, *threaded_once/1-2*, *threaded_peek/1-2*, *threaded_cancel/1*, *threaded/1*

built-in predicate

threaded_peek/1-2

Description

```
threaded_peek(Goal)
threaded_peek(Goal, Tag)
```

Checks if the result of proving `Goal` in a new thread is already available. This call succeeds or fails without blocking execution waiting for a reply to be available. When there is no thread proving the goal, the predicate generates an exception.

The argument of this predicate should be a *variant* of the argument of the corresponding *threaded_call/1* or *threaded_once/1* call. When the predicate argument is subsumed by the `threaded_call/1` or `threaded_once/1` call argument, the `threaded_peek/1` call will succeed iff its argument unifies with an already available solution of the (more general) goal.

The `threaded_peek/2` variant accepts a threaded call identifier tag generated by the calls to the *threaded_call/2* and *threaded_once/2* predicates. Tags shall be regarded as an opaque term; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_peek(+callable) - zero_or_one
threaded_peek(+callable, +nonvar) - zero_or_one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
No thread is running for proving `Goal` in the `Object` calling context:
    `existence_error(thread, Object)`
`Tag` is a variable:
    `instantiation_error`

Examples

To check for an asynchronous goal proof result:

```
threaded_peek(Goal)
```

To check for an asynchronous message to *self* result:

```
threaded_peek(::Goal)
```

To check for an asynchronous message result:

```
threaded_peek(Object::Goal)
```

See also

threaded_call/1-2, *threaded_exit/1-2*, *threaded_ignore/1*, *threaded_once/1-2*, *threaded_cancel/1*, *threaded/1*

built-in predicate

threaded_cancel/1

Description

```
threaded_cancel(Tag)
```

Cancels a tagged threaded call. When there is no asynchronous call with the given tag, calling this predicate succeeds assuming the asynchronous call has already terminated or canceled. The threaded call identifier tag is generated by calls to the *threaded_call/2* and *threaded_once/2* predicates. Tags shall be regarded as an opaque term; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_cancel(+nonvar) - one
```

Errors

`Tag` is a variable:
    `instantiation_error`

Examples

(none)

See also

threaded_call/1-2, *threaded_exit/1-2*, *threaded_ignore/1*, *threaded_once/1-2*, *threaded/1*

built-in predicate

threaded_wait/1

Description

```
threaded_wait(Term)
threaded_wait([Term| Terms])
```

Suspends the thread making the call until a notification is received that unifies with `Term`. The call must be made within the same object (*this*) containing the calls to the *threaded_notify/1* predicate that will eventually send the notification. The argument may also be a list of notifications, `[Term| Terms]`. In this case, the thread making the call will suspend until all notifications in the list are received.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_wait(?term) - one
threaded_wait(+list(term)) - one
```

Errors

(none)

Examples

```
% wait until the "data_available" notification is received:
..., threaded_wait(data_available), ...
```

See also

threaded_notify/1

built-in predicate

threaded_notify/1

Description

```
threaded_notify(Term)
threaded_notify([Term| Terms])
```

Sends `Term` as a notification to any thread suspended waiting for it in order to proceed. The call must be made within the same object (*this*) containing the calls to the *threaded_wait/1* predicate waiting for the notification. The argument may also be a list of notifications, `[Term| Terms]`. In this case, all notifications in the list will be sent to any threads suspended waiting for them in order to proceed.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *threads* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_notify(@term) - one
threaded_notify(@list(term)) - one
```

Errors

(none)

Examples

```
% send a "data_available" notification:
..., threaded_notify(data_available), ...
```

See also

threaded_wait/1

2.4.8 Multi-threading engines

built-in predicate

threaded_engine_create/3

Description

```
threaded_engine_create(AnswerTemplate, Goal, Engine)
```

Creates a new engine for proving the given goal and defines an answer template for retrieving the goal solution bindings. A message queue for passing arbitrary terms to the engine is also created. If the name for the engine is not given, a unique name is generated and returned. Engine names shall be regarded as opaque terms; users shall not rely on its type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Meta-predicate template

```
threaded_engine_create(*, 0, *)
```

Modes and number of proofs

```
threaded_engine_create(@term, @callable, @nonvar) - one
threaded_engine_create(@term, @callable, --nonvar) - one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
`Engine` is the name of an existing engine:
    `permission_error(create, engine, Engine)`

Examples

```
% create a new engine to enumerate list elements:
| ?- threaded_engine_create(X, member(X, [1,2,3]), worker_1).
```

See also

threaded_engine_destroy/1, *threaded_engine_self/1*, *threaded_engine/1*, *threaded_engine_next/2*, *threaded_engine_next_reified/2*

built-in predicate

threaded_engine_destroy/1

Description

```
threaded_engine_destroy(Engine)
```

Stops and destroys an engine.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_destroy(@nonvar) - one
```

Errors

`Engine` is a variable:
    `instantiation_error`
`Engine` is neither a variable nor the name of an existing engine:
    `existence_error(engine, Engine)`

Examples

```
% stop the worker_1 engine:
| ?- threaded_engine_destroy(worker_1).

% stop all engines:
| ?- forall(
        threaded_engine(Engine),
        threaded_engine_destroy(Engine)
    ).
```

See also

threaded_engine_create/3, *threaded_engine_self/1*, *threaded_engine/1*

built-in predicate

threaded_engine/1

Description

```
threaded_engine(Engine)
```

Enumerates, by backtracking, all existing engines. Engine names shall be regarded as opaque terms; users shall not rely on their type.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine(?nonvar) - zero_or_more
```

Errors

(none)

Examples

```
% check that the worker_1 engine exists:
| ?- threaded_engine(worker_1).

% write the names of all existing engines:
| ?- forall(
        threaded_engine(Engine),
        (writeq(Engine), nl)
    ).
```

See also

threaded_engine_create/3, *threaded_engine_self/1*, *threaded_engine_destroy/1*

built-in predicate

threaded_engine_self/1

Description

```
threaded_engine_self(Engine)
```

Queries the name of engine calling the predicate. Fails if not called from within an engine or if the argument doesn't unify with the engine name.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_self(?nonvar) - zero_or_one
```

Errors

(none)

Examples

```
% find the name of the engine making the query:
..., threaded_engine_self(Engine), ...

% check if the the engine making the query is worker_1:
..., threaded_engine_self(worker_1), ...
```

See also

threaded_engine_create/3, *threaded_engine_destroy/1*, *threaded_engine/1*

built-in predicate

threaded_engine_next/2

Description

```
threaded_engine_next(Engine, Answer)
```

Retrieves an answer from an engine and signals it to start computing the next answer. This predicate blocks until an answer becomes available. The predicate fails when there are no more solutions to the engine goal. If the engine goal throws an exception, calling this predicate will re-throw the exception and subsequent calls will fail.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_next(@nonvar, ?term) - zero_or_one
```

Errors

`Engine` is a variable:
    `instantiation_error`
`Engine` is neither a variable nor the name of an existing engine:
    `existence_error(engine, Engine)`

Examples

```
% get the next answer from the worker_1 engine:
| ?- threaded_engine_next(worker_1, Answer).
```

See also

threaded_engine_create/3, *threaded_engine_next_reified/2*, *threaded_engine_yield/1*

built-in predicate

threaded_engine_next_reified/2

Description

```
threaded_engine_next_reified(Engine, Answer)
```

Retrieves an answer from an engine and signals it to start computing the next answer. This predicate always succeeds and blocks until an answer becomes available. Answers are returned using the terms `the(Answer)`, `no`, and `exception(Error)`.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_next_reified(@nonvar, ?nonvar) - one
```

Errors

`Engine` is a variable:
    `instantiation_error`
`Engine` is neither a variable nor the name of an existing engine:
    `existence_error(engine, Engine)`

Examples

```
% get the next reified answer from the worker_1 engine:
| ?- threaded_engine_next_reified(worker_1, Answer).
```

See also

threaded_engine_create/3, *threaded_engine_next/2*, *threaded_engine_yield/1*

built-in predicate

threaded_engine_yield/1

Description

```
threaded_engine_yield(Answer)
```

Returns an answer independent of the solutions of the engine goal. Fails if not called from within an engine. This predicate is usually used when the engine goal is a call to a recursive predicate processing terms from the engine term queue.

This predicate blocks until the returned answer is consumed.

Note that this predicate should not be called as the last element of a conjunction resulting in an engine goal solution as, in this case, an answer will always be returned. For example, instead of `(threaded_engine_yield(ready); member(X,[1,2,3]))` use `(X=ready; member(X,[1,2,3]))`.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_yield(@term) - zero_or_one
```

Errors

(none)

Examples

```
% returns the atom "ready" as an engine answer:
..., threaded_engine_yield(ready), ...
```

See also

threaded_engine_create/3, *threaded_engine_next/2*, *threaded_engine_next_reified/2*

built-in predicate

threaded_engine_post/2

Description

```
threaded_engine_post(Engine, Term)
```

Posts a term to the engine term queue.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_post(@nonvar, @term) - one
```

Errors

`Engine` is a variable:

```
instantiation_error
```

`Engine` is neither a variable nor the name of an existing engine:

```
existence_error(engine, Engine)
```

Examples

```
% post the atom "ready" to the worker_1 engine queue:
| ?- threaded_engine_post(worker_1, ready).
```

See also

threaded_engine_fetch/1

built-in predicate

threaded_engine_fetch/1

Description

```
threaded_engine_fetch(Term)
```

Fetches a term from the engine term queue. Blocks until a term is available. Fails if not called from within an engine.

Note

This predicate requires a *backend Prolog compiler* providing compatible multi-threading primitives. The value of the read-only *engines* flag is set to `supported` when that is the case.

Modes and number of proofs

```
threaded_engine_fetch(?term) - zero_or_one
```

Errors

(none)

Examples

```
% fetch a term from the engine term queue:
..., threaded_engine_fetch(Term), ...
```

See also

threaded_engine_post/2

2.4.9 Compiling and loading source files

built-in predicate

logtalk_compile/1

Description

```
logtalk_compile(File)
logtalk_compile(Files)
```

Compiles to disk a *source file* or a list of source files using the default compiler flag values. The Logtalk source file name extension (by default, `.lgt`) can be omitted. Source file paths can be absolute, relative to the current directory, or use *library notation*. This predicate can also be used to compile Prolog source files as Logtalk source code. When no recognized Logtalk or Prolog extension is specified, the compiler tries first to append a Logtalk source file extension and then a Prolog source file extension. If that fails, the compiler tries to use the file name as-is. The recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*.

Note

This predicate does not load into memory the compiled source file. If you want to both compile and load a source file, use instead the *logtalk_load/1* built-in predicate.

When this predicate is called from the top-level interpreter, relative source file paths are resolved using the current working directory. When the calls are made from a source file, relative source file paths are resolved using the source file directory.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails on the first error found during compilation of a source file. In this case, no file with the compiled code is written to disk.

Modes and number of proofs

```
logtalk_compile(@source_file_name) - zero_or_one
logtalk_compile(@list(source_file_name)) - zero_or_one
```

Errors

`File` is a variable:
    `instantiation_error`
`Files` is a variable or a list with an element which is a variable:
    `instantiation_error`
`File`, or an element `File` of the `Files` list, is neither a variable nor a source file name:
    `type_error(source_file_name, File)`
`File`, or an element `File` of the `Files` list, uses library notation but the library does not exist:
    `existence_error(library, Library)`
`File` or an element `File` of the `Files` list does not exist:
    `existence_error(file, File)`

Examples

```
% compile to disk the "set" source file in the
% current directory:
| ?- logtalk_compile(set).

% compile to disk the "tree" source file in the
% "types" library directory:
| ?- logtalk_load(types(tree)).

% compile to disk the "listp" and "list" source
% files in the current directory:
| ?- logtalk_compile([listp, list]).
```

See also

logtalk_compile/2, *logtalk_load/1*, *logtalk_load/2*, *logtalk_make/0*, *logtalk_make/1*, *logtalk_library_path/2*

built-in predicate

logtalk_compile/2

Description

```
logtalk_compile(File, Flags)
logtalk_compile(Files, Flags)
```

Compiles to disk a *source file* or a list of source files using a list of compiler flags. The Logtalk source file name extension (by default, `.lgt`) can be omitted. Source file paths can be absolute, relative to the current directory, or use *library notation*. This predicate can also be used to compile Prolog source files as Logtalk source code. When no recognized Logtalk or Prolog extension is specified, the compiler tries first to append a Logtalk source file extension and then a Prolog source file extension. If that fails, the compiler tries to use the file name as-is. Compiler flags are represented as *flag(value)*. For a description of the available compiler flags, please see the *Compiler flags* section in the User Manual. The recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*.

Note

This predicate does not load into memory the compiled source file. If you want to both compile and load a source file, use instead the *logtalk_load/2* built-in predicate.

When this predicate is called from the top-level interpreter, relative source file paths are resolved using the current working directory. When the calls are made from a source file, relative source file paths are resolved by default using the source file directory (unless a *relative_to* flag is passed).

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails on the first error found during compilation of a source file. In this case, no file with the compiled code is written to disk.

Warning

The compiler flags specified in the second argument only apply to the files listed in the first argument. Notably, if you are compiling a *loader file*, the flags only apply to the loader file itself.

Modes and number of proofs

```
logtalk_compile(@source_file_name, @list(compiler_flag)) - zero_or_one
logtalk_compile(@list(source_file_name), @list(compiler_flag)) - zero_or_one
```

Errors

`File` is a variable:
    `instantiation_error`
`Files` is a variable or a list with an element which is a variable:
    `instantiation_error`
`File`, or an element `File` of the `Files` list, is neither a variable nor a source file name:
    `type_error(source_file_name, File)`
`File`, or an element `File` of the `Files` list, uses library notation but the library does not exist:

`existence_error(library, Library)`

`File` or an element `File` of the `Files` list, does not exist:

`existence_error(file, File)`

`Flags` is a variable or a list with an element which is a variable:

`instantiation_error`

`Flags` is neither a variable nor a proper list:

`type_error(list, Flags)`

An element `Flag` of the `Flags` list is not a valid compiler flag:

`type_error(compiler_flag, Flag)`

An element `Flag` of the `Flags` list defines a value for a read-only compiler flag:

`permission_error(modify, flag, Flag)`

An element `Flag` of the `Flags` list defines an invalid value for a flag:

`domain_error(flag_value, Flag+Value)`

Examples

```
% compile to disk the "list" source file in the
% current directory using default compiler flags:
| ?- logtalk_compile(list, []).

% compile to disk the "tree" source file in the "types"
% library directory with the source_data flag turned on:
| ?- logtalk_compile(types(tree), [source_data(on)]).

% compile to disk the "file_system" source file in the
% current directory with portability warnings suppressed:
| ?- logtalk_compile(file_system, [portability(silent)]).
```

See also

logtalk_compile/1, *logtalk_load/1*, *logtalk_load/2*, *logtalk_make/0*, *logtalk_make/1*, *logtalk_library_path/2*

built-in predicate

logtalk_load/1

Description

```
logtalk_load(File)
logtalk_load(Files)
```

Compiles to disk and then loads to memory a *source file* or a list of source files using the default compiler flag values. The Logtalk source file name extension (by default, `.lgt`) can be omitted. Source file paths can be absolute, relative to the current directory, or use *library notation*. This predicate can also be used to compile Prolog source files as Logtalk source code. When no recognized Logtalk or Prolog extension is specified, the

compiler tries first to append a Logtalk source file extension and then a Prolog source file extension. If that fails, the compiler tries to use the file name as-is. The recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*.

When this predicate is called from the top-level interpreter, relative source file paths are resolved using the current working directory. When the calls are made from a source file, relative source file paths are resolved using the source file directory.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails on the first error found during compilation of a source file. In this case, the source file contents is not loaded.

Depending on the *backend Prolog compiler*, the shortcuts `{File}` or `{File1, File2, ...}` may be used as an alternative. Check the adapter files for the availability of these shortcuts as they are not part of the language (and thus should only be used at the top-level interpreter).

Modes and number of proofs

```
logtalk_load(@source_file_name) - zero_or_one
logtalk_load(@list(source_file_name)) - zero_or_one
```

Errors

`File` is a variable:
    `instantiation_error`
`Files` is a variable or a list with an element which is a variable:
    `instantiation_error`
`File`, or an element `File` of the `Files` list, is neither a variable nor a source file name:
    `type_error(source_file_name, File)`
`File`, or an element `File` of the `Files` list, uses library notation but the library does not exist:
    `existence_error(library, Library)`
`File` or an element `File` of the `Files` list, does not exist:
    `existence_error(file, File)`

Examples

```
% compile and load the "set" source file in the
% current directory:
| ?- logtalk_load(set).

% compile and load the "tree" source file in the
% "types" library directory:
| ?- logtalk_load(types(tree)).

% compile and load the "listp" and "list" source
% files in the current directory:
| ?- logtalk_load([listp, list]).
```

See also

logtalk_compile/1, *logtalk_compile/2*, *logtalk_load/2*, *logtalk_make/0*, *logtalk_make/1*, *logtalk_library_path/2*

built-in predicate

logtalk_load/2

Description

```
logtalk_load(File, Flags)
logtalk_load(Files, Flags)
```

Compiles to disk and then loads to memory a *source file* or a list of source files using a list of compiler flags. The Logtalk source file name extension (by default, `.lgt`) can be omitted. Source file paths can be absolute, relative to the current directory, or use *library notation*. Compiler flags are represented as *flag(value)*. This predicate can also be used to compile Prolog source files as Logtalk source code. When no recognized Logtalk or Prolog extension is specified, the compiler tries first to append a Logtalk source file extension and then a Prolog source file extension. If that fails, the compiler tries to use the file name as-is. For a description of the available compiler flags, please see the *Compiler flags* section in the User Manual. The recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*. The recognized Logtalk and Prolog file extensions are defined in the *backend adapter files*.

When this predicate is called from the top-level interpreter, relative source file paths are resolved using the current working directory. When the calls are made from a source file, relative source file paths are resolved by default using the source file directory (unless a *relative_to* flag is passed).

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails on the first error found during compilation of a source file. In this case, the source file contents is not loaded.

Warning

The compiler flags specified in the second argument only apply to the files listed in the first argument and not to any files that those files may load or compile. Notably, if you are loading a *loader file*, the flags only apply to the loader file itself and not to the files loaded by it.

Modes and number of proofs

```
logtalk_load(@source_file_name, @list(compiler_flag)) - zero_or_one
logtalk_load(@list(source_file_name), @list(compiler_flag)) - zero_or_one
```

Errors

`File` is a variable:
    `instantiation_error`
`Files` is a variable or a list with an element which is a variable:
    `instantiation_error`
`File`, or an element `File` of the `Files` list, is neither a variable nor a source file name:
    `type_error(source_file_name, File)`
`File`, or an element `File` of the `Files` list, uses library notation but the library does not exist:
    `existence_error(library, Library)`
`File` or an element `File` of the `Files` list, does not exist:
    `existence_error(file, File)`
`Flags` is a variable or a list with an element which is a variable:
    `instantiation_error`
`Flags` is neither a variable nor a proper list:
    `type_error(list, Flags)`
An element `Flag` of the `Flags` list is not a valid compiler flag:
    `type_error(compiler_flag, Flag)`
An element `Flag` of the `Flags` list defines a value for a read-only compiler flag:
    `permission_error(modify, flag, Flag)`
An element `Flag` of the `Flags` list defines an invalid value for a flag:
    `domain_error(flag_value, Flag+Value)`

Examples

```
% compile and load the "list" source file in the
% current directory using default compiler flags:
| ?- logtalk_load(list, []).

% compile and load the "tree" source file in the "types"
% library directory with the source_data flag turned on:
| ?- logtalk_load(types(tree)).

% compile and load the "file_system" source file in the
% current directory with portability warnings suppressed:
| ?- logtalk_load(file_system, [portability(silent)]).
```

See also

logtalk_compile/1, *logtalk_compile/2*, *logtalk_load/1*, *logtalk_make/0*, *logtalk_make/1*, *logtalk_library_path/2*

built-in predicate

logtalk_make/0

Description

```
logtalk_make
```

Reloads all Logtalk source files that have been modified since the time they were last loaded. Only source files loaded using the *logtalk_load/1* and *logtalk_load/2* predicates are reloaded. Non-modified files will also be reloaded when a previous attempt to load them failed or when there is a change to the compilation mode (i.e., when the files were loaded without explicit *debug* or *optimize* flags and the default values of these flags changed after loading; no check is made, however, for other implicit compiler flags that may have changed since loading). When an included file is modified, this predicate reloads its main file (i.e., the file that contains the *include/1* directive).

Depending on the *backend Prolog compiler*, the shortcut `{*}` may be used as an alternative. Check the *adapter files* for the availability of the shortcut as it is not part of the language.

Warning

Only use the `{*}` shortcut at the top-level interpreter and never in source files.

This predicate can be extended by the user by defining clauses for the *logtalk_make_target_action/1* multifile and dynamic hook predicate using the argument `all`. The additional user defined actions are run after the default one.

Modes and number of proofs

```
logtalk_make - one
```

Errors

(none)

Examples

```
% reload all files modified since last loaded:
| ?- logtalk_make.
```

See also

logtalk_compile/1, *logtalk_compile/2*, *logtalk_load/1*, *logtalk_load/2*, *logtalk_make/1*, *logtalk_make_target_action/1*

built-in predicate

logtalk_make/1

Description

```
logtalk_make(Target)
```

Runs a make target. Prints a warning message and fails when the target is not valid.

Allows reloading all Logtalk source files that have been modified since last loaded when called with the target `all`, deleting all intermediate files generated by the compilation of Logtalk source files when called with the target `clean`, checking for code issues when called with the target `check`, listing of circular dependencies between pairs or trios of objects when called with the target `circular`, generating documentation when called with the target `documentation`, and deleting the *dynamic binding* caches with the target `caches`.

There are also three variants of the `all` target: `debug`, `normal`, and `optimal`. These targets change the compilation mode (by changing the default value of the *debug* and *optimize* flags) and reload all affected files (i.e., all files loaded without an explicit `debug/1` or `optimize/1` compiler option).

When using the `all` target, only source files loaded using the *logtalk_load/1* and *logtalk_load/2* predicates are reloaded. Non-modified files will also be reloaded when a previous attempt to load them failed or when there is a change to the compilation mode (i.e., when the files were loaded without explicit *debug* or *optimize* flags and the default values of these flags changed after loading; no check is made, however, for other implicit compiler flags that may have changed since loading). When an included file is modified, this target reloads its main file (i.e., the file that contains the *include/1* directive).

When using the `check` or `circular` targets, be sure to compile your source files with the *source_data* flag turned on for complete and detailed reports.

The `check` target scans for missing entities (objects, protocols, categories, and modules), missing entity predicates, and duplicated library aliases. Predicates for messages sent to objects that implement the forwarding built-in protocol are not reported. While this usually avoids only false positives, it may also result in failure to report true missing predicates in some cases.

When using the `circular` target, be prepared for a lengthy computation time for applications with a large combined number of objects and message calls. Only mutual and triangular dependencies are checked due to the computational cost. Circular dependencies occur when an object sends a message to a second object that, in turn, sends a message to the first object. These circular dependencies are often a consequence of lack of separation of concerns. But, when they cannot be fixed, the only practical consequence is a small performance cost as some of the messages would be forced to use dynamic binding.

The `documentation` target requires the `doclet` tool and a single *doclet object* to be loaded. See the `doclet` tool documentation for more details.

Depending on the *backend Prolog compiler*, the following top-level shortcuts are usually defined:

- `{*}` - `logtalk_make(all)`
- `{!}` - `logtalk_make(clean)`
- `{?}` - `logtalk_make(check)`
- `{@}` - `logtalk_make(circular)`
- `{#}` - `logtalk_make(documentation)`
- `{$}` - `logtalk_make(caches)`
- `{+d}` - `logtalk_make(debug)`
- `{+n}` - `logtalk_make(normal)`
- `{+o}` - `logtalk_make(optimal)`

Check the *adapter files* for the availability of these shortcuts as they are not part of the language.

> **Warning**
>
> Only use the shortcuts at the top-level interpreter and never in source files.

The target actions can be extended by defining clauses for the multifile and dynamic hook predicate *logtalk_make_target_action(Target)* where `Target` is one of the targets listed above. The additional user-defined actions are run after the default ones.

Modes and number of proofs

```
logtalk_make(+atom) - zero_or_one
```

Errors

(none)

Examples

```
% reload loaded source files in debug mode:
| ?- logtalk_make(debug).

% check for code issues in the loaded source files:
| ?- logtalk_make(check).

% delete all intermediate files generated by
% the compilation of Logtalk source files:
| ?- logtalk_make(clean).
```

> **See also**
>
> *logtalk_compile/1*, *logtalk_compile/2*, *logtalk_load/1*, *logtalk_load/2*, *logtalk_make/0*, *logtalk_make_target_action/1*

built-in predicate

logtalk_make_target_action/1

Description

```
logtalk_make_target_action(Target)
```

Multifile and dynamic hook predicate that allows defining user actions for the *logtalk_make/1* targets. The user defined actions are run after the default ones using a failure driven loop. This loop does not catch any exceptions thrown when calling the user-defined actions.

Modes and number of proofs

```
logtalk_make_target_action(+atom) - zero_or_more
```

Errors

(none)

Examples

```
% integrate the dead_code_scanner tool with logtalk_make/1

:- multifile(logtalk_make_target_action/1).
:- dynamic(logtalk_make_target_action/1).

logtalk_make_target_action(check) :-
    dead_code_scanner::all.
```

See also

logtalk_make/1, *logtalk_make/0*

built-in predicate

logtalk_library_path/2

Description

```
logtalk_library_path(Library, Path)
```

Dynamic and multifile user-defined predicate, allowing the declaration of aliases to *library* paths. Library aliases may also be used in the second argument (using the notation *alias(path)*). Paths must always end with the path directory separator character (`'/'`).

> **Warning**
>
> Library aliases should be unique. The *logtalk_make/1* built-in predicate can be used to detect and report duplicated library aliases using the `check` target.

Clauses for this predicate should preferably be facts. Defining rules to dynamically compute at runtime both library alias names and their paths, although sometimes handy, can inadvertently result in endless loops when those rules also attempt to expand library paths. When asserting facts for this predicate, use preferably `asserta/1` instead of `assertz/1` to help ensure the facts will be used before any rules.

Relative paths (e.g., `'../'` or `'./'`) should only be used within the *alias(path)*) notation so that library paths can always be expanded to absolute paths independently of the (usually unpredictable) current directory at the time the `logtalk_library_path/2` predicate is called.

When working with a relocatable application, the actual application installation directory can be retrieved by calling the *logtalk_load_context/2* predicate with the `directory` key and using the returned value to define the `logtalk_library_path/2` predicate. On a *settings file* file or a *loader file* file, simply use an *initialization/1* directive to wrap the call to the `logtalk_load_context/2` predicate and the assert of the `logtalk_library_path/2` fact.

This predicate may be used to override the default *scratch directory* by defining the library alias `scratch_directory` in a backend Prolog initialization file (assumed to be loaded prior to Logtalk loading). This allows e.g. Logtalk to be installed in a read-only directory by setting this alias to the operating-system directory for temporary files. It also allows several Logtalk instances to run concurrently without conflict by using a unique scratch directory per instance (e.g., using a process ID or a UUID generator).

This predicate may be used to override the default location used by the *packs* tool to store registries and packs by defining the `logtalk_packs` library alias in settings file or in a backend Prolog initialization file (assumed to be loaded prior to Logtalk loading).

The *logtalk* built-in object provides an expand_library_path/2 predicate that can be used to expand library aliases and files expressed using library notation.

Modes and number of proofs

```
logtalk_library_path(?atom, -atom) - zero_or_more
logtalk_library_path(?atom, -compound) - zero_or_more
```

Errors

(none)

Examples

```
:- initialization((
   logtalk_load_context(directory, Directory),
   asserta(logtalk_library_path(my_application_root, Directory))
)).
```

```
| ?- logtalk_library_path(viewpoints, Path).

Path = examples('viewpoints/')
yes

| ?- logtalk_library_path(Library, Path).

Library = home,
Path = '$HOME/' ;

Library = logtalk_home,
Path = '$LOGTALKHOME/' ;

Library = logtalk_user
Path = '$LOGTALKUSER/' ;

Library = examples
Path = logtalk_user('examples/') ;

Library = library
Path = logtalk_user('library/') ;

Library = viewpoints
Path = examples('viewpoints/')
yes
```

```
| ?- logtalk::expand_library_path(viewpoints, Path).

Path = '/Users/pmoura/logtalk/examples/viewpoints/'.
yes

| ?- logtalk::expand_library_path(viewpoints('loader.lgt'), Path).

Path = '/Users/pmoura/logtalk/examples/viewpoints/loader.lgt'.
yes
```

See also

logtalk_compile/1, *logtalk_compile/2*, *logtalk_load/1*, *logtalk_load/2*

built-in predicate

logtalk_load_context/2

Description

```
logtalk_load_context(Key, Value)
```

Provides access to the Logtalk compilation/loading context. The following keys are currently supported:

- `entity_identifier` - identifier of the entity being compiled if any
- `entity_prefix` - internal prefix for the entity compiled code
- `entity_type` - returns the value `module` when compiling a module as an object
- `entity_relation` - returns the entity relations as declared in the entity opening directive
- `source` - full path of the source file being compiled
- `file` - the actual file being compiled, different from `source` only when processing an `include/1` directive
- `basename` - source file basename
- `directory` - source file directory
- `stream` - input stream being used to read source file terms
- `target` - the full path of the intermediate Prolog file
- `flags` - the list of the explicit flags used for the compilation of the source file
- `term` - the source file term being compiled
- `term_position` - the position of the term being compiled (`StartLine-EndLine`)
- `variables` - the variables of the term being compiled (`[Variable1, ...]`)
- `variable_names` - the variable names of the term being compiled (`[Name1=Variable1, ...]`)
- `variable_names(Term)` - the variable names of the term being compiled (`[Name1=Variable1, ...]`)
- `singletons` - the singleton variables of the term being compiled (`[Name1=Variable1, ...]`)
- `singletons(Term)` - the singleton variables of the term being compiled (`[Name1=Variable1, ...]`)
- `parameter_variables` - list of parameter variable names and positions (`[Name1-Position1, ...]`)

For the `entity_relation` key, the possible values are:

- `extends_protocol(Protocol, ParentProtocol, Scope)`
- `implements_protocol(ObjectOrCategory, Protocol, Scope)`
- `extends_category(Category, ParentCategory, Scope)`
- `imports_category(Object, Category, Scope)`
- `extends_object(Prototype, Parent, Scope)`
- `instantiates_class(Instance, Class, Scope)`
- `specializes_class(Class, Superclass, Scope)`
- `complements_object(Category, Object)`

Calling this predicate with the `parameter_variables` key only succeeds when compiling a parametric entity containing parameter variables.

This predicate is usually called by the *term_expansion/2* and *goal_expansion/2* methods. It can also be called directly from *initialization/1* directives in a source file. Note that the entity keys are only available when compiling an entity term or from an object `initialization/1` directive.

> **Warning**
>
> The `term_position` key is only supported in *backend Prolog compilers* that provide access to the start and end lines of a read term. When such support is not available, the value `-1` is returned for both the start and the end lines.
>
> Variables in the values of the `term`, `variables`, `variable_names`, and `singletons` keys are not shared with, respectively, the term and goal arguments of the `term_expansion/2` and `goal_expansion/2` methods. Use instead the `variable_names(Term)` and `singletons(Term)` keys.

Modes and number of proofs

```
logtalk_load_context(?callable, -nonvar) - zero_or_more
```

Errors

`Key` is neither a variable nor a callable term:

```
type_error(callable, Key)
```

`Key` is a callable term but not a valid key:

```
domain_error(logtalk_load_context_key, Key)
```

Examples

```
% expand source file terms only if they are entity terms
term_expansion(Term, ExpandedTerms) :-
    logtalk_load_context(entity_identifier, _),
    ....

% expand source file term while accessing its variable names
term_expansion(Term, ExpandedTerms) :-
    logtalk_load_context(variable_names(Term), VariableNames),
    ....

% define a library alias based on the source directory
:- initialization((
    logtalk_load_context(directory, Directory),
    assertz(logtalk_library_path(my_app, Directory))
)).
```

See also

term_expansion/2, *goal_expansion/2*, *initialization/1*

2.4.10 Flags

built-in predicate

current_logtalk_flag/2

Description

```
current_logtalk_flag(Flag, Value)
```

Enumerates, by backtracking, the current Logtalk flag values. For a description of the predefined compiler flags, please see the *Compiler flags* section in the User Manual.

Modes and number of proofs

```
current_logtalk_flag(?atom, ?atom) - zero_or_more
```

Errors

`Flag` is neither a variable nor an atom:
    `type_error(atom, Flag)`
`Flag` is an atom but an invalid flag:
    `domain_error(flag, Value)`

Examples

```
% get the current value of the source_data flag:
| ?- current_logtalk_flag(source_data, Value).
```

See also

create_logtalk_flag/3, *set_logtalk_flag/2*

built-in predicate

set_logtalk_flag/2

Description

```
set_logtalk_flag(Flag, Value)
```

Sets global, default, flag values. For local flag scope, use the corresponding *set_logtalk_flag/2* directive. To set a global flag value when compiling and loading a source file, wrap the calls to this built-in predicate with an *initialization/1* directive. For a description of the predefined compiler flags, please see the *Compiler flags* section in the User Manual.

Modes and number of proofs

```
set_logtalk_flag(+atom, +nonvar) - one
```

Errors

`Flag` is a variable:
    `instantiation_error`
`Value` is a variable:
    `instantiation_error`
`Flag` is neither a variable nor an atom:
    `type_error(atom, Flag)`
`Flag` is an atom but an invalid flag:
    `domain_error(flag, Flag)`
`Value` is not a valid value for flag `Flag`:
    `domain_error(flag_value, Flag + Value)`
`Flag` is a read-only flag:
    `permission_error(modify, flag, Flag)`

Examples

```
% turn off globally and by default the compiler
% unknown entities warnings:
| ?- set_logtalk_flag(unknown_entities, silent).
```

See also

create_logtalk_flag/3, *current_logtalk_flag/2*

built-in predicate

create_logtalk_flag/3

Description

```
create_logtalk_flag(Flag, Value, Options)
```

Creates a new Logtalk flag and sets its default value. User-defined flags can be queried and set in the same way as predefined flags by using, respectively, the *current_logtalk_flag/2* and *set_logtalk_flag/2* built-in predicates. For a description of the predefined compiler flags, please see the *Compiler flags* section in the User Manual.

This predicate is based on the specification of the SWI-Prolog `create_prolog_flag/3` built-in predicate and supports the same options: `access(Access)`, where `Access` can be either `read_write` (the default) or `read_only`; `keep(Keep)`, where `Keep` can be either `false` (the default) or `true`, for deciding if an existing definition of the flag should be kept or replaced by the new one; and `type(Type)` for specifying the type of the flag, which can be `boolean`, `atom`, `integer`, `float`, or `term` (which only restricts the flag value to ground terms). When the `type/1` option is not specified, the type of the flag is inferred from its initial value.

Modes and number of proofs

```
create_logtalk_flag(+atom, +ground, +list(ground)) - one
```

Errors

`Flag` is a variable:
    `instantiation_error`
`Value` is not a ground term:
    `instantiation_error`
`Options` is not a ground term:
    `instantiation_error`
`Flag` is neither a variable nor an atom:
    `type_error(atom, Flag)`
`Options` is neither a variable nor a list:
    `type_error(atom, Flag)`
`Value` is not a valid value for flag Flag:
    `domain_error(flag_value, Flag + Value)`
`Flag` is a system-defined flag:
    `permission_error(modify, flag, Flag)`
An element `Option` of the list `Options` is not a valid option
    domain_error(flag_option,Option)
The list `Options` contains a `type(Type)` option and `Value` is not of type `Type`
    type_error(Type, Value)

Examples

```
% create a new boolean flag with default value set to false:
| ?- create_logtalk_flag(pretty_print_blobs, false, []).
```

See also

current_logtalk_flag/2, *set_logtalk_flag/2*

2.4.11 Linter

built-in predicate

logtalk_linter_hook/7

Description

```
logtalk_linter_hook(Goal, Flag, File, Lines, Type, Entity, Warning)
```

Multifile user-defined predicate, supporting the definition of custom linter warnings. Experimental. The `Goal` argument can be a message-sending goal, `Object::Message`, a call to a Prolog built-in predicate, or a call to a module predicate, `Module:Predicate`. The `Flag` argument must be a supported linter flag. The `Warning` argument must be a valid `core` message term. For a given `Goal`, only the first successful call to this predicate is considered.

Modes and number of proofs

```
logtalk_linter_hook(@callable, +atom, +atom, +pair(integer), +atom, @object_identifier, --
↪callable) - zero_or_one
```

Errors

(none)

Examples

```
:- multifile(user::logtalk_linter_hook/7).
% warn about using list::append/3 to construct a list from an head and a tail
user::logtalk_linter_hook(
    list::append(L1,L2,L), suspicious_calls,
    File, Lines, Type, Entity,
    suspicious_call(File, Lines, Type, Entity, list::append(L1,L2,L), [L=[Head|L2]])
) :-
    nonvar(L1),
    L1 = [Head].
```

2.5 Built-in methods

2.5.1 Logic and control

built-in method

`!/0`

Description

```
!
```

Always succeeds with the side-effect of discarding choice-points. See also the ISO Prolog standard definition. This built-in method is declared as a public method and can be used as a message to an object.

Modes and number of proofs

```
! - one
```

Errors

(none)

Examples

(none)

See also

true/0, *fail/0*, *false/0*, *repeat/0*

built-in method

`true/0`

Description

```
true
```

Always succeeds. See also the ISO Prolog standard definition. This built-in method is declared as a public method and can be used as a message to an object.

Modes and number of proofs

```
true - one
```

Errors

(none)

Examples

(none)

See also

!/0, *fail/0*, *false/0*, *repeat/0*

built-in method

`fail/0`

Description

```
fail
```

Always fails. See also the ISO Prolog standard definition. This built-in method is declared as a public method and can be used as a message to an object.

Modes and number of proofs

```
fail - one
```

Errors

(none)

Examples

(none)

See also

!/0, *true/0*, *false/0*, *repeat/0*

built-in method

`false/0`

Description

```
false
```

Always fails. See also the ISO Prolog standard definition. This built-in method is declared as a public method and can be used as a message to an object.

Modes and number of proofs

```
false - one
```

Errors

(none)

Examples

(none)

See also

!/0, *true/0*, *fail/0*, *repeat/0*

built-in method

`repeat/0`

Description

```
repeat
```

Always succeeds when called and when backtracking into its call with an infinite number of choice-points. See also the ISO Prolog standard definition. This built-in method is declared as a public method and can be used as a message to an object.

Modes and number of proofs

```
repeat - one_or_more
```

Errors

(none)

Examples

(none)

See also

!/0, *true/0*, *fail/0*, *false/0*

2.5.2 Execution context

built-in method

`context/1`

Description

```
context(Context)
```

Returns the execution context for a predicate clause using the term `logtalk(Head,ExecutionContext)` where `Head` is the head of the clause containing the call. This private predicate is mainly used for providing an error context when type-checking predicate arguments. The `ExecutionContext` term should be regarded as an opaque term, which can be decoded using the logtalk::execution_context/7 predicate. Calls to this predicate are inlined at compilation time.

Warning

As the execution context term includes the clause head, the head of the clause calling the `context(Context)` method cannot contain the `Context` variable as that would result in the creation of a cyclic term. The compiler detects and reports any offending clauses by throwing a `representation_error(acyclic_term)` error.

Modes and number of proofs

```
context(--callable) - one
```

Errors

(none)

Examples

```
foo(A, N) :-
    % type-check arguments
    context(Context),
    type::check(atom, A, Context),
    type::check(integer, N, Context),
    % arguments are fine; go ahead
    ... .
```

See also

parameter/2, *self/1*, *sender/1*, *this/1*

built-in method

parameter/2

Description

```
parameter(Number, Term)
```

Used in *parametric objects* (and parametric categories), this private method provides runtime access to the parameter values of the entity that contains the predicate clause whose body is being executed by using the argument number in the entity identifier. This predicate is implemented as a unification between its second argument and the corresponding implicit execution-context argument in the predicate clause making the call. This unification occurs at the clause head when the second argument is not instantiated (the most common case). When the second argument is instantiated, the unification must be delayed to runtime and thus occurs at the clause body.

Entity parameters can also be accessed using *parameter variables*, which use the syntax `_VariableName_`. The compiler recognizes occurrences of these variables in directives and clauses. Parameter variables allow us to abstract parameter positions, thus simplifying code maintenance.

Modes and number of proofs

```
parameter(+integer, ?term) - zero_or_one
```

Errors

`Number` is a variable:
`instantiation_error`
`Number` is neither a variable nor an integer value:
`type_error(integer, Number)`
`Number` is smaller than one or greater than the parametric entity identifier arity:
`domain_error(out_of_range, Number)`
`Entity` identifier is not a compound term:
`type_error(compound, Entity)`

Examples

```
:- object(box(_Color, _Weight)).

    ...

    % this clause is translated into
    % a fact upon compilation
    color(Color) :-
        parameter(1, Color).

    % upon compilation, the >/2 call will be
    % the single goal in the clause body
    heavy :-
        parameter(2, Weight),
        Weight > 10.
    ...
```

The same example using *parameter variables*:

```
:- object(box(_Color_, _Weight_)).

    ...

    color(_Color_).

    heavy :-
        _Weight_ > 10.

    ...
```

See also

context/1, *self/1*, *sender/1*, *this/1*

built-in method

self/1

Description

```
self(Self)
```

Unifies its argument with the object that received the message under processing.

This private method is compiled into a unification between its argument and the corresponding implicit context argument in the predicate clause making the call. This unification occurs at the clause head when the argument is not bound at compile-time (the most common case).

Modes and number of proofs

```
self(?object_identifier) - zero_or_one
```

Errors

(none)

Examples

```
% upon compilation, the write/1 call will be
% the first goal in the clause body
test :-
    self(Self),
    write('executing a method in behalf of '),
    writeq(Self), nl.
```

See also

context/1, *parameter/2*, *sender/1*, *this/1*

built-in method

sender/1

Description

```
sender(Sender)
```

Unifies its argument with the object that sent the message under processing.

This private method is translated into a unification between its argument and the corresponding implicit context argument in the predicate clause making the call. This unification occurs at the clause head when the argument is not bound at compile-time (the most common case).

Modes and number of proofs

```
sender(?object_identifier) - zero_or_one
```

Errors

(none)

Examples

```
% after compilation, the write/1 call will
% be the first goal in the clause body
test :-
    sender(Sender),
    write('executing a method to answer a message sent by '),
    writeq(Sender), nl.
```

See also

context/1, *parameter/2*, *self/1*, *this/1*

built-in method

this/1

Description

```
this(This)
```

Unifies its argument with the identifier of the object calling this method. When this method is called from a category, the argument is unified with the object importing the category on whose behalf the clause containing the call is being used to prove the current goal.

This private method is implemented as a unification between its argument and the corresponding implicit execution-context argument in the predicate clause making the call. This unification occurs at the clause head when the argument is not bound at compile-time (the most common case).

This method is useful for avoiding hard-coding references to an object identifier or for retrieving all object parameters with a single call when using parametric objects.

Modes and number of proofs

```
this(?object_identifier) - zero_or_one
```

Errors

(none)

Examples

```
% after compilation, the write/1 call will
% be the first goal in the clause body
test :-
    this(This),
    write('Using a predicate clause contained in '),
    writeq(This), nl.
```

See also

context/1, *parameter/2*, *self/1*, *sender/1*

2.5.3 Reflection

built-in method

current_op/3

Description

```
current_op(Priority, Specifier, Operator)
```

Enumerates, by backtracking, the visible operators declared for an object. Operators not declared using a scope directive are not enumerated.

Modes and number of proofs

```
current_op(?operator_priority, ?operator_specifier, ?atom) - zero_or_more
```

Errors

`Priority` is neither a variable nor an integer:
`type_error(integer, Priority)`
`Priority` is an integer but not a valid operator priority:
`domain_error(operator_priority, Priority)`
`Specifier` is neither a variable nor an atom:
`type_error(atom, Specifier)`
`Specifier` is an atom but not a valid operator specifier:
`domain_error(operator_specifier, Specifier)`
`Operator` is neither a variable nor an atom:
`type_error(atom, Operator)`

Examples

To enumerate, by backtracking, the local operators or the operators visible in *this*:
`current_op(Priority, Specifier, Operator)`
To enumerate, by backtracking, the public and protected operators visible in *self*:
`::current_op(Priority, Specifier, Operator)`
To enumerate, by backtracking, the public operators visible for an explicit object:
`Object::current_op(Priority, Specifier, Operator)`

See also

current_predicate/1, *predicate_property/2*, *op/3*

built-in method

current_predicate/1

Description

```
current_predicate(Predicate)
```

Enumerates, by backtracking, visible, user-defined, object predicates. Built-in predicates and predicates not declared using a scope directive are not enumerated.

This predicate also succeeds for any predicates listed in *uses/2* and *use_module/2* directives.

When `Predicate` is bound at compile-time to a `(:)/2` term, this predicate enumerates module predicates (assuming that the *backend Prolog compiler* supports modules).

Modes and number of proofs

```
current_predicate(?predicate_indicator) - zero_or_more
```

Errors

`Predicate` is neither a variable nor a valid predicate indicator:
`type_error(predicate_indicator, Predicate)`
`Predicate` is a `Name/Arity` term but `Functor` is neither a variable nor an atom:
`type_error(atom, Name)`
`Predicate` is a `Name/Arity` term but `Arity` is neither a variable nor an integer:
`type_error(integer, Arity)`
`Predicate` is a `Name/Arity` term but `Arity` is a negative integer:
`domain_error(not_less_than_zero, Arity)`

Examples

To enumerate, by backtracking, the locally visible user predicates or the user predicates visible in *this*:
`current_predicate(Predicate)`
To enumerate, by backtracking, the public and protected user predicates visible in *self*:
`::current_predicate(Predicate)`
To enumerate, by backtracking, the public user predicates visible for an explicit object:
`Object::current_predicate(Predicate)`

An example of enumerating locally visible object predicates. These include predicates listed using *uses/2* and *use_module/2* directives:

```
:- object(foo).

    :- uses(bar, [
        baz/1, quux/2
    ]).

    :- public(pred/1).
    pred(X) :-
       current_predicate(X).

:- end_object.
```

```
| ?- foo::pred(X).
X = pred/1 ;
X = baz/1 ;
X = quux/2 ;
no
```

See also

current_op/3, *predicate_property/2*, *uses/2*, *use_module/2*

built-in method

`predicate_property/2`

Description

```
predicate_property(Predicate, Property)
```

Enumerates, by backtracking, the properties of a visible object predicate. Properties for predicates not declared using a scope directive are not enumerated. The valid predicate properties are listed in the language grammar section on *predicate properties* and described in the User Manual section on *predicate properties*.

When `Predicate` is listed in a *uses/2* or *use_module/2* directive, properties are enumerated for the referenced object or module predicate.

When `Predicate` is bound at compile-time to a `(:)/2` term, this predicate enumerates properties for module predicates (assuming that the *backend Prolog compiler* supports modules).

Modes and number of proofs

```
predicate_property(+callable, ?predicate_property) - zero_or_more
```

Errors

`Predicate` is a variable:

```
instantiation_error
```

`Predicate` is neither a variable nor a callable term:

```
type_error(callable, Predicate)
```

`Property` is neither a variable nor a valid predicate property:

```
domain_error(predicate_property, Property)
```

Examples

To enumerate, by backtracking, the properties of a locally visible user predicate or a user predicate visible in *this*:

```
predicate_property(Predicate, Property)
```

To enumerate, by backtracking, the properties of a public or protected predicate visible in *self*:

```
::predicate_property(Predicate, Property)
```

To enumerate, by backtracking, the properties of a public predicate visible in an explicit object:

```
Object::predicate_property(Predicate, Property)
```

An example of enumerating properties for locally visible object predicates. These include predicates listed using *uses/2* and *use_module/2* directives:

```
:- object(foo).

    :- uses(bar, [
        baz/1, quux/2
    ]).

    :- public(pred/1).
    pred_prop(Pred, Prop) :-
       predicate_property(Pred, Prop).

:- end_object.
```

```
| ?- foo::pred(baz(_), Prop).
Prop = logtalk ;
Prop = scope(public) ;
Prop = public ;
Prop = declared_in(bar) ;
...
```

See also

current_op/3, *current_predicate/1*, *uses/2*, *use_module/2*

2.5.4 Database

built-in method

abolish/1

Description

```
abolish(Predicate)
```

Abolishes a runtime declared object dynamic predicate or an object local dynamic predicate. Only predicates that are dynamically declared at runtime (using a call to the *asserta/1* or *assertz/1* built-in methods) can be abolished.

When the predicate indicator is declared in a *uses/2* or *use_module/2* directive, the predicate is abolished in the referenced object or module. When the backend Prolog compiler supports a module system, the predicate argument can also be module qualified.

Modes and number of proofs

```
abolish(@predicate_indicator) - one
```

Errors

`Predicate` is a variable:
    `instantiation_error`
`Functor` is a variable:
    `instantiation_error`
`Arity` is a variable:
    `instantiation_error`
`Predicate` is neither a variable nor a valid predicate indicator:
    `type_error(predicate_indicator, Predicate)`
`Functor` is neither a variable nor an atom:
    `type_error(atom, Functor)`
`Arity` is neither a variable nor an integer:
    `type_error(integer, Arity)`
`Predicate` is statically declared:
    `permission_error(modify, predicate_declaration, Name/Arity)`
`Predicate` is a private predicate:
    `permission_error(modify, private_predicate, Name/Arity)`
`Predicate` is a protected predicate:
    `permission_error(modify, protected_predicate, Name/Arity)`
`Predicate` is a static predicate:
    `permission_error(modify, static_predicate, Name/Arity)`
`Predicate` is not declared for the object receiving the message:
    `existence_error(predicate_declaration, Name/Arity)`

Examples

To abolish a local dynamic predicate or a dynamic predicate in *this*:
    `abolish(Predicate)`
To abolish a public or protected dynamic predicate in *self*:
    `::abolish(Predicate)`
To abolish a public dynamic predicate in an explicit object:
    `Object::abolish(Predicate)`

See also

asserta/1, *assertz/1*, *clause/2*, *retract/1*, *retractall/1 dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

built-in method

asserta/1

Description

```
asserta(Head)
asserta((Head:-Body))
```

Asserts a clause as the first one for an object dynamic predicate.

When the predicate was not previously declared (using a scope directive), a dynamic predicate declaration is added to the object. In this case, the predicate scope depends on how this method is called:

asserta(Clause)
: The predicate is dynamically declared as *private*.

::asserta(Clause)
: The predicate is dynamically declared as *protected*.

Object::asserta(Clause)
: The predicate is dynamically declared as *public*.

Note, however, that dynamically declaring a new predicate requires either a local assert or the *dynamic_declarations* compiler flag set to `allow` when the object was created or compiled.

When the predicate indicator for `Head` is declared in a *uses/2* or *use_module/2* directive, the clause is asserted in the referenced object or module. When the backend Prolog compiler supports a module system, the predicate argument can also be module qualified.

This method may be used to assert clauses for predicates that are not declared dynamic for dynamic objects provided that the predicates are declared in *this*. This allows easy initialization of dynamically created objects when writing constructors.

Modes and number of proofs

```
asserta(+clause) - one
```

Errors

`Head` is a variable:
: `instantiation_error`

`Head` is neither a variable nor a callable term:
: `type_error(callable, Head)`

`Body` cannot be converted to a goal:
: `type_error(callable, Body)`

The predicate indicator of `Head`, `Name/Arity`, is that of a private predicate:
: `permission_error(modify, private_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, is that of a protected predicate:
: `permission_error(modify, protected_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, is that of a static predicate:
: `permission_error(modify, static_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, does not match a declared predicate and the target object was created or compiled with support for dynamic declaration of predicates turned off:

```
permission_error(create, predicate_declaration, Name/Arity)
```

Examples

To assert a clause as the first one for a local dynamic predicate or a dynamic predicate in *this*:

```
asterta(Clause)
```

To assert a clause as the first one for any public or protected dynamic predicate in *self*:

```
::asterta(Clause)
```

To assert a clause as the first one for any public dynamic predicate in an explicit object:

```
Object::asterta(Clause)
```

An example of asserting clauses in *this* and in *self* from a category:

```
:- category(attributes,
    implements(attributes_protocol)).

    :- private(attr_/1).
    :- dynamic(attr_/1).

    set_in_this(A, X) :-
       asterta(attr_(A, X)).

    set_in_self(A, X) :-
       ::asterta(attr_(A, X)).

    ...
```

An example of asserting clauses into another object with the predicates listed using a *uses/2* directive (similar when using a *use_module/2* directive):

```
:- object(reasoner(_KnowledgeBase_)).

    :- uses(_KnowledgeBase_, [
        foo/1, bar/1
    ]).

    baz(X) :-
       % compiled as _KnowledgeBase_::assertz(foo(X))
       asterta(foo(X)).

   foobar(Name, Argument) :-
      Clause =.. [Name, Argument],
      % runtime resolved to _KnowledgeBase_::assertz(Clause)
      % when Name is either foo or bar
      asterta(Clause).

    ...
```

See also

abolish/1, *assertz/1*, *clause/2*, *retract/1*, *retractall/1 dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

built-in method

assertz/1

Description

```
assertz(Head)
assertz((Head:-Body))
```

Asserts a clause as the last one for a dynamic predicate.

When the predicate was not previously declared (using a scope directive), a dynamic predicate declaration is added to the object. In this case, the predicate scope depends on how this method is called:

assertz(Clause)
: The predicate is dynamically declared as *private*.

::assertz(Clause)
: The predicate is dynamically declared as *protected*.

Object::assertz(Clause)
: The predicate is dynamically declared as *public*.

Note, however, that dynamically declaring a new predicate requires either a local assert or the *dynamic_declarations* compiler flag set to `allow` when the object was created or compiled.

When the predicate indicator for `Head` is declared in a *uses/2* or *use_module/2* directive, the clause is asserted in the referenced object or module. When the backend Prolog compiler supports a module system, the predicate argument can also be module qualified.

This method may be used to assert clauses for predicates that are not declared dynamic for dynamic objects provided that the predicates are declared in *this*. This allows easy initialization of dynamically created objects when writing constructors.

Modes and number of proofs

```
assertz(+clause) - one
```

Errors

`Head` is a variable:
: `instantiation_error`

`Head` is neither a variable nor a callable term:
: `type_error(callable, Head)`

`Body` cannot be converted to a goal:
: `type_error(callable, Body)`

The predicate indicator of `Head`, `Name/Arity`, is that of a private predicate:
: `permission_error(modify, private_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, is that of a protected predicate:

`permission_error(modify, protected_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, is that of a static predicate:

`permission_error(modify, static_predicate, Name/Arity)`

The predicate indicator of `Head`, `Name/Arity`, does not match a declared predicate and the target object was created/compiled with support for dynamic declaration of predicates turned off:

`permission_error(create, predicate_declaration, Name/Arity)`

Examples

To assert a clause as the last one for a local dynamic predicate or a dynamic predicate in *this*:

`assertz(Clause)`

To assert a clause as the last one for any public or protected dynamic predicate in *self*:

`::assertz(Clause)`

To assert a clause as the last one for any public dynamic predicate in an explicit object:

`Object::assertz(Clause)`

An example of asserting clauses in *this* and in *self* from a category:

```
:- category(attributes,
    implements(attributes_protocol)).

    :- private(attr_/1).
    :- dynamic(attr_/1).

    set_in_this(A, X) :-
       assertz(attr_(A, X)).

    set_in_self(A, X) :-
       ::assertz(attr_(A, X)).

    ...
```

An example of asserting clauses into another object with the predicates listed using a *uses/2* directive (similar when using a *use_module/2* directive):

```
:- object(reasoner(_KnowledgeBase_)).

    :- uses(_KnowledgeBase_, [
        foo/1, bar/1
    ]).

    baz(X) :-
       % compiled as _KnowledgeBase_::assertz(foo(X))
       assertz(foo(X)).

   foobar(Name, Argument) :-
      Clause =.. [Name, Argument],
      % runtime resolved to _KnowledgeBase_::assertz(Clause)
      % when Name is either foo or bar
      assertz(Clause).
```

(continues on next page)

(continued from previous page)

```
    ...
```

See also

abolish/1, *asserta/1*, *clause/2*, *retract/1*, *retractall/1 dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

built-in method

clause/2

Description

```
clause(Head, Body)
```

Enumerates, by backtracking, the clauses of a dynamic predicate.

When the predicate indicator for `Head` is declared in a *uses/2* or *use_module/2* directive, the predicate enumerates the clauses in the referenced object or module. When the backend Prolog compiler supports a module system, the head argument can also be module qualified.

This method may be used to enumerate clauses for predicates that are not declared dynamic for dynamic objects provided that the predicates are declared in *this*.

Modes and number of proofs

```
clause(+callable, ?body) - zero_or_more
```

Errors

`Head` is a variable:
    `instantiation_error`
`Head` is neither a variable nor a callable term:
    `type_error(callable, Head)`
`Body` is neither a variable nor a callable term:
    `type_error(callable, Body)`
The predicate indicator of `Head`, `Name/Arity`, is that of a private predicate:
    `permission_error(access, private_predicate, Name/Arity)`
The predicate indicator of `Head`, `Name/Arity`, is that of a protected predicate:
    `permission_error(access, protected_predicate, Name/Arity)`
The predicate indicator of `Head`, `Name/Arity`, is that of a static predicate:
    `permission_error(access, static_predicate, Name/Arity)`
`Head` is not a declared predicate:
    `existence_error(predicate_declaration, Name/Arity)`

Examples

To retrieve a matching clause of a local dynamic predicate or a dynamic predicate in *this*:

```
clause(Head, Body)
```

To retrieve a matching clause of a public or protected dynamic predicate in *self*:

```
::clause(Head, Body)
```

To retrieve a matching clause of a public dynamic predicate in an explicit object:

```
Object::clause(Head, Body)
```

See also

abolish/1, *asserta/1*, *assertz/1*, *retract/1*, *retractall/1 dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

built-in method

retract/1

Description

```
retract(Head)
retract((Head:-Body))
```

Retracts a clause for an object dynamic predicate. On backtracking, the predicate retracts the next matching clause.

When the predicate indicator for `Head` is declared in a *uses/2* or *use_module/2* directive, the clause is retracted in the referenced object or module. When the backend Prolog compiler supports a module system, the predicate argument can also be module qualified.

This method may be used to retract clauses for predicates that are not declared dynamic for dynamic objects provided that the predicates are declared in *this*.

Modes and number of proofs

```
retract(+clause) - zero_or_more
```

Errors

`Head` is a variable:

```
instantiation_error
```

`Head` is neither a variable nor a callable term:

```
type_error(callable, Head)
```

The predicate indicator of `Head`, `Name/Arity`, is that of a private predicate:

```
permission_error(modify, private_predicate, Name/Arity)
```

The predicate indicator of `Head`, `Name/Arity`, is that of a protected predicate:

```
permission_error(modify, protected_predicate, Name/Arity)
```

The predicate indicator of `Head`, `Name/Arity`, is that of a static predicate:

```
permission_error(modify, static_predicate, Name/Arity)
```

The predicate indicator of `Head`, `Name/Arity`, is not declared:

```
existence_error(predicate_declaration, Name/Arity)
```

Examples

To retract a matching clause of a dynamic predicate in *this*:

```
retract(Clause)
```

To retract a matching clause of a public or protected dynamic predicate in *self*:

```
::retract(Clause)
```

To retract a matching clause of a public dynamic predicate in an explicit object:

```
Object::retract(Clause)
```

See also

abolish/1, *asserta/1*, *assertz/1*, *clause/2*, *retractall/1*, *dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

built-in method

`retractall/1`

Description

```
retractall(Head)
```

Retracts all clauses with a matching head for an object dynamic predicate.

When the predicate indicator for `Head` is declared in a *uses/2* or *use_module/2* directive, the clauses are retracted in the referenced object or module. When the backend Prolog compiler supports a module system, the predicate argument can also be module qualified.

This method may be used to retract clauses for predicates that are not declared dynamic for dynamic objects provided that the predicates are declared in *this*.

Modes and number of proofs

```
retractall(@callable) - one
```

Errors

`Head` is a variable:
`instantiation_error`
`Head` is neither a variable nor a callable term:
`type_error(callable, Head)`
The predicate indicator of `Head`, `Name/Arity`, is that of a private predicate:
`permission_error(modify, private_predicate, Name/Arity)`
The predicate indicator of `Head`, `Name/Arity`, is that of a protected predicate:
`permission_error(modify, protected_predicate, Name/Arity)`
The predicate indicator of `Head`, `Name/Arity`, is that of a static predicate:
`permission_error(modify, static_predicate, Name/Arity)`
The predicate indicator of `Head`, `Name/Arity`, is not declared:
`existence_error(predicate_declaration, Name/Arity)`

Examples

To retract all clauses with a matching head of a dynamic predicate in *this*:
`retractall(Head)`
To retract all clauses with a matching head of a public or protected dynamic predicate in *self*:
`::retractall(Head)`
To retract all clauses with a matching head of a public dynamic predicate in an explicit object:
`Object::retractall(Head)`

See also

abolish/1, *asserta/1*, *assertz/1*, *clause/2*, *retract/1*, *dynamic/0*, *dynamic/1*, *uses/2*, *use_module/2*

2.5.5 Meta-calls

built-in method

call/1-N

Description

```
call(Goal)
call(Closure, Arg1, ...)
```

Calls a goal constructed by appending additional arguments to a *closure*. The upper limit for `N` depends on the upper limit for the arity of a compound term of the *backend Prolog compiler*. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object. The `Closure` argument can also be a lambda expression or a Logtalk control construct. When using a backend Prolog compiler supporting a module system, calls in the format `call(Module:Closure, Arg1, ...)` may also be used.

This meta-predicate is opaque to cuts in its arguments.

Meta-predicate template

```
call(0)
call(1, *)
call(2, *, *)
...
```

Modes and number of proofs

```
call(+callable) - zero_or_more
call(+callable, ?term) - zero_or_more
call(+callable, ?term, ?term) - zero_or_more
...
```

Errors

`Goal` is a variable:

```
instantiation_error
```

`Goal` is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

`Closure` is a variable:

```
instantiation_error
```

`Closure` is neither a variable nor a callable term:

```
type_error(callable, Closure)
```

Examples

Call a goal, constructed by appending additional arguments to a closure, in the context of the object or category containing the call:

```
call(Closure, Arg1, Arg2, ...)
```

To send a goal, constructed by appending additional arguments to a closure, as a message to *self*:

```
call(::Closure, Arg1, Arg2, ...)
```

To send a goal, constructed by appending additional arguments to a closure, as a message to an explicit object:

```
call(Object::Closure, Arg1, Arg2, ...)
```

See also

ignore/1, *once/1*, *(\+)/1*

built-in method

`ignore/1`

Description

```
ignore(Goal)
```

This predicate succeeds whether its argument succeeds or fails and it is not re-executable. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

This meta-predicate is opaque to cuts in its argument.

Meta-predicate template

```
ignore(0)
```

Modes and number of proofs

```
ignore(+callable) - one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`

Examples

Call a goal and succeeding even if it fails:
    `ignore(Goal)`
To send a message succeeding even if it fails to *self*:
    `ignore(::Goal)`
To send a message succeeding even if it fails to an explicit object:
    `ignore(Object::Goal)`

See also

call/1-N, *once/1*, *(\+)/1*

built-in method

once/1

Description

```
once(Goal)
```

This predicate behaves as `call(Goal)` but it is not re-executable. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

This meta-predicate is opaque to cuts in its argument.

Meta-predicate template

```
once(0)
```

Modes and number of proofs

```
once(+callable) - zero_or_one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`

Examples

Call a goal deterministically in the context of the object or category containing the call:
    `once(Goal)`
To send a goal as a non-backtracable message to *self*:
    `once(::Goal)`
To send a goal as a non-backtracable message to an explicit object:
    `once(Object::Goal)`

See also

call/1-N, *ignore/1*, *(\+)/1*

built-in method

(\+)/1

Description

```
\+ Goal
```

Not-provable meta-predicate. True iff `call(Goal)` is false. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Warning

The argument is always compiled (for improved performance). As a consequence, when the argument is a control construct (e.g., a conjunction), any meta-variables will be wrapped with the equivalent to the `call/1` control construct. Note that these semantics differ from the ISO Prolog Core standard specification for the `(\+)/1` built-in predicate. For example, assuming a conforming system:

```
| ?- X = !, \+ (member(Y,[1,2,3]), X, write(Y), fail).
1

X = !
```

But in Logtalk `X` is compiled into a meta-call, which is not cut-transparent:

```
yes
| ?- logtalk << (X = !, \+ (member(Y,[1,2,3]), X, write(Y), fail)).
123

X = !
```

Note that the ISO Prolog Core standard doesn't specify a cut-transparent alternative to the `call/1` control construct.

Meta-predicate template

```
\+ 0
```

Modes and number of proofs

```
\+ +callable - zero_or_one
```

Errors

`Goal` is a variable:
`instantiation_error`
`Goal` is neither a variable nor a callable term:
`type_error(callable, Goal)`

Examples

Not-provable goal in the context of the object or category containing the call:
`\+ Goal`
Not-provable goal sent as a message to *self*:
`\+ ::Goal`
Not-provable goal sent as a message to an explicit object:
`\+ Object::Goal`

See also

call/1-N, *ignore/1*, *once/1*

2.5.6 Error handling

built-in method

catch/3

Description

```
catch(Goal, Catcher, Recovery)
```

Catches exceptions thrown by a goal. See also the ISO Prolog standard definition. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

This method can also be used in grammar rules. In this case, the meta-arguments are interpreted as grammar rule bodies.

Modes and number of proofs

```
catch(?callable, ?term, ?callable) - zero_or_more
```

Errors

(none)

Examples

(none)

See also

throw/1, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *evaluation_error/1*, *representation_error/1* *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

throw/1

Description

```
throw(Exception)
```

Throws an exception. See also the ISO Prolog standard definition. This built-in method is declared private and thus cannot be used as a message to an object.

Modes and number of proofs

```
throw(+nonvar) - error
```

Errors

`Exception` is a variable:

```
instantiation_error
```

`Exception` does not unify with the second argument of any call of *catch/3*:

```
system_error
```

Examples

(none)

See also

catch/3, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *evaluation_error/1*, *representation_error/1* *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

`instantiation_error/0`

Description

```
instantiation_error
```

Throws an instantiation error. Used when an argument or one of its sub-arguments is a variable but a non-variable term is required. For example, trying to open a file with a variable for the input/output mode.

This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of calls:

```
...,
context(Context),
throw(error(instantiation_error, Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Modes and number of proofs

```
instantiation_error - error
```

Errors

When called:

    instantiation_error

Examples

```
...,
var(Handler),
instantiation_error.
```

See also

catch/3, *throw/1*, *context/1*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

uninstantiation_error/1

Description

```
uninstantiation_error(Culprit)
```

Throws an uninstantiation error. Used when an argument or one of its sub-arguments is bound but a variable is required. For example, trying to open a file with a stream argument bound.

This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of calls:

```
...,
context(Context),
throw(error(uninstantiation_error(Culprit), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Modes and number of proofs

```
uninstantiation_error(@nonvar) - error
```

Errors

When called:

```
uninstantiation_error(Culprit)
```

Examples

```
...,
var(Handler),
uninstantiation_error(my_stream).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

type_error/2

Description

```
type_error(Type, Culprit)
```

Throws a type error. Used when the type of an argument is incorrect. For example, trying to use a non-callable term as a message. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(type_error(Type,Culprit), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Type` include all the types defined by the `type` library object and by other libraries such as `os`, `expecteds`, and `optionals`. The value of `Culprit` is the argument or one of its sub-terms that caused the error.

Modes and number of proofs

```
type_error(@nonvar, @term) - error
```

Errors

When called:

```
type_error(Type, Culprit)
```

Examples

```
...,
\+ atom(Name),
type_error(atom, Name).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1* *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*,

built-in method

domain_error/2

Description

```
domain_error(Domain, Culprit)
```

Throws a domain error. Used when an argument is of the correct type but outside the valid domain. For example, trying to use an atom as an operator specifier that is not a valid specifier. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(domain_error(Domain,Culprit), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Domain` include:

- `character_code_list`
- `close_option`
- `flag_option`
- `flag_value`
- `compiler_flag`
- `flag`
- `prolog_flag`
- `io_mode`
- `non_empty_list`
- `not_less_than_zero`
- `operator_priority`
- `operator_specifier`
- `read_option`

- source_sink
- stream
- stream_option
- stream_or_alias
- stream_position
- stream_property
- write_option
- character_code_list
- text_encoding
- directive
- object_directive
- protocol_directive
- category_directive
- object_relation
- protocol_relation
- category_relation
- object_property
- protocol_property
- category_property
- predicate_property
- meta_argument_specifier
- meta_directive_template
- closure
- allocation
- redefinition
- message_sending_goal
- class
- prototype
- scope
- boolean

The value of `Culprit` is the argument or one of its sub-terms that caused the error.

Modes and number of proofs

```
domain_error(+atom, @nonvar) - error
```

Errors

When called:

```
domain_error(Domain, Culprit)
```

Examples

```
...,
atom(Color),
\+ color(Color),
domain_error(color, Color).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

consistency_error/3

Description

```
consistency_error(Consistency, Argument1, Argument2)
```

Throws a consistency error. Used when two directives or predicate arguments are individually correct but together are not consistent. For example, a predicate and its alias having different arity in a `uses/2` directive. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(consistency_error(Consistency,Argument1,Argument2), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values representing `Consistency` checks include:

- `same_arity`
- `same_number_of_parameters`
- `same_number_of_arguments`
- `same_closure_specification`

Modes and number of proofs

```
consistency_error(+atom, @nonvar, @nonvar) - error
```

Errors

When called:

```
consistency_error(Consistency, Argument1, Argument2)
```

Examples

```
% code that will trigger consistency errors when compiled:

% predicates (and non-terminals) aliases must have the same
% arity as the original predicates (and non-terminals)
:- uses(list, [
    member/2 as in/1
]).

% meta-predicate templates should be consistent with how closures
% are used regarding the number of additional arguments
:- public(p/2).
:- meta_predicate(p(1, *)).

p(G, A) :-
    call(G, A, 2).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

existence_error/2

Description

```
existence_error(Thing, Culprit)
```

Throws an existence error. Used when the subject of an operation does not exist. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(existence_error(Thing,Culprit), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Thing` include:

- `predicate`
- `non_terminal`
- `predicate_declaration`
- `procedure`
- `source_sink`
- `stream`
- `object`
- `protocol`
- `category`
- `module`
- `ancestor`
- `library`
- `file`
- `directive`
- `engine`
- `thread`
- `goal_thread`

The value of `Culprit` is the argument or one of its sub-terms that caused the error.

Modes and number of proofs

```
existence_error(@nonvar, @nonvar) - error
```

Errors

When called:

```
existence_error(Thing, Culprit)
```

Examples

```
...,
\+ current_object(payroll),
existence_error(object, payroll).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *evaluation_error/1*, *permission_error/3*, *representation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

permission_error/3

Description

```
permission_error(Operation, PermissionType, Culprit)
```

Throws a permission error. Used when an operation is not allowed. For example, sending a message for a predicate that is not within the scope of the sender. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(permission_error(Operation,PermissionType,Culprit), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Operation` include:

- `access`
- `create`
- `declare`
- `define`
- `modify`
- `open`
- `input`
- `output`
- `reposition`
- `repeat`

Possible values for `PermissionType` include:

- `predicate_declaration`
- `protected_predicate`

- private_predicate
- static_predicate
- dynamic_predicate
- predicate
- non_terminal
- database
- object
- static_object
- static_protocol
- static_category
- entity_relation
- operator
- flag
- engine
- binary_stream
- text_stream
- source_sink
- stream
- past_end_of_stream

The value of `Culprit` is the argument or one of its sub-terms that caused the error.

Modes and number of proofs

```
permission_error(@nonvar, @nonvar, @nonvar) - error
```

Errors

When called:

```
permission_error(Operation, PermissionType, Culprit)
```

Examples

```
...,
\+ writable(File),
permission_error(modify, file, File).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *representation_error/1*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

`representation_error/1`

Description

```
representation_error(Flag)
```

Throws a representation error. Used when some representation limit is exceeded. For example, trying to construct a compound term that exceeds the maximum arity supported by the backend Prolog system. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
..., 
context(Context),
throw(error(representation_error(Flag), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Flag` include:

- `character`
- `character_code`
- `in_character_code`
- `max_arity`
- `max_integer`
- `min_integer`
- `acyclic_term`
- `lambda_parameters`
- `entity_prefix`

Modes and number of proofs

```
representation_error(+atom) - error
```

Errors

When called:

```
representation_error(Flag)
```

Examples

```
...,
Code > 127,
representation_error(character_code).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *evaluation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

evaluation_error/1

Description

```
evaluation_error(Error)
```

Throws an evaluation error. Used when evaluating an arithmetic expression generates an exception. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(evaluation_error(Error), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Error` include:

- `float_overflow`
- `int_overflow`
- `undefined`
- `underflow`
- `zero_divisor`

Modes and number of proofs

```
evaluation_error(@nonvar) - error
```

Errors

When called:

```
evaluation_error(Exception)
```

Examples

```
...,
Divisor =:= 0,
evaluation_error(zero_divisor).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *resource_error/1*, *syntax_error/1*, *system_error/0*

built-in method

resource_error/1

Description

```
resource_error(Resource)
```

Throws a resource error. Used when a required resource (e.g., memory or disk space) to complete execution is not available. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(resource_error(Resource), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Possible values for `Resource` include:

- `engines`
- `threads`
- `coinduction`
- `soft_cut_support`
- `text_encoding_support`

Modes and number of proofs

```
resource_error(@nonvar) - error
```

Errors

When called:

```
resource_error(Resource)
```

Examples

```
...,
empty(Tank),
resource_error(gas).
```

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *syntax_error/1*, *system_error/0*

built-in method

syntax_error/1

Description

```
syntax_error(Description)
```

Throws a syntax error. Used when the sequence of characters being read are not syntactically valid. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(syntax_error(Description), Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Modes and number of proofs

```
syntax_error(@nonvar) - error
```

Errors

When called:

```
syntax_error(Description)
```

Examples

(none)

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1*, *system_error/0 resource_error/1*

built-in method

system_error/0

Description

```
system_error
```

Throws a system error. Used when runtime execution can no longer proceed. For example, an exception is thrown without an active catcher. This built-in method is declared private and thus cannot be used as a message to an object. Calling this predicate is equivalent to the following sequence of goals:

```
...,
context(Context),
throw(error(system_error, Context)).
```

This allows the user to generate errors in the same format used by the runtime.

Modes and number of proofs

```
system_error - error
```

Errors

When called:

```
system_error
```

Examples

(none)

See also

catch/3, *throw/1*, *context/1*, *instantiation_error/0*, *uninstantiation_error/1*, *type_error/2*, *domain_error/2*, *consistency_error/3*, *existence_error/2*, *permission_error/3*, *representation_error/1 evaluation_error/1*, *resource_error/1*, *syntax_error/1*,

2.5.7 All solutions

built-in method

bagof/3

Description

```
bagof(Template, Goal, List)
```

Collects a bag of solutions for the goal for each set of instantiations of the free variables in the goal. The order of the elements in the bag follows the order of the goal solutions. The free variables in the goal are the variables that occur in the goal but not in the template. Free variables can be ignored, however, by using the `^/2` existential qualifier. For example, if `T` is term containing all the free variables that we want to ignore, we can write `T^Goal`. Note that the term `T` can be written as `V1^V2^...`.

When there are free variables, this method is re-executable on backtracking. This method fails when there are no solutions, never returning an empty list.

This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Meta-predicate template

```
bagof(*, ^, *)
```

Modes and number of proofs

```
bagof(@term, +callable, -list) - zero_or_more
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
`Goal` is a call to a non-existing predicate:
    `existence_error(procedure, Predicate)`

Examples

To find a bag of solutions in the context of the object or category containing the call:
    `bagof(Template, Goal, List)`
To find a bag of solutions of sending a message to *self*:
    `bagof(Template, ::Message, List)`
To find a bag of solutions of sending a message to an explicit object:
    `bagof(Template, Object::Message, List)`

See also

findall/3, *findall/4*, *forall/2*, *setof/3*

built-in method

findall/3

Description

```
findall(Template, Goal, List)
```

Collects a list of solutions for the goal. The order of the elements in the list follows the order of the goal solutions. It succeeds returning an empty list when the goal has no solutions.

This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Meta-predicate template

```
findall(*, 0, *)
```

Modes and number of proofs

```
findall(?term, +callable, ?list) - zero_or_one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
`Goal` is a call to a non-existing predicate:
    `existence_error(procedure, Predicate)`

Examples

To find all solutions in the context of the object or category containing the call:

```
findall(Template, Goal, List)
```

To find all solutions of sending a message to *self*:

```
findall(Template, ::Message, List)
```

To find all solutions of sending a message to an explicit object:

```
findall(Template, Object::Message, List)
```

See also

bagof/3, *findall/4*, *forall/2*, *setof/3*

built-in method

`findall/4`

Description

```
findall(Template, Goal, List, Tail)
```

Variant of the *findall/3* method that allows passing the tail of the results list. It succeeds returning the tail argument when the goal has no solutions.

This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Meta-predicate template

```
findall(*, 0, *, *)
```

Modes and number of proofs

```
findall(?term, +callable, ?list, ?term) - zero_or_one
```

Errors

`Goal` is a variable:
    `instantiation_error`
`Goal` is neither a variable nor a callable term:
    `type_error(callable, Goal)`
`Goal` is a call to a non-existing predicate:
    `existence_error(procedure, Predicate)`

Examples

To find all solutions in the context of the object or category containing the call:
    `findall(Template, Goal, List, Tail)`
To find all solutions of sending a message to *self*:
    `findall(Template, ::Message, List, Tail)`
To find all solutions of sending a message to an explicit object:
    `findall(Template, Object::Message, List, Tail)`

See also

bagof/3, *findall/3*, *forall/2*, *setof/3*

built-in method

forall/2

Description

```
forall(Generator, Test)
```

For all solutions of `Generator`, `Test` is true. This meta-predicate implements a *generate-and-test* loop using a definition equivalent to `\+ (Generator, \+ Test)`. As a consequence, no variables in the arguments are bound by a call to this predicate. This predicate often provides a better alternative to a *failure-driven loop* as an unexpected `Test` failure will not be ignored as it will make the `forall/2` call fail.

This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Meta-predicate template

```
forall(0, 0)
```

Modes and number of proofs

```
forall(@callable, @callable) - zero_or_one
```

Errors

Either `Generator` or `Test` is a variable:
`instantiation_error`
`Generator` is neither a variable nor a callable term:
`type_error(callable, Generator)`
`Test` is neither a variable nor a callable term:
`type_error(callable, Test)`

Examples

To call both goals in the context of the object or category containing the call:
`forall(Generator, Test)`
To send both goals as messages to *self*:
`forall(::Generator, ::Test)`
To send both goals as messages to explicit objects:
`forall(Object1::Generator, Object2::Test)`

See also

bagof/3, *findall/3*, *findall/4*, *setof/3*

built-in method

setof/3

Description

```
setof(Template, Goal, List)
```

Collects a set of solutions for the goal for each set of instantiations of the free variables in the goal. The solutions are sorted using standard term order. The free variables in the goal are the variables that occur in the goal but not in the template. Free variables can be ignored, however, by using the `^/2` existential qualifier. For example, if `T` is term containing all the free variables that we want to ignore, we can write `T^Goal`. Note that the term `T` can be written as `V1^V2^...`.

When there are free variables, this method is re-executable on backtracking. This method fails when there are no solutions, never returning an empty list.

This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Meta-predicate template

```
setof(*, ^, *)
```

Modes and number of proofs

```
setof(@term, +callable, -list) - zero_or_more
```

Errors

`Goal` is a variable:
`instantiation_error`
`Goal` is neither a variable nor a callable term:
`type_error(callable, Goal)`
`Goal` is a call to a non-existing predicate:
`existence_error(procedure, Predicate)`

Examples

To find a set of solutions in the context of the object or category containing the call:
`setof(Template, Goal, List)`
To find a set of solutions of sending a message to *self*:
`setof(Template, ::Message, List)`
To find a set of solutions of sending a message to an explicit object:
`setof(Template, Object::Message, List)`

See also

bagof/3, *findall/3*, *findall/4*, *forall/2*

2.5.8 Event handling

built-in method

`before/3`

Description

```
before(Object, Message, Sender)
```

User-defined method for handling *before events*. This method is declared in the monitoring built-in protocol as a public predicate and is automatically called by the runtime for messages sent using the *(::)/2* control construct from within objects compiled with the *events* flag set to `allow`.

Note that you can make this predicate scope protected or private by using, respectively, *protected or private implementation* of the `monitoring` protocol.

Modes and number of proofs

```
before(?object_identifier, ?callable, ?object_identifier) - zero_or_more
```

Errors

(none)

Examples

```
:- object(...,
    implements(monitoring),
    ...).

    % write a log message when a message is sent:
    before(Object, Message, Sender) :-
        writeq(Object), write('::'), writeq(Message),
        write(' from '), writeq(Sender), nl.
```

See also

after/3, *abolish_events/5*, *current_event/5*, *define_events/5*

built-in method

after/3

Description

```
after(Object, Message, Sender)
```

User-defined method for handling *after events*. This method is declared in the monitoring built-in protocol as a public predicate and is automatically called by the runtime for messages sent using the *(::)/2* control construct from within objects compiled with the *events* flag set to `allow`.

Note that you can make this predicate scope protected or private by using, respectively, *protected or private implementation* of the `monitoring` protocol.

Modes and number of proofs

```
after(?object_identifier, ?callable, ?object_identifier) - zero_or_more
```

Errors

(none)

Examples

```
:- object(...,
    implements(monitoring),
    ...).

    % write a log message when a message is successful:
    after(Object, Message, Sender) :-
        writeq(Object), write('::'), writeq(Message),
        write(' from '), writeq(Sender), nl.
```

See also

before/3, *abolish_events/5*, *current_event/5*, *define_events/5*

2.5.9 Message forwarding

built-in method

forward/1

Description

```
forward(Message)
```

User-defined method for forwarding unknown messages sent to an object (using the *(::)/2* control construct), automatically called by the runtime when defined. This method is declared in the forwarding built-in protocol as a *public* predicate. Note that you can make its scope protected or private by using, respectively, *protected or private implementation* of the `forwarding` protocol.

Modes and number of proofs

```
forward(+callable) - zero_or_more
```

Errors

(none)

Examples

```
:- object(proxy,
    implements(forwarding),
    ...).

    forward(Message) :-
        % delegate unknown messages to the "real" object
        [real::Message].
```

See also

[]/1

2.5.10 Definite clause grammar rules

built-in method

call//1-N

Description

```
call(Closure)
call(Closure, Arg1, ...)
call(Object::Closure, Arg1, ...)
call(::Closure, Arg1, ...)
call(^^Closure, Arg1, ...)
...
```

This non-terminal takes a *closure* and is processed by appending the two implicit grammar rule arguments to the arguments of the closure. This built-in non-terminal is interpreted as a private non-terminal and thus cannot be used as a message to an object.

Using this non-terminal is recommended when calling a predicate whose last two arguments are the two implicit grammar rule arguments to avoid hard-coding assumptions about how grammar rules are compiled into clauses. Note that the compiler ensures zero overhead when using this non-terminal with a bound argument at compile-time. To call a predicate with a different argument order, use a *lambda expression* or define a *predicate alias*. For example:

```
square -->
    call([Number, Double]>>(Double is Number*Number)).
```

When using a *backend Prolog compiler* supporting a module system, calls in the format `call(Module:Closure)` may also be used.

Meta-non-terminal template

```
call(0)
call(1, *)
call(2, *, *)
...
```

Modes and number of proofs

```
call(+callable) - zero_or_more
call(+callable, ?term) - zero_or_more
call(+callable, ?term, ?term) - zero_or_more
...
```

Errors

`Closure` is a variable:

```
instantiation_error
```

`Closure` is neither a variable nor a callable term:

```
type_error(callable, Closure)
```

Examples

Calls a goal, constructed by appending the two implicit grammar rule arguments to the closure, in in the context of the object or category containing the call:

```
call(Closure)
```

To make a *super* call, constructed by appending the two implicit grammar rule arguments to the closure:

```
call(^^Closure)
```

To send a goal, constructed by appending the two implicit grammar rule arguments to the closure, as a message to *self*:

```
call(::Closure)
```

To send a goal, constructed by appending the two implicit grammar rule arguments to the closure, as a message to an explicit object:

```
call(Object::Closure)
```

See also

eos//0, *phrase//1*, *phrase/2*, *phrase/3*

built-in method

eos//0

Description

```
eos
```

This non-terminal matches the end-of-input. It is implemented by checking that the implicit difference list unifies with `[]-[]`.

Modes and number of proofs

```
eos - zero_or_one
```

Errors

(none)

Examples

```
abc --> a, b, c, eos.
```

See also

call//1-N, *phrase//1*, *phrase/2*, *phrase/3*

built-in method

phrase//1

Description

```
phrase(GrammarRuleBody)
```

This non-terminal takes a grammar rule body and parses it using the two implicit grammar rule arguments. A common use is to wrap what otherwise would be a *naked meta-variable* in a grammar rule body when defining a meta non-terminal.

Meta-non-terminal template

```
phrase(0)
```

Modes and number of proofs

```
phrase(+callable) - zero_or_more
```

Errors

`GrammarRuleBody` is a variable:

```
instantiation_error
```

`GrammarRuleBody` is neither a variable nor a callable term:

```
type_error(callable, GrammarRuleBody)
```

Examples

(none)

See also

call//1-N, *phrase/2*, *phrase/3*

built-in method

phrase/2

Description

```
phrase(GrammarRuleBody, Input)
phrase(::GrammarRuleBody, Input)
phrase(Object::GrammarRuleBody, Input)
```

True when the `GrammarRuleBody` grammar rule body can be applied to the `Input` list of tokens. In the most common case, `GrammarRuleBody` is a non-terminal defined by a grammar rule. This built-in method is declared private and thus cannot be used as a message to an object. When using a *backend Prolog compiler* supporting a module system, calls in the format `phrase(Module:GrammarRuleBody, Input)` may also be used.

This method is opaque to cuts in the first argument. When the first argument is sufficiently instantiated at compile-time, the method call is compiled in order to eliminate the implicit overheads of converting the grammar rule body into a goal and meta-calling it. For performance reasons, the second argument is only type-checked at compile-time.

Meta-predicate template

```
phrase(2, *)
```

Modes and number of proofs

```
phrase(+callable, ?list) - zero_or_more
```

Errors

`GrammarRuleBody` is a variable:
    `instantiation_error`
`GrammarRuleBody` is neither a variable nor a callable term:
    `type_error(callable, GrammarRuleBody)`

Examples

To parse a list of tokens using a local non-terminal:
    `phrase(NonTerminal, Input)`
To parse a list of tokens using a non-terminal within the scope of *self*:
    `phrase(::NonTerminal, Input)`
To parse a list of tokens using a public non-terminal of an explicit object:
    `phrase(Object::NonTerminal, Input)`

See also

call//1-N, *phrase//1*, *phrase/3*

built-in method

phrase/3

Description

```
phrase(GrammarRuleBody, Input, Rest)
phrase(::GrammarRuleBody, Input, Rest)
phrase(Object::GrammarRuleBody, Input, Rest)
```

True when the `GrammarRuleBody` grammar rule body can be applied to the `Input-Rest` difference list of tokens. In the most common case, `GrammarRuleBody` is a non-terminal defined by a grammar rule. This built-in method is declared private and thus cannot be used as a message to an object. When using a *backend Prolog compiler* supporting a module system, calls in the format `phrase(Module:GrammarRuleBody, Input, Rest)` may also be used.

This method is opaque to cuts in the first argument. When the first argument is sufficiently instantiated at compile-time, the method call is compiled in order to eliminate the implicit overheads of converting the grammar rule body into a goal and meta-calling it. For performance reasons, the second and third arguments are only type-checked at compile time.

Meta-predicate template

```
phrase(2, *, *)
```

Modes and number of proofs

```
phrase(+callable, ?list, ?list) - zero_or_more
```

Errors

`GrammarRuleBody` is a variable:
`instantiation_error`
`GrammarRuleBody` is neither a variable nor a callable term:
`type_error(callable, GrammarRuleBody)`

Examples

To parse a list of tokens using a local non-terminal:
`phrase(NonTerminal, Input, Rest)`
To parse a list of tokens using a non-terminal within the scope of *self*:
`phrase(::NonTerminal, Input, Rest)`
To parse a list of tokens using a public non-terminal of an explicit object:
`phrase(Object::NonTerminal, Input, Rest)`

See also

call//1-N, *phrase/2*, *phrase/3*

2.5.11 Term and goal expansion

built-in method

expand_term/2

Description

```
expand_term(Term, Expansion)
```

Expands a term. The most common use is to expand a grammar rule into a clause. Users may override the default Logtalk grammar rule translator by defining clauses for the *term_expansion/2* hook predicate.

The expansion works as follows: if the first argument is a variable, then it is unified with the second argument; if the first argument is not a variable and there are local or inherited clauses for the `term_expansion/2`

hook predicate within scope, then this predicate is called to provide an expansion that is then unified with the second argument; if the `term_expansion/2` predicate is not used and the first argument is a compound term with functor `(-->)/2` then the default Logtalk grammar rule translator is used, with the resulting clause being unified with the second argument; when the translator is not used, the two arguments are unified. The `expand_term/2` predicate may return a single term or a list of terms.

This built-in method may be used to expand a grammar rule into a clause for use with the built-in database methods.

Automatic term expansion is only performed at compile-time (to expand terms read from a source file) when using a *hook object*. This predicate can be used by the user to manually perform term expansion at runtime (for example, to convert a grammar rule into a clause).

Modes and number of proofs

```
expand_term(?term, ?term) - one
```

Errors

(none)

Examples

(none)

See also

expand_goal/2, *goal_expansion/2*, *term_expansion/2*

built-in method

`term_expansion/2`

Description

```
term_expansion(Term, Expansion)
```

Defines an expansion for a term. This predicate, when defined and within scope, is automatically called by the *expand_term/2* method. When that is not the case, the `expand_term/2` method only uses the default expansions. Use of this predicate by the `expand_term/2` method may be restricted by changing its default public scope.

The `term_expansion/2` predicate may return a list of terms. Returning an empty list effectively suppresses the term.

Term expansion may also be applied when compiling source files by defining the object providing access to the `term_expansion/2` clauses as a *hook object*. Clauses for the `term_expansion/2` predicate defined within an object or a category are **never** used in the compilation of the object or the category itself. Moreover, in this context, terms wrapped using the *{}/1* compiler bypass control construct are not expanded and any expanded term wrapped in this control construct will not be further expanded.

Objects and categories implementing this predicate should declare that they implement the expanding protocol if no ancestor already declares it. This protocol implementation relation can be declared as either *protected or private* to restrict the scope of this predicate.

Modes and number of proofs

```
term_expansion(+nonvar, -nonvar) - zero_or_one
term_expansion(+nonvar, -list(nonvar)) - zero_or_one
```

Errors

(none)

Examples

```
term_expansion((:- license(default)), (:- license(gplv3))).
term_expansion(data(Millimeters), data(Meters)) :- Meters is Millimeters / 1000.
```

See also

expand_goal/2, *expand_term/2*, *goal_expansion/2*, *logtalk_load_context/2*

built-in method

expand_goal/2

Description

```
expand_goal(Goal, ExpandedGoal)
```

Expands a goal. The expansion works as follows: if the first argument is a variable, then it is unified with the second argument; if the first argument is not a variable and there are local or inherited clauses for the `goal_expansion/2` hook predicate within scope, then this predicate is recursively called until a fixed-point is reached to provide an expansion that is then unified with the second argument; if the `goal_expansion/2` predicate is not within scope, the two arguments are unified.

Automatic goal expansion is only performed at compile-time (to expand the body of clauses and meta-directives read from a source file) when using *hook objects*. This predicate can be used by the user to manually perform goal expansion at runtime (for example, before asserting a clause).

Modes and number of proofs

```
expand_goal(?term, ?term) - one
```

Errors

(none)

Examples

(none)

See also

expand_term/2, *goal_expansion/2*, *term_expansion/2*

built-in method

goal_expansion/2

Description

```
goal_expansion(Goal, ExpandedGoal)
```

Defines an expansion for a goal. The first argument is the goal to be expanded. The expanded goal is returned in the second argument. This predicate is called recursively on the expanded goal until a fixed point is reached. Thus, care must be taken to avoid compilation loops. This predicate, when defined and within scope, is automatically called by the *expand_goal/2* method. Use of this predicate by the `expand_goal/2` method may be restricted by changing its default public scope.

Goal expansion may also be applied when compiling source files by defining the object providing access to the `goal_expansion/2` clauses as a *hook object*. Clauses for the `goal_expansion/2` predicate defined within an object or a category are **never** used in the compilation of the object or the category itself. Moreover, in this context, goals wrapped using the *{}/1* compiler bypass control construct are not expanded and any expanded goal wrapped in this control construct will not be further expanded.

Objects and categories implementing this predicate should declare that they implement the expanding built-in protocol if no ancestor already declares it. This protocol implementation relation can be declared as either *protected or private* to restrict the scope of this predicate.

Modes and number of proofs

```
goal_expansion(+callable, -callable) - zero_or_one
```

Errors

(none)

Examples

```
goal_expansion(write(Term), (write_term(Term, []), nl)).
goal_expansion(read(Term), (write('Input: '), {read(Term)})).
```

See also

expand_goal/2, *expand_term/2*, *term_expansion/2*, *logtalk_load_context/2*

2.5.12 Coinduction hooks

built-in method

coinductive_success_hook/1-2

Description

```
coinductive_success_hook(Head, Hypothesis)
coinductive_success_hook(Head)
```

User-defined hook predicates that are automatically called in case of coinductive success when proving a query for a coinductive predicates. The hook predicates are called with the head of the coinductive predicate on coinductive success and, optionally, with the hypothesis that has used to reach coinductive success.

When both hook predicates are defined, the `coinductive_success_hook/1` clauses are only used if no `coinductive_success_hook/2` clause applies. The compiler ensures zero performance penalties when defining coinductive predicates without a corresponding definition for the coinductive success hook predicates.

The compiler assumes that these hook predicates are defined as static predicates in order to optimize their use.

Modes and number of proofs

```
coinductive_success_hook(+callable, +callable) - zero_or_one
coinductive_success_hook(+callable) - zero_or_one
```

Errors

(none)

Examples

```
% Are there "occurrences" of arg1 in arg2?
:- public(member/2).
:- coinductive(member/2).

member(X, [X| _]).
member(X, [_| T]) :-
    member(X, T).

% Are there infinitely many "occurrences" of arg1 in arg2?
:- public(comember/2).
:- coinductive(comember/2).
comember(X, [_| T]) :-
    comember(X, T).

coinductive_success_hook(member(_, _)) :-
    fail.
coinductive_success_hook(comember(X, L)) :-
    member(X, L).
```

See also

coinductive/1

2.5.13 Message printing

built-in method

`print_message/3`

Description

```
print_message(Kind, Component, Term)
```

Built-in method for printing a message represented by a term, which is converted to the message text using the *logtalk::message_tokens(Term, Component)* hook non-terminal. This method is declared in the logtalk built-in object as a public predicate. The line prefix and the output stream used for each `Kind-Component` pair can be found using the *logtalk::message_prefix_stream(Kind, Component, Prefix, Stream)* hook predicate.

This predicate starts by converting the message term to a list of tokens and by calling the *logtalk::message_hook(Message, Kind, Component, Tokens)* hook predicate. If this predicate succeeds, the `print_message/3` predicate assumes that the message has been successfully printed.

By default: messages of kind `debug` or `debug(_)` are only printed when the `debug` flag is turned on; messages of kind `banner`, `comment`, or `comment(_)` are only printed when the `report` flag is set to `on`; messages of kind `warning` and `warning(_)` are not printed when the `report` flag is set to `off`; messages of kind `silent` and `silent()` are not printed (but can be intercepted).

Modes and number of proofs

```
print_message(+nonvar, +nonvar, +nonvar) - one
```

Errors

(none)

Examples

```
..., logtalk::print_message(information, core, redefining_entity(object, foo)), ...
```

See also

message_hook/4, *message_prefix_stream/4*, *message_tokens//2*, *print_message_tokens/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

message_tokens//2

Description

```
message_tokens(Message, Component)
```

User-defined non-terminal hook used to rewrite a message term into a list of tokens and declared in the logtalk built-in object as a public, multifile, and dynamic non-terminal. The list of tokens can be printed by calling the *print_message_tokens/3* method. This non-terminal hook is automatically called by the *print_message/3* method.

Modes and number of proofs

```
message_tokens(+nonvar, +nonvar) - zero_or_more
```

Errors

(none)

Examples

```
:- multifile(logtalk::message_tokens//2).
:- dynamic(logtalk::message_tokens//2).

logtalk::message_tokens(redefining_entity(Type, Entity), core) -->
    ['Redefining ~w ~q'-[Type, Entity], nl].
```

See also

message_hook/4, *message_prefix_stream/4*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

message_hook/4

Description

```
message_hook(Message, Kind, Component, Tokens)
```

User-defined hook method for intercepting printing of a message, declared in the logtalk built-in object as a public, multifile, and dynamic predicate. This hook method is automatically called by the *print_message/3* method. When the call succeeds, the `print_message/3` method assumes that the message has been successfully printed.

Modes and number of proofs

```
message_hook(@nonvar, @nonvar, @nonvar, @list(nonvar)) - zero_or_one
```

Errors

(none)

Examples

```
:- multifile(logtalk::message_hook/4).
:- dynamic(logtalk::message_hook/4).

% print silent messages instead of discarding them as default
logtalk::message_hook(_, silent, core, Tokens) :-
    logtalk::message_prefix_stream(silent, core, Prefix, Stream),
    logtalk::print_message_tokens(Stream, Prefix, Tokens).
```

See also

message_prefix_stream/4, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

message_prefix_stream/4

Description

```
message_prefix_stream(Kind, Component, Prefix, Stream)
```

User-defined hook method for specifying the default prefix and stream for printing a message for a given kind and *component*. This method is declared in the logtalk built-in object as a public, multifile, and dynamic predicate.

The prefix is printed for each line in the list of tokens generated by the message tokenization (new lines are specified using the `nl` token; see the documentation of the logtalk built-in object for details).

Modes and number of proofs

```
message_prefix_stream(?nonvar, ?nonvar, ?atom, ?stream_or_alias) - zero_or_more
```

Errors

(none)

Examples

```
:- multifile(logtalk::message_prefix_stream/4).
:- dynamic(logtalk::message_prefix_stream/4).

logtalk::message_prefix_stream(information, core, '% ', user_output).
```

See also

message_prefix_file/6, *message_hook/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

message_prefix_file/6

Description

```
message_prefix_file(Kind, Component, Prefix, File, Mode, Options)
```

Experimental user-defined hook method for specifying the prefix and file for copying messages for a given kind and *component*. This method is declared in the logtalk built-in object as a public, multifile, and dynamic predicate. The valid values for `Mode` are `write` and `append`. The valid options are the same as the standard `open/4` predicate.

The prefix is printed for each line in the list of tokens generated by the message tokenization (new lines are specified using the `nl` token; see the documentation of the logtalk built-in object for details).

Modes and number of proofs

```
message_prefix_file(?nonvar, ?nonvar, ?atom, ?atom, ?atom, ?list(compound)) - zero_or_more
```

Errors

(none)

Examples

```
:- multifile(logtalk::message_prefix_file/6).
:- dynamic(logtalk::message_prefix_file/6).

logtalk::message_prefix_file(comment, app, '% ', 'comments.txt', append, []).
```

See also

message_prefix_stream/4, *message_hook/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

print_message_tokens/3

Description

```
print_message_tokens(Stream, Prefix, Tokens)
```

Built-in method for printing a list of message tokens, declared in the logtalk built-in object as a public predicate. This method is automatically called by the *print_message/3* method (assuming that the message was not intercepted by a *message_hook/4* definition) and calls the user-defined hook predicate *print_message_token/4* for each token. When a call to this hook predicate succeeds, the `print_message_tokens/3` predicate assumes that the token has been printed. When the call fails, the `print_message_tokens/3` predicate uses a default printing procedure for the token.

Modes and number of proofs

```
print_message_tokens(@stream_or_alias, +atom, @list(nonvar)) - zero_or_one
```

Errors

(none)

Examples

```
...,
logtalk::print_message_tokens(user_error, '% ', ['Redefining ~w ~q'-[object,foo], nl]),
...
```

See also

message_hook/4, *message_prefix_stream/4*, *message_tokens//2*, *print_message/3*, *print_message_token/4*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

built-in method

print_message_token/4

Description

```
print_message_token(Stream, Prefix, Token, Tokens)
```

User-defined hook method for printing a message token, declared in the logtalk built-in object as a public, multifile, and dynamic predicate. It allows the user to intercept the printing of a message token. This hook method is automatically called by the *print_message_tokens/3* built-in method for each token.

Modes and number of proofs

```
print_message_token(@stream_or_alias, @atom, @nonvar, @list(nonvar)) - zero_or_one
```

Errors

(none)

Examples

```
:- multifile(logtalk::print_message_token/4).
:- dynamic(logtalk::print_message_token/4).

% ignore all flush tokens
logtalk::print_message_token(_Stream, _Prefix, flush, _Tokens).
```

See also

message_hook/4, *message_prefix_stream/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *ask_question/5*, *question_hook/6*, *question_prompt_stream/4*

2.5.14 Question asking

built-in method

ask_question/5

Description

```
ask_question(Kind, Component, Question, Check, Answer)
```

Built-in method for asking a question represented by a term, `Question`, which is converted to the question text using the *logtalk::message_tokens(Question, Component)* hook predicate. This method is declared in the `logtalk` built-in object as a public predicate. The default question prompt and the input stream used for each `Kind-Component` pair can be found using the *logtalk::question_prompt_stream(Kind, Component, Prompt, Stream)* hook predicate. The `Check` argument is a *closure* that is converted into a checking goal by extending it with the user supplied answer. This predicate implements a read-loop that terminates when the check goal succeeds.

This predicate starts by calling the *logtalk::question_hook(Question, Kind, Component, Tokens, Check, Answer)* hook predicate. If this predicate succeeds, the `ask_question/5` predicate assumes that the question have been successfully asked and replied.

Modes and number of proofs

```
ask_question(+nonvar, +nonvar, +nonvar, +callable, -term) - one
```

Meta-predicate template

```
ask_question(*, *, *, 1, *)
```

Errors

(none)

Examples

```
...,
logtalk::ask_question(enter_age, question, my_app, integer, Age),
...
```

See also

question_hook/6, *question_prompt_stream/4*, *message_hook/4*, *message_prefix_stream/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*

built-in method

question_hook/6

Description

```
question_hook(Question, Kind, Component, Tokens, Check, Answer)
```

User-defined hook method for intercepting asking a question, declared in the logtalk built-in object as a public, multifile, and dynamic predicate. This hook method is automatically called by the *ask_question/5* method. When the call succeeds, the `ask_question/5` method assumes that the question have been successfully asked and replied.

Modes and number of proofs

```
question_hook(+nonvar, +nonvar, +nonvar, +list(nonvar), +callable, -term) - zero_or_one
```

Meta-predicate template

```
question_hook(*, *, *, *, 1, *)
```

Errors

(none)

Examples

```
:- multifile(logtalk::question_hook/6).
:- dynamic(logtalk::question_hook/6).

% use a pre-defined answer instead of asking the user
logtalk::question_hook(upper_limit, question, my_app, _, _, 3.7).
```

See also

ask_question/5, *question_prompt_stream/4* *message_hook/4*, *message_prefix_stream/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*,

built-in method

question_prompt_stream/4

Description

```
question_prompt_stream(Kind, Component, Prompt, Stream)
```

User-defined hook method for specifying the default prompt and input stream for asking a question for a given kind and *component*. This method is declared in the logtalk built-in object as a public, multifile, and dynamic predicate.

Modes and number of proofs

```
question_prompt_stream(?nonvar, ?nonvar, ?atom, ?stream_or_alias) - zero_or_more
```

Errors

(none)

Examples

```
:- multifile(logtalk::question_prompt_stream/4).
:- dynamic(logtalk::question_prompt_stream/4).

logtalk::question_prompt_stream(question, debugger, '    > ', user_input).
```

See also

ask_question/5, *question_hook/6*, *message_hook/4*, *message_prefix_stream/4*, *message_tokens//2*, *print_message/3*, *print_message_tokens/3*, *print_message_token/4*

CHAPTER

THREE

TUTORIAL

3.1 List predicates

In this example, we will illustrate the use of:

- objects
- protocols

by using common list utility predicates.

3.1.1 Defining a list object

We will start by defining an object, `list`, containing predicate definitions for some common list predicates like `append/3`, `length/2`, and `member/2`:

```
:- object(list).

    :- public([
        append/3, length/2, member/2
    ]).

    append([], List, List).
    append([Head| Tail], List, [Head| Tail2]) :-
        append(Tail, List, Tail2).

    length(List, Length) :-
        length(List, 0, Length).

    length([], Length, Length).
    length([_| Tail], Acc, Length) :-
        Acc2 is Acc + 1,
        length(Tail, Acc2, Length).

    member(Element, [Element| _]).
    member(Element, [_| List]) :-
        member(Element, List).

:- end_object.
```

What is different here from a regular Prolog program? The definitions of the list predicates are the usual ones. We have two new directives, *object/1-5* and *end_object/0*, that encapsulate the object's code. In Logtalk,

by default, all object predicates are private; therefore, we have to explicitly declare all predicates that we want to be public, that is, that we want to call from outside the object. This is done using the *public/1* scope directive.

After we copy the object code to a text file and saved it under the name `list.lgt`, we need to change the Prolog working directory to the one used to save our file (consult your Prolog compiler reference manual). Then, after starting Logtalk (see the *Installing and running Logtalk* section on the User Manual), we can compile and load the object using the *logtalk_load/1* Logtalk built-in predicate:

```
| ?- logtalk_load(list).

object list loaded
yes
```

We can now try goals like:

```
| ?- list::member(X, [1, 2, 3]).

X = 1;
X = 2;
X = 3;
no
```

or:

```
| ?- list::length([1, 2, 3], L).

L = 3
yes
```

The infix operator *(::)/2* is used in Logtalk to send a message to an object. The message must match a public object predicate. If we try to call a non-public predicate such as the `length/3` auxiliary predicate an exception will be generated:

```
| ?- list::length([1, 2, 3], 0, L).

uncaught exception:
    error(
        existence_error(predicate_declaration, length/3),
        logtalk(list::length([1,2,3],0,_), ...)
    )
```

The exception term describes the type of error and the context where the error occurred.

3.1.2 Defining a list protocol

As we saw in the above example, a Logtalk object may contain predicate directives and predicate definitions (clauses). The set of predicate directives defines what we call the object's *protocol* or interface. An interface may have several implementations. For instance, we may want to define a new object that implements the list predicates using difference lists. However, we do not want to repeat the predicate directives in the new object. Therefore, what we need is to split the object's protocol from the object's implementation by defining a new Logtalk entity known as a protocol. Logtalk protocols are compilation units, at the same level as objects and categories. That said, let us define a `listp` protocol:

```
:- protocol(listp).

    :- public([
        append/3, length/2, member/2
    ]).

:- end_protocol.
```

Similar to what we have done for objects, we use the *protocol/1-2* and *end_protocol/0* directives to encapsulate the predicate directives. We can improve this protocol by documenting the call/return modes and the number of proofs of each predicate using the *mode/2* directive:

```
:- protocol(listp).

    :- public(append/3).
    :- mode(append(?list, ?list, ?list), zero_or_more).

    :- public(length/2).
    :- mode(length(?list, ?integer), zero_or_more).

    :- public(member/2).
    :- mode(member(?term, ?list), zero_or_more).

:- end_protocol.
```

We now need to change our definition of the `list` object by removing the predicate directives and by declaring that the object implements the `listp` protocol:

```
:- object(list,
    implements(listp)).

    append([], List, List).
    append([Head| Tail], List, [Head| Tail2]) :-
        append(Tail, List, Tail2).
    ...

:- end_object.
```

The protocol declared in `listp` may now be alternatively implemented using difference lists by defining a new object, `difflist`:

```
:- object(difflist,
    implements(listp).

    append(L1-X, X-L2, L1-L2).
    ...

:- end_object.
```

3.1.3 Summary

- It is easy to define a simple object: just write your Prolog code inside starting and ending object directives and add the necessary scope directives. The object will be self-defining and ready to use.
- Define a protocol when you may want to provide or enable several alternative definitions to a given set of predicates. This way we avoid needless repetition of predicate directives.

3.2 Dynamic object attributes

In this example, we will illustrate the use of:

- categories
- category predicates
- dynamic predicates

by defining a category that implements a set of predicates for handling dynamic object attributes.

3.2.1 Defining a category

We want to define a set of predicates to handle dynamic object attributes. We need public predicates to set, get, and delete attributes, and a private dynamic predicate to store the attribute values. Let us name these predicates `set_attribute/2` and `get_attribute/2`, for getting and setting an attribute value, `del_attribute/2` and `del_attributes/2`, for deleting attributes, and `attribute_/2`, for storing the attributes values.

But we do not want to encapsulate these predicates in an object. Why? Because they are a set of useful, closely related, predicates that may be used by several, unrelated, objects. If defined at an object level, we would be constrained to use inheritance in order to have the predicates available to other objects. Furthermore, this could force us to use multi-inheritance or to have some kind of generic root object containing all kinds of possible useful predicates.

For this kind of situation, Logtalk enables the programmer to encapsulate the predicates in a *category*, so that they can be used in any object. A category is a Logtalk entity, at the same level as objects and protocols. It can contain predicate directives and predicate definitions. Category predicates can be imported by any object, without code duplication and without resorting to inheritance.

When defining category predicates, we need to remember that a category can be imported by more than one object. Thus, calls to the built-in methods that handle the private dynamic predicate (such as *assertz/1* or *retract/1*) must be made either in the context of *self*, using the *message to self* control structure, *(::)/1*, or in the context of *this* (i.e., in the context of the object importing the category). This way, we ensure that when we call one of the attribute predicates on an object, the intended object own definition of `attribute_/2` will be used. The predicate definitions are straightforward. For example, if opting to store the attributes in *self*:

```
:- category(attributes).

    :- public(set_attribute/2).
    :- mode(set_attribute(+nonvar, +nonvar), one).

    :- public(get_attribute/2).
    :- mode(get_attribute(?nonvar, ?nonvar), zero_or_more).

    :- public(del_attribute/2).
```

(continues on next page)

(continued from previous page)

```
    :- mode(del_attribute(?nonvar, ?nonvar), zero_or_more).

    :- public(del_attributes/2).
    :- mode(del_attributes(@term, @term), one).

    :- private(attribute_/2).
    :- mode(attribute_(?nonvar, ?nonvar), zero_or_more).
    :- dynamic(attribute_/2).

    set_attribute(Attribute, Value):-
        ::retractall(attribute_(Attribute, _)),
        ::assertz(attribute_(Attribute, Value)).

    get_attribute(Attribute, Value):-
        ::attribute_(Attribute, Value).

    del_attribute(Attribute, Value):-
        ::retract(attribute_(Attribute, Value)).

    del_attributes(Attribute, Value):-
        ::retractall(attribute_(Attribute, Value)).

:- end_category.
```

The alternative, opting to store the attributes on *this*, is similar: just delete the `(::)/1` operator from the code above.

We have two new directives, *category/1-4* and *end_category/0*, that encapsulate the category code. If needed, we can put the predicate directives inside a protocol that will be implemented by the category:

```
:- category(attributes,
    implements(attributes_protocol)).

    ...

:- end_category.
```

Any protocol can be implemented by either an object, a category, or both.

3.2.2 Importing the category

We reuse a category's predicates by importing them into an object:

```
:- object(person,
    imports(attributes)).

    ...

:- end_object.
```

After compiling and loading this object and our category, we can now try queries like:

```
| ?- person::set_attribute(name, paulo).

yes

| ?- person::set_attribute(gender, male).

yes

| ?- person::get_attribute(Attribute, Value).

Attribute = name, Value = paulo ;
Attribute = gender, Value = male ;
no
```

3.2.3 Summary

- Categories are similar to objects: we just write our predicate directives and definitions bracketed by opening and ending category directives.
- An object reuses a category by importing it. The imported predicates behave as if they have been defined in the object itself.
- When do we use a category instead of an object? Whenever we have a set of closely related predicates that we want to reuse in several, unrelated, objects without being constrained by inheritance relations. Thus, categories can be interpreted as object building components.

3.3 A reflective class-based system

When compiling an object, Logtalk distinguishes prototypes from instances or classes by examining the object relations. If an object instantiates and/or specializes another object, then it is compiled as an instance or class, otherwise it is compiled as a prototype. A consequence of this is that, in order to work with instance or classes, we always have to define root objects for the instantiation and specialization hierarchies (however, we are not restricted to a single hierarchy). The best solution is often to define a reflective class-based system [Maes87], where every class is also an object and, as such, an instance of some class.

In this example, we are going to define the basis for a reflective class-based system, based on an extension of the ideas presented in [Cointe87]. This extension provides, along with root objects for the instantiation and specialization hierarchies, explicit support for abstract classes [Moura94].

3.3.1 Defining the base classes

We will start by defining three classes: `object`, `abstract_class`, and `class`. The class `object` will contain all predicates common to all objects. It will be the root of the inheritance graph:

```
:- object(object,
    instantiates(class)).

    % predicates common to all objects

:- end_object.
```

The class `abstract_class` specializes `object` by adding predicates common to all classes. It will be the default meta-class for abstract classes:

```
:- object(abstract_class,
    instantiates(class),
    specializes(object)).

    % predicates common to all classes

:- end_object.
```

The class `class` specializes `abstract_class` by adding predicates common to all instantiable classes. It will be the root of the instantiation graph and the default meta-class for instantiable classes:

```
:- object(class,
    instantiates(class),
    specializes(abstract_class)).

    % predicates common to all instantiable classes

:- end_object.
```

Note that all three objects are instances of class `class`. The instantiation and specialization relationships are chosen so that each object may use the predicates defined in itself and in the other two objects, with no danger of *message lookup* endless loops.

3.3.2 Summary

- An object that does not instantiate or specialize other objects is always compiled as a prototype.
- An instance must instantiate at least one object (its class). Similarly, a class must at least specialize or instantiate other object.
- The distinction between abstract classes and instantiable classes is an operational one, depending on the class inherited methods. A class is instantiable if inherits methods for creating instances. Conversely, a class is abstract if does not inherit any instance creation method.

3.4 Profiling programs

In this example, we will illustrate the use of:

- events
- monitors

by defining a simple profiler that prints the starting and ending time for processing a message sent to an object.

3.4.1 Messages as events

In a pure object-oriented system, all computations start by sending messages to objects. We can thus define an *event* as the sending of a message to an object. An event can then be specified by the tuple `(Object, Message, Sender)`. This definition can be refined by interpreting the sending of a message and the return of the control to the object that has sent the message as two distinct events. We call these events respectively `before` and `after`. Therefore, we end up by representing an event by the tuple `(Event, Object, Message, Sender)`. For instance, if we send the message:

```
| ?- foo::bar(X).

X = 1
yes
```

the two corresponding events will be:

```
(before, foo, bar(X), user)
(after, foo, bar(1), user)
```

Note that the second event is only generated if the message succeeds. If the message as a goal has multiple solutions, then an `after` event will be generated for each solution.

Events are automatically generated by the message-sending mechanisms for each public message sent using the *(::)/2* operator.

3.4.2 Profilers as monitors

A monitor is an object that reacts whenever a spied event occurs. The monitor actions are defined by two event handlers: *before/3* for `before` events and *after/3* for `after` events. These predicates are automatically called by the message-sending mechanisms when an event registered for the monitor occurs. These event handlers are declared as public predicates in the `monitoring` built-in protocol.

In our example, we need a way to get the current time before and after we process a message. We will assume that we have a `time` object implementing a `cpu_time/1` predicate that returns the current CPU time for the Prolog session:

```
:- object(time).

    :- public(cpu_time/1).
    :- mode(cpu_time(-number), one).
    ...

:- end_object.
```

Our profiler will be named `stop_watch`. It must define event handlers for the `before` and `after` events that will print the event description (object, message, and sender) and the current time:

```
:- object(stop_watch,
    % event handler predicates protocol
    implements(monitoring)).

    :- uses(time, [cpu_time/1]).

    before(Object, Message, Sender) :-
```

(continues on next page)

(continued from previous page)

```
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl, write('STARTING at '),
        cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

    after(Object, Message, Sender) :-
        write(Object), write(' <-- '), writeq(Message),
        write(' from '), write(Sender), nl, write('ENDING at '),
        cpu_time(Seconds), write(Seconds), write(' seconds'), nl.

:- end_object.
```

After compiling and loading the `stop_watch` object (and the objects that we want to profile), we can use the *define_events/5* built-in predicate to set up our profiler. For example, to profile all messages that are sent to the object `foo`, we need to call the goal:

```
| ?- define_events(_, foo, _, _, stop_watch).

yes
```

This call will register `stop_watch` as a monitor to all messages sent to object `foo`, for both `before` and `after` events. Note that we say "as a monitor", not "the monitor": we can have any number of monitors over the same events.

From now on, every time we sent a message to `foo`, the `stop_watch` monitor will print the starting and ending times for the message execution. For instance:

```
| ?- foo::bar(X).

foo <-- bar(X) from user
STARTING at 12.87415 seconds
foo <-- bar(1) from user
ENDING at 12.87419 seconds

X = 1
yes
```

To stop profiling the messages sent to `foo` we use the *abolish_events/5* built-in predicate:

```
| ?- abolish_events(_, foo, _, _, stop_watch).

yes
```

This call will abolish all events defined over the object `foo` assigned to the `stop_watch` monitor.

3.4.3 Summary

- An event is defined as the sending of a (public) message to an object.
- There are two kinds of events: `before` events, generated before a message is processed, and `after` events, generated after the message processing has completed successfully.
- Any object can be declared as a monitor to any event. A monitor shall reference the `monitoring` built-in protocol in the object opening directive.
- A monitor defines event handlers, the predicates *before/3* and *after/3*, that are automatically called by the runtime engine when a spied event occurs.
- Three built-in predicates, *define_events/5*, *current_event/5*, and *abolish_events/5*, enables us to define, query, and abolish both events and monitors.

CHAPTER

FOUR

FAQ

4.1 General

- *Why are all versions of Logtalk numbered 2.x or 3.x?*
- *Why do I need a Prolog compiler to use Logtalk?*
- *Is the Logtalk implementation based on Prolog modules?*
- *Does the Logtalk implementation use term-expansion?*

4.1.1 Why are all versions of Logtalk numbered 2.x or 3.x?

The numbers "2" and "3" in the Logtalk version string refer to, respectively, the second and the third generations of the Logtalk language. Development of Logtalk 2 started in January 1998, with the first alpha release for registered users in July and the first public beta on October. The first stable version of Logtalk 2 was released in February 9, 1999. Development of Logtalk 3 started in April 2012, with the first public alpha released in August 21, 2012. The first stable version of Logtalk 3 was released in January 7, 2015.

4.1.2 Why do I need a Prolog compiler to use Logtalk?

Currently, the Logtalk language is implemented as a *trans-compiler* to Prolog instead of as a standalone compiler. Compilation of Logtalk source files is performed in two steps. First, the Logtalk compiler converts a source file to a Prolog file. Second, the chosen Prolog compiler is called by Logtalk to compile the intermediate Prolog file generated on the first step. The implementation of Logtalk as a trans-compiler allows users to use Logtalk together with features only available on specific Prolog compilers.

4.1.3 Is the Logtalk implementation based on Prolog modules?

No. Logtalk is (currently) implemented in plain Prolog code. Only a few Prolog compilers include a module system, with several compatibility problems between them. Moreover, the current ISO Prolog standard for modules is next to worthless and is ignored by most of the Prolog community. Nevertheless, the Logtalk compiler is able to compile simple modules (using a common subset of module directives) as objects for backward-compatibility with existing code (see the *Prolog integration and migration* for details).

4.1.4 Does the Logtalk implementation use term-expansion?

No. Term-expansion mechanisms are not standard and are not available in all supported Prolog compilers.

4.2 Compatibility

- *What are the backend Prolog compiler requirements to run Logtalk?*
- *Can I use constraint-based packages with Logtalk?*
- *Can I use Logtalk objects and Prolog modules at the same time?*

4.2.1 What are the backend Prolog compiler requirements to run Logtalk?

See the backend Prolog compiler requirements guide.

4.2.2 Can I use constraint-based packages with Logtalk?

Usually, yes. Some constraint-based packages may define operators which clash with the ones defined by Logtalk. In these cases, compatibility with Logtalk depends on the constraint-based packages providing an alternative for accessing the functionality provided by those operators. When the constraint solver is encapsulated using a Prolog module, a possible workaround is to use either explicit module qualification or encapsulate the call using the *{}/1* control construct (thus bypassing the Logtalk compiler).

4.2.3 Can I use Logtalk objects and Prolog modules at the same time?

Yes. In order to call a module predicate from within an object (or category) you may use an *use_module/2* directive or use explicit module qualification (possibly wrapping the call using the Logtalk control construct *{}/1* that allows bypassing of the Logtalk compiler when compiling a predicate call). Logtalk also allows modules to be compiled as objects (see the *Prolog integration and migration* for details).

4.3 Installation

- *The integration scripts/shortcuts are not working!*
- *I get errors when starting up Logtalk after upgrading to the latest version!*

4.3.1 The integration scripts/shortcuts are not working!

Check that the `LOGTALKHOME` and `LOGTALKUSER` environment variables are defined, that the Logtalk user folder is available in the location pointed by `LOGTALKUSER` (you can create this folder by running the `logtalk_user_setup` shell script), and that the Prolog compilers that you want to use are supported and available from the system path. If the problem persists, run the shell script that creates the integration script or shortcut manually and check for any error message or additional instructions. For some Prolog compilers such as XSB and Ciao, the first call of the integration script or shortcut must be made by an administrator user. If you are using Windows, make sure that any anti-virus or other security software that you might have installed is not silently blocking some of the installer tasks.

4.3.2 I get errors when starting up Logtalk after upgrading to the latest version!

Changes in the Logtalk compiler between releases may render Prolog adapter files from older versions incompatible with new ones. You may need to update your local Logtalk user files by running e.g. the `logtalk_user_setup` shell script. Check the `UPGRADING.md` file on the root of the Logtalk installation directory and the release notes for any incompatible changes to the adapter files.

4.4 Portability

- *Are my Logtalk applications portable across Prolog compilers?*
- *Are my Logtalk applications portable across operating systems?*

4.4.1 Are my Logtalk applications portable across Prolog compilers?

Yes, as long you don't use built-in predicates or special features only available on some Prolog compilers. There is a *portability* compiler flag that you can set to instruct Logtalk to print a warning for each occurrence of non-ISO Prolog standard features such as proprietary built-in predicates. In addition, it is advisable that you constrain, if possible, the use of platform or compiler dependent code to a small number of objects with clearly defined protocols. You may also use Logtalk support for conditional compilation to compile different entity or predicate definitions depending on the backend Prolog compiler being used.

4.4.2 Are my Logtalk applications portable across operating systems?

Yes, as long you don't use built-in predicates or special features that your chosen backend Prolog compiler only supports on some operating-systems. You may need to change the end-of-line characters of your source files to match the ones on the target operating system and the expectations of your Prolog compiler. Some Prolog compilers silently fail to compile source files with the wrong end-of-line characters.

4.5 Programming

- *Should I use prototypes or classes in my application?*
- *Can I use both classes and prototypes in the same application?*
- *Can I mix classes and prototypes in the same hierarchy?*
- *Can I use a protocol or a category with both prototypes and classes?*
- *What support is provided in Logtalk for defining and using components?*
- *What support is provided in Logtalk for reflective programming?*

4.5.1 Should I use prototypes or classes in my application?

Prototypes and classes provide different patterns of code reuse. A prototype encapsulates code that can be used by itself and by its descendent prototypes. A class encapsulates code to be used by its descendent instances. Prototypes provide the best replacement to the use of modules as encapsulation units, avoiding the need to instantiate a class in order to access its code.

4.5.2 Can I use both classes and prototypes in the same application?

Yes. In addition, you may freely exchange messages between prototypes, classes, and instances.

4.5.3 Can I mix classes and prototypes in the same hierarchy?

No. However, you may define as many prototype hierarchies and class hierarchies and classes as needed by your application.

4.5.4 Can I use a protocol or a category with both prototypes and classes?

Yes. A protocol may be implemented by both prototypes and classes in the same application. Likewise, a category may be imported by both prototypes and classes in the same application.

4.5.5 What support is provided in Logtalk for defining and using components?

Logtalk supports component-based programming (since its inception on January 1998), by using *categories* (which are first-class entities like objects and protocols). Logtalk categories can be used with both classes and prototypes and are inspired by the Smalltalk-80 (documentation-only) concept of method categories and by Objective-C categories, hence the name. For more information, please consult the *Categories* section and the examples provided with the distribution.

4.5.6 What support is provided in Logtalk for reflective programming?

Logtalk supports meta-classes, behavioral reflection through the use of event-driven programming, and structural reflection through the use of a set of built-in predicates and built-in methods which allow us to query the system about existing entities, entity relations, and entity predicates.

4.6 Troubleshooting

- *Using compiler options on calls to the Logtalk compiling and loading predicates does not work!*
- *Gecko-based browsers (e.g., Firefox) show non-rendered HTML entities when browsing XML documenting files!*
- *Compiling a source file results in errors or warnings but the Logtalk compiler reports a successful compilation with zero errors and zero warnings!*

4.6.1 Using compiler options on calls to the Logtalk compiling and loading predicates does not work!

Using compiler options on calls to the Logtalk *logtalk_compile/2* and *logtalk_load/2* built-in predicates only apply to the file being compiled. If the first argument is a *loader file*, the compiler options will only be used in the compilation of the loader file itself, not in the compilation of the files loaded by the loader file. The solution is to edit the loader file and add the compiler options to the calls that compile/load the individual files.

4.6.2 Gecko-based browsers (e.g., Firefox) show non-rendered HTML entities when browsing XML documenting files!

Using Gecko-based browsers (e.g., Firefox) show non-rendered HTML entities (e.g., `–`) when browsing XML documenting files after running the `lgt2xml` shell script in the directory containing the XML documenting files. This is a consequence of the lack of support for the `disable-output-escaping` attribute in the browser XSLT processor. The workaround is to use another browser (e.g., Safari or Opera) or to use instead the `lgt2html` shell script in the directory containing the XML documenting files to convert them to (X)HTML files for browsing.

4.6.3 Compiling a source file results in errors or warnings but the Logtalk compiler reports a successful compilation with zero errors and zero warnings!

This may happen when your Prolog compiler implementation of the ISO Prolog standard `write_canonical/2` built-in predicate is buggy and writes terms that cannot be read back when consulting the intermediate Prolog files generated by the Logtalk compiler. Often, syntax errors found when consulting result in error messages but not in exceptions as the Prolog compiler tries to continue the compilation despite the problems found. As the Logtalk compiler relies on the exception mechanisms to catch compilation problems, it may report zero errors and zero warnings despite the error messages. Send a bug report to the Prolog compiler developers asking them to fix the `write_canonical/2` buggy implementation.

4.7 Usability

- *Is there a shortcut for compiling and loading source files?*
- *Is there an equivalent directive to the ensure_loaded/1 Prolog directive?*
- *Are there shortcuts for the make functionality?*

4.7.1 Is there a shortcut for compiling and loading source files?

Yes. With most backend Prolog compilers, you can use `{File}` as a shortcut for `logtalk_load(File)`. For compiling and loading multiple files simply use `{File1, File2, ...}`. See the documentation of the *logtalk_load/1* predicate for details.

4.7.2 Is there an equivalent directive to the `ensure_loaded/1` Prolog directive?

You can use the goal `logtalk_load(File, [reload(skip)])` to ensure that `File` it is only loaded once. See the documentation of the *logtalk_load/2* predicate for details.

4.7.3 Are there shortcuts for the make functionality?

Yes. With most backend Prolog compilers, you can use `{*}` as a shortcut for `logtalk_make(all)` to reload all files modified since last compiled and loaded, `{!}` as a shortcut for `logtalk_make(clean)` to delete all intermediate Prolog files generated by the compilation of Logtalk source files, `{?}` as a shortcut for `logtalk_make(missing)` to list missing entities and predicates, and `{@}` as a shortcut for `logtalk_make(circular)` to list circular references. See the documentation of the *logtalk_make/1* predicate for details.

4.8 Deployment

- *Can I create standalone applications with Logtalk?*

4.8.1 Can I create standalone applications with Logtalk?

It depends on the Prolog compiler that you use to run Logtalk. Assuming that your Prolog compiler supports the creation of standalone executables, your application must include the adapter file for your compiler and the Logtalk compiler and runtime. The distribution includes embedding scripts for selected backend Prolog compilers and embedding examples.

For instructions on how to embed Logtalk and Logtalk applications, see the embedding guide.

4.9 Performance

- *Is Logtalk implemented as a meta-interpreter?*
- *What kind of code Logtalk generates when compiling objects? Dynamic code? Static code?*
- *How about message-sending performance? Does Logtalk use static binding or dynamic binding?*
- *How does Logtalk performance compare with plain Prolog and with Prolog modules?*

4.9.1 Is Logtalk implemented as a meta-interpreter?

No. Objects and their encapsulated predicates are compiled, not meta-interpreted. In particular, inheritance relations are pre-compiled for improved performance. Moreover, no meta-interpreter is used even for objects compiled in debug mode.

4.9.2 What kind of code Logtalk generates when compiling objects? Dynamic code? Static code?

Static objects are compiled to static code. Static objects containing dynamic predicates are also compiled to static code, except, of course, for the dynamic predicates themselves. Dynamic objects are necessarily compiled to dynamic code. As in Prolog programming, for best performance, dynamic object predicates and dynamic objects should only be used when truly needed.

4.9.3 How about message-sending performance? Does Logtalk use static binding or dynamic binding?

Logtalk supports both static binding and dynamic binding. When static binding is not possible, Logtalk uses dynamic binding coupled with a caching mechanism that avoids repeated lookups of predicate declarations and predicate definitions. This is a solution common to other programming languages supporting dynamic binding. Message lookups are automatically cached the first time a message is sent. Cache entries are automatically removed when loading entities or using Logtalk dynamic features that invalidate the cached lookups. Whenever static binding is used, message-sending performance is essentially the same as a predicate call in plain Prolog. Performance of dynamic binding when lookups are cached is close to the performance that would be achieved with static binding. See the User Manual section on *performance* for more details.

4.9.4 Which Prolog-dependent factors are most crucial for good Logtalk performance?

Logtalk compiles objects assuming first-argument indexing for static code. First-argument indexing of dynamic code, when available, helps to improve performance due to the automatic caching of method lookups and the necessary use of book-keeping tables by the runtime engine (this is specially important when using event-driven programming). Dynamic objects and static objects containing dynamic predicates also benefit from first-argument indexing of dynamic predicates. The availability of multi-argument indexing, notably for dynamic predicates, also benefits dynamic binding performance.

4.9.5 How does Logtalk performance compare with plain Prolog and with Prolog modules?

Plain Prolog, Prolog modules, and Logtalk objects provide different trade-offs between performance and features. In general, for a given predicate definition, the best performance will be attained using plain Prolog, second will be Prolog modules (assuming no explicitly qualified calls are used), and finally Logtalk objects. Whenever static binding is used, the performance of Logtalk is equal or close to that of plain Prolog (depending on the Prolog virtual machine implementation and compiler optimizations). See the simple benchmark test results using some popular Prolog compilers.

4.10 Licensing

- *What's the Logtalk distribution license?*
- *Can Logtalk be used in commercial applications?*
- *What's the final license for a combination of Logtalk with a Prolog compiler?*

4.10.1 What's the Logtalk distribution license?

Logtalk follows the Apache License 2.0.

4.10.2 Can Logtalk be used in commercial applications?

Yes, the Apache License 2.0 allows commercial use. See e.g. the Apache License and Distribution FAQ.

4.10.3 What's the final license for a combination of Logtalk with a Prolog compiler?

See the licensing guide for details and relevant resources.

4.11 Support

- *Are there professional consulting, training and supporting services?*

4.11.1 Are there professional consulting, training and supporting services?

Yes. Please visit logtalk.pt for professional consulting, development, training, and other support services.

CHAPTER

FIVE

DEVELOPER TOOLS

The documentation of each developer tool can also be found in the tool directory in the `NOTES.md` file.

5.1 Overview

The following developer tools are available, each one with its own `loader.lgt` loader file (except for the built-in `linter` and `make` tools, which are integrated with the compiler/runtime) and `NOTES.md` documentation files:

- `asdf`
- `assertions`
- `code_metrics`
- `dead_code_scanner`
- `debug_messages`
- `debugger`
- `diagrams`
- `doclet`
- `help`
- `issue_creator`
- `lgtdoc`
- `lgtunit`
- `linter`
- `make`
- `packs`
- `ports_profiler`
- `profiler`
- `tutor`
- `wrapper`

5.1.1 Loading the developer tools

To load the main developer tools, use the following goal:

```
| ?- logtalk_load(tools(loader)).
```

The `ports_profiler` tool is not loaded by default; however, as it conflicts with the `debugger` tool as both provide a debug handler that must be unique in a running session.

The `profiler` tool is also not loaded by default, as it provides integration with selected backend Prolog compiler profilers that are not portable.

The `tutor` tool is also not loaded by default, given its usefulness mainly for new users that need help understanding compiler warning and error messages.

The `wrapper` tool is also not loaded by default, given its specialized purpose and beta status.

To load a specific tool, either change your Prolog working directory to the tool folder and then compile and load the corresponding loader utility file or simply use library notation as argument for the compiling and loading predicates. For example:

```
| ?- logtalk_load(lgtunit(loader)).
```

5.1.2 Tools documentation

Specific notes about each tool can be found in the corresponding `NOTES.md` files. HTML documentation for each tool API can be found in the `docs` directory (open the `../docs/handbook/index.html` file with your web browser).

5.1.3 Tools common flags

The `lgtdoc`, `lgtunit`, and `issue_creator` tools share a `suppress_path_prefix` flag that can be used to suppress a prefix when printing file paths. For example (after loading the tools):

```
| ?- set_logtalk_flag(suppress_path_prefix, '/home/jdoe/').
```

This flag is mainly used to avoid user specific path prefixes appearing in documentation, test logs, and bug reports.

5.1.4 Tools requirements

Some of the developer tools have third-party dependencies. For example, the `lgtdoc` tool depends on XSLT processors to generate documentation final formats and uses Sphinx for the preferred HTML final format. Be sure to consult the tools documentation details on those requirements and possible alternatives. For convenience, follows a global list of the main tool requirements and suggestions for installing them per operating-system. If your operating-system or a dependency for it is not listed, see the dependency websites for installation instructions.

Tool dependencies for full functionality

- `diagrams`: d2, Graphviz
- `help`: `info`
- `issue_creator`: `gh`, `glab`
- `lgtdoc`: Sphinx, `libxslt`, `fop`, `texlive`, `texinfo`, `pandoc`, `xsltproc`
- `lgtunit`: Allure, `coreutils`, `gsed`
- `packs`: `coreutils`, `libarchive`, `gnupg2`, `git`, `curl`, `wget`, `direnv`

Python dependencies (all operating-systems)

```
$ pip install --upgrade pygments
$ pip install --upgrade sphinx
$ pip install --upgrade sphinx_rtd_theme
```

In alternative, if your want to use the same versions used to build release documentation:

```
$ cd manuals/sources
$ python3 -m pip install -r requirements.txt
```

macOS - MacPorts

```
$ sudo port install d2 graphviz
$ sudo port install texinfo pandoc
$ sudo port install libxslt fop texlive
$ sudo port install gsed
$ sudo port install coreutils gsed libarchive gnupg2 git curl wget direnv
$ sudo port install gh glab
```

macOS - Homebrew

```
$ brew install d2 graphviz
$ brew install texinfo pandoc
$ brew install libxslt fop texlive
$ brew install allure gsed
$ brew install coreutils libarchive gnupg2 git curl wget direnv
$ brew install gh glab
```

Ubuntu

```
$ sudo apt install graphviz
$ sudo apt install texinfo pandoc
$ sudo apt install xsltproc fop texlive-latex-extra tex-gyre latexmk
$ sudo apt install coreutils libarchive-tools gnupg2 git curl wget direnv
$ sudo apt install gh
```

RedHat

```
$ sudo dnf install graphviz
$ sudo dnf install texinfo pandoc
$ sudo dnf install libxslt fop
$ sudo dnf install bsdtar gnupg2 git curl wget direnv
```

Windows - Chocolatey

```
> choco install pandoc xsltproc
> choco install d2 graphviz
> choco install xsltproc apache-fop texlive
> choco install gnupg git
> choco install gh glab
> choco install wget
> choco install sed
```

Installers

- https://d2lang.com/tour/install
- https://www.graphviz.org/download/
- https://allurereport.org/docs/
- https://www.gnupg.org/
- https://gitforwindows.org
- https://cli.github.com
- https://glab.readthedocs.io
- https://eternallybored.org/misc/wget/

Windows - PowerShell add-ons

```
PS> Install-Module -Name Set-PsEnv
```

5.2 asdf

A Logtalk plugin for the asdf extendable version manager is available at:

https://github.com/LogtalkDotOrg/asdf-logtalk

This plugin provides an alternative to the `logtalk_version_select` script that can be useful to manage Logtalk versions when developing solutions that use other languages and tools that can also be handled by `asdf`.

5.3 assertions

The `assertions.lgt` file contains definitions for two meta-predicates, `assertion/1-2`, which allows the use of assertions on your source code to print warning and error messages (using the message printing mechanism). The `assertions_messages.lgt` file defines the default message translations generated when assertions succeed, fail, or throw an exception.

5.3.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#assertions

5.3.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(assertions(loader)).
```

5.3.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(assertions(tester)).
```

5.3.4 Adding assertions to your source code

The `assertion/1` predicate takes a goal as argument. For example:

```
foo(L) :-
    assertions::assertion(non_empty_list(L)),
    ...
```

The `assertion/2` predicate takes as arguments a term for passing context information and a goal. Using again a unit test as an example:

```
foo(L) :-
    assertions::assertion(foo_list_alerts, non_empty_list(L)),
    ...
```

When using a large number of assertions, you can use a lighter syntax by adding a `uses/2` directive. For example:

```
:- uses(assertions, [assertion/1, assertion/2]).
```

5.3.5 Automatically adding file and line context information to assertions

The `assertions/1` parametric object can be used as a hook object to automatically add file and line context information, represented by the term `file_lines(File, BeginLine-EndLine)`, to calls to the `assertion/1` predicate by goal-expanding it to calls to the `assertion/2` predicate (the expansion assumes that a `uses/2` directive is being used in the code that will be expanded to direct `assertion/1` calls to the `assertions` object). For example, assuming the file using assertions is named `source`, it would be compiled and loaded using the call:

```
logtalk_load(source, [hook(assertions(debug))])
```

5.3.6 Suppressing assertion calls from source code

The `assertions/1` parametric object can be used as a hook object to suppress calls to the `assertion/1-2` predicates using goal-expansion (the expansion assumes `assertions::assertion/1-2` messages). For example, assuming the file using assertions is named `source`, it would be compiled and loaded using the call:

```
logtalk_load(source, [hook(assertions(production))])
```

5.3.7 Redirecting assertion failure messages

By default, assertion failures and errors are printed to the standard output stream. These messages, however, can be intercepted by defining the `logtalk::message_hook/4` multifile predicate. For example:

```
:- category(redirect_assertions_messages).

    :- multifile(logtalk::message_hook/4).
    :- dynamic(logtalk::message_hook/4).

    logtalk::message_hook(Message, error, assertions, _) :-
```

(continues on next page)

(continued from previous page)

```
        writeq(my_log_file, Message), write(my_log_file, '.\n').

:- end_category.
```

5.3.8 Converting assertion failures into errors

If you want an assertion failure to result in a failure or a runtime error, you can intercept the assertion failure messages, optionally still printing them, and throw an error. For example:

```
:- category(assertions_failures_to_errors).

    :- multifile(logtalk::message_hook/4).
    :- dynamic(logtalk::message_hook/4).

    logtalk::message_hook(Message, error, assertions, Tokens) :-
        % uncomment the next two lines to also print the default message
        % logtalk::message_prefix_stream(error, assertions, Prefix, Stream),
        % logtalk::print_message_tokens(Stream, Prefix, Tokens),
        throw(error(Message, _)).

:- end_category.
```

In alternative, if you want assertions to always trigger an exception, use instead the `lgtunit` tool `assertions/1-2` public predicates.

5.4 `code_metrics`

The purpose of this tool is to assess qualities of source code that may predict negative aspects such as entity coupling, cohesion, complexity, error-proneness, and overall maintainability. It is meant to be extensible via the addition of objects implementing new metrics.

This tool provides predicates for computing metrics for source files, entities, libraries, files, and directories. The actual availability of a particular predicate depends on the specific metric. A set of predicates prints, by default, the computed metric values to the standard output. A second set of predicates computes and returns a score (usually a compound term with the computed metric values as arguments).

5.4.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#code-metrics

5.4.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(code_metrics(loader)).
```

5.4.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(code_metrics(tester)).
```

5.4.4 Available metrics

Currently, the following metrics are provided:

- Number of Clauses (`noc_metric`)
- Number of Rules (`nor_metric`)
- Unique Predicate Nodes (`upn_metric`)
- Cyclomatic Complexity (`cc_metric`)
- Depth of Inheritance (`dit_metric`)
- Efferent coupling, afferent coupling, instability, and abstractness (`coupling_metric`)
- Documentation (`doc_metric`)
- Source code size (`size_metric`)
- Halstead complexity (`halstead_metric` and `halstead_metric(Stroud)`)

A helper object, `code_metrics`, is also provided, allowing running all loaded individual metrics. For code coverage metrics, see the `lgtunit` tool documentation.

5.4.5 Coupling metrics

- Efferent coupling (`Ce`): Number of entities that an entity depends on. These include objects receiving messages from the entity, plus the implemented protocols, imported categories, and extended/instantiated/specialized objects.
- Afferent coupling (`Ca`): Number of entities that depend on an entity. For a protocol, the number of protocols that extend it, plus the number of objects and categories that implement it. For a category, the number of objects that import it. For an object, the number of categories and objects that send messages to it plus the number of objects that extend/instantiate/specialize it.
- Instability: Computed as `Ce / (Ce + Ca)`. Measures an entity resilience to change. Ranging from 0.0 to 1.0, with 0.0 indicating a maximally stable entity and 1.0 indicating a maximally unstable entity. Ideally, an entity is either maximally stable or maximally unstable.
- Abstractness: Computed as the ratio between the number of static predicates with scope directives without a local definition and the number of static predicates with scope directives. Measures the rigidity of an entity. Ranging from 0.0 to 1.0, with 0.0 indicating a fully concrete entity and 1.0 indicating a fully abstract entity.

The dependencies count includes direct entity relations plus predicate calls or dynamic updates to predicates in external objects or categories.

For more information on the interpretation of the coupling metric scores, see, e.g., the original paper by Robert Martin:

```
@inproceedings{citeulike:1579528,
    author = "Martin, Robert",
    booktitle = "Workshop Pragmatic and Theoretical Directions in Object-Oriented Software␣
↪Metrics",
    citeulike-article-id = 1579528,
    citeulike-linkout-0 = "http://www.objectmentor.com/resources/articles/oodmetrc.pdf",
    keywords = "diplomarbeit",
    organization = "OOPSLA'94",
    posted-at = "2007-08-21 11:08:44",
    priority = 0,
    title = "OO Design Quality Metrics - An Analysis of Dependencies",
    url = "http://www.objectmentor.com/resources/articles/oodmetrc.pdf",
    year = 1994
}
```

The coupling metric was also influenced by the metrics rating system in Microsoft Visual Studio and aims to eventually emulate the functionality of a maintainability index score.

5.4.6 Halstead metric

Predicates declared, user-defined, and called are interpreted as *operators*. Built-in predicates and built-in control constructs are ignored. Predicate arguments are abstracted, assumed distinct, and interpreted as *operands*. Note that this definition of operands is a significant deviation from the original definition, which used syntactic literals. A computation closer to the original definition of the metric would require switching to use the parser to collect information on syntactic literals, which would imply a much larger computation cost. The number of predicate calls doesn't include calls to built-in predicates and can underestimate recursive calls.

The computation of this metric is parameterized by the *Stroud* coefficient for computing the time required to program (default is 18). The following individual measures are computed:

- Number of distinct predicates (declared, defined, called, or updated; `Pn`).
- Number of predicate arguments (assumed distinct; `PAn`).
- Number of predicate calls/updates + number of clauses (`Cn`).
- Number of predicate call/update arguments + number of clause head arguments (`CAn`).
- Entity vocabulary (`EV`). Computed as `EV = Pn + PAn`.
- Entity length (`EL`). Computed as `EL = Cn + CAn`.
- Volume (`V`). Computed as `V = EL * log2(EV)`.
- Difficulty (`D`). Computed as `D = (Pn/2) * (CAn/An)`.
- Effort (`E`). Computed as `E = D * V`.
- Time required to program (`T`). Computed as `T = E/k` seconds (where `k` is the Stroud number; defaults to 18).
- Number of delivered bugs (`B`). Computed as `B = V/3000`.

5.4.7 UPN metric

The Unique Predicate Nodes (UPN) metric is described in the following paper:

```
@article{MOORES199845,
    title = "Applying Complexity Measures to Rule-Based Prolog Programs",
    journal = "Journal of Systems and Software",
    volume = "44",
    number = "1",
    pages = "45 - 52",
    year = "1998",
    issn = "0164-1212",
    doi = "https://doi.org/10.1016/S0164-1212(98)10042-0",
    url = "http://www.sciencedirect.com/science/article/pii/S0164121298100420",
    author = "Trevor T Moores"
}
```

The nodes include called and updated predicates independently of where they are defined. It also includes multifile predicates contributed to other entities.

5.4.8 Cyclomatic complexity metric

The cyclomatic complexity metric evaluates an entity code complexity by measuring the number of linearly independent paths through the code. In its current implementation, all defined predicates that are not called or updated are counted as graph-connected components (the reasoning being that these predicates can be considered entry points). The implementation uses the same predicate abstraction as the UPN metric. The defined predicates include multifile predicate definitions contributed by the entity to other entities.

For more details on this metric, see the original paper by Thomas J. McCabe:

```
@inproceedings{McCabe:1976:CM:800253.807712,
    author = "McCabe, Thomas J.",
    title = "A Complexity Measure",
    booktitle = "Proceedings of the 2Nd International Conference on Software Engineering",
    series = "ICSE '76",
    year = 1976,
    location = "San Francisco, California, USA",
    pages = "407--",
    url = "http://dl.acm.org/citation.cfm?id=800253.807712",
    acmid = 807712,
    publisher = "IEEE Computer Society Press",
    address = "Los Alamitos, CA, USA",
    keywords = "Basis, Complexity measure, Control flow, Decomposition, Graph theory,␣
↪Independence, Linear, Modularization, Programming, Reduction, Software, Testing",
}
```

5.4.9 Usage

All metrics require the source code to be analyzed to be loaded with the `source_data` flag turned on. For usage examples, see the `SCRIPT.txt` file in the tool directory.

Be sure to fully understand the metrics individual meanings and any implementation limitations before using them to support any evaluation or decision process.

5.4.10 Excluding code from analysis

A set of options is available to specify code that should be excluded when applying code metrics:

- `exclude_directories(Directories)`
  list of directories to exclude (default is `[]`); all sub-directories of the excluded directories are also excluded; directories may be listed by full or relative path
- `exclude_files(Files)`
  list of source files to exclude (default is `[]`); files may be listed by full path or basename, with or without extension
- `exclude_libraries(Libraries)`
  list of libraries to exclude (default is `[startup, scratch_directory]`)
- `exclude_entities(Entities)`
  list of entities to exclude (default is `[]`)

5.4.11 Defining new metrics

New metrics can be implemented by defining an object that imports the `code_metric` category and implements its score predicates. There is also a `code_metrics_utilities` category that defines useful predicates for the definition of metrics.

5.4.12 Third-party tools

The following open-source command-line programs can count blank lines, comment lines, and lines of source code in many programming languages, including Logtalk:

- `cloc` - https://github.com/AlDanial/cloc
- `ohcount` - https://github.com/blackducksoftware/ohcount
- `tokei` - https://github.com/XAMPPRocky/tokei

5.4.13 Applying metrics to Prolog modules

Some of the metrics can also be applied to Prolog modules that Logtalk is able to compile as objects. For example, if the Prolog module file is named `module.pl`, try:

```
| ?- logtalk_load(module, [source_data(on)]).
```

Due to the lack of standardization of module systems and the abundance of proprietary extensions, this solution is not expected to work for all cases.

5.4.14 Applying metrics to plain Prolog code

Some of the metrics can also be applied to plain Prolog code. For example, if the Prolog file is named `code.pl`, simply define an object including its code:

```
:- object(code).
    :- include('code.pl').
:- end_object.
```

Save the object to an e.g. `code.lgt` file in the same directory as the Prolog file and then load it in debug mode:

```
| ?- logtalk_load(code, [source_data(on)]).
```

In alternative, use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(code, [hook(object_wrapper_hook), source_data(on)]).
```

With either wrapping solution, pay special attention to any compilation warnings that may signal issues that could prevent the plain Prolog code from working when wrapped by an object.

5.5 dead_code_scanner

This tool detects *likely* dead code in Logtalk entities and in Prolog modules compiled as objects. Predicates (and non-terminals) are classified as dead code when:

- There is no scope directive for them, and they are not called, directly or indirectly, by any predicate with a (local or inherited) scope directive.
- They are listed in `uses/2` and `use_module/2` directives but not called.

Besides dead code, this tool can also help detect other problems in the code that often result in reporting false positives. For example, typos in `alias/2` directives, missing scope directives, and missing `meta_non_terminal/1` and `meta_predicate/1` directives.

Given the possibility of false positives, care must be taken before deleting reported dead code to ensure that it's, in fact, code that is not used. A common cause of false positives is the use of conditional compilation directives to provide implementations for predicates missing in some systems or different predicate implementations per operating-system.

The `dead_code_scanner.lgt` source file implements the scanning predicates for finding dead code in entities, libraries, and directories. The source file `dead_code_scanner_messages.lgt` defines the default translations for the messages printed when scanning for dead code. These messages can be intercepted to customize the output, e.g., to make it less verbose or for integration with, e.g., GUI IDEs and continuous integration servers.

5.5.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#dead-code-scanner

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.5.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(dead_code_scanner(loader)).
```

5.5.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(dead_code_scanner(tester)).
```

5.5.4 Usage

This tool provides a set of predicates that allows scanning entities, libraries, files, and directories. See the tool API documentation for details. The source code to be analyzed should be loaded with the `source_data` and `optimize` flags turned on (possibly set from a loader file).

As an example, assume that we want to scan an application with a library alias `my_app`. The following goals could be used:

```
| ?- set_logtalk_flag(source_data, on),
     set_logtalk_flag(optimize, on).
yes

| ?- logtalk_load(my_app(loader)).
...
yes

| ?- dead_code_scanner::library(my_app).
...
```

For complex applications that make use of sub-libraries, there are also `rlibrary/1-2` predicates that perform a recursive scan of a library and all its sub-libraries. Conversely, we may be interested in scanning a single entity:

```
| ?- dead_code_scanner::entity(some_object).
...
```

For other usage examples, see the `SCRIPT.txt` file in the tool directory.

5.5.5 Excluding code from analysis

A set of options is available to specify code that should be excluded when looking for unused predicates (and non-terminals):

- `exclude_directories(Directories)`
  list of directories to exclude (default is `[]`); all sub-directories of the excluded directories are also excluded; directories may be listed by full or relative path
- `exclude_files(Files)`
  list of source files to exclude (default is `[]`); files may be listed by full path or basename, with or without extension
- `exclude_libraries(Libraries)`
  list of libraries to exclude (default is `[startup, scratch_directory]`)
- `exclude_entities(Entities)`
  list of entities to exclude (default is `[]`)

5.5.6 Integration with the `make` tool

The `loader.lgt` file sets a make target action that will call the `dead_code_scanner::all` goal whenever the `logtalk_make(check)` goal (or its top-level abbreviation, `{?}`) is called.

5.5.7 Caveats

Use of local meta-calls with goal arguments only known at runtime can result in false positives. When using library or user-defined meta-predicates, compilation of the source files with the `optimize` flag turned on may allow meta-calls to be resolved at compile-time and thus allow calling information for the meta-arguments to be recorded, avoiding false positives for predicates that are only meta-called.

5.5.8 Scanning Prolog modules

This tool can also be applied to Prolog modules that Logtalk is able to compile as objects. For example, if the Prolog module file is named `module.pl`, try:

```
| ?- logtalk_load(module, [source_data(on)]).
```

Due to the lack of standardization of module systems and the abundance of proprietary extensions, this solution is not expected to work for all cases.

5.5.9 Scanning plain Prolog files

This tool can also be applied to plain Prolog code. For example, if the Prolog file is named `code.pl`, simply define an object including its code:

```
:- object(code).
    :- include('code.pl').
:- end_object.
```

Save the object to an e.g. `code.lgt` file in the same directory as the Prolog file and then load it in debug mode:

```
| ?- logtalk_load(code, [source_data(on), optimize(on)]).
```

In alternative, use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(code, [hook(object_wrapper_hook), source_data(on), optimize(on)]).
```

With either wrapping solution, pay special attention to any compilation warnings that may signal issues that could prevent the plain Prolog from being fully analyzed when wrapped by an object.

5.6 debug_messages

By default, `debug` and `debug(Group)` messages are only printed when the `debug` flag is turned on. These messages are also suppressed when compiling code with the `optimize` flag turned on. This tool supports selective enabling of `debug` and `debug(Group)` messages in normal and debug modes.

5.6.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#debug-messages

For general information on debugging, open in a web browser the following link and consult the debugging section of the User Manual:

../../handbook/userman/debugging.html

5.6.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(debug_messages(loader)).
```

5.6.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(debug_messages(tester)).
```

5.6.4 Usage

The tool provides two sets of predicates. The first set allows enabling and disabling of all `debug` and `debug(Group)` messages for a given component. The second set allows enabling and disabling of `debug(Group)` messages for a given group and component for fine-grained control.

Upon loading the tool, all debug messages are skipped. The user is then expected to use the tool API to selectively enable the messages that will be printed. As an example, consider the following object, part of a `xyz` component:

```
:- object(foo).

    :- public([bar/0, baz/0]).
    :- uses(logtalk, [print_message/3]).

    bar :-
        print_message(debug(bar), xyz, @'bar/0 called').

    baz :-
        print_message(debug(baz), xyz, @'baz/0 called').

:- end_object.
```

Assuming the object `foo` is compiled and loaded in normal or debug mode, after also loading this tool, `bar/0` and `baz/0` messages will not print any debug messages:

```
| ?- {debug_messages(loader), foo}.
...
yes

| ?- foo::(bar, baz).
yes
```

We can then enable all debug messages for the `xyz` component:

```
| ?- debug_messages::enable(xyx).
yes

| ?- foo::(bar, baz).
bar/0 called
baz/0 called
yes
```

Or we can selectively enable only debug messages for a specific group:

```
| ?- debug_messages::disable(xyx).
yes

| ?- debug_messages::enable(xyx, bar).
yes

| ?- foo::(bar, baz).
bar/0 called
yes
```

5.7 debugger

This tool provides the default Logtalk command-line debugger. Unlike Prolog systems, the Logtalk debugger is a regular application, using a public API. As a consequence, it must be explicitly loaded by the programmer, either manually at the top-level interpreter or automatically from a settings file.

5.7.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#debugger

5.7.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(debugger(loader)).
```

When the code to be debugged runs computationally expensive initializations, loading this tool after the code may have a noticeable impact on loading times.

5.7.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(debugger(tester)).
```

5.7.4 Usage

Debugging Logtalk source code (with this debugger) requires compiling source files using the `debug(on)` compiler flag. For example:

```
| ?- logtalk_load(my_buggy_code, [debug(on)]).
```

In alternative, you may also turn on the `debug` flag globally by typing:

```
| ?- set_logtalk_flag(debug, on).
```

But note that loader files may override this flag setting (e.g., by using `debug(off)` or `optimize(on)` options for loaded files). If that's the case, you will need to either edit the loader files or write customized loader files enabling debugging. For detailed information on using the debugger, consult the debugging section of the User Manual:

../../handbook/userman/debugging.html

The `debugger_messages.lgt` source file defines the default debugger message translations.

The `dump_trace.lgt` provides a simple solution for dumping a goal trace to a file. For example:

```
| ?- dump_trace::start_redirect_to_file('trace.txt', some_goal),
     dump_trace::stop_redirect_to_file.
```

A full trace can also be obtained at the top-level by using the `S` (Skip) command at the call port for the top-level goal when tracing it.

5.7.5 Alternative debugger tools

Logtalk provides basic support for the SWI-Prolog graphical tracer. The **required** settings are described in the `samples/settings-sample.lgt` file. Logtalk queries can be traced using this tool by using the `gtrace/0-1` predicates. For example:

```
| ?- gtrace(foo::bar).
```

Or alternatively:

```
| ?- gtrace, foo::bar.
```

You can also use the `gspy/1` predicate to spy on a Logtalk predicate specified as `Entity::Functor/Arity` when using the graphical tracer. When using this tool, internal Logtalk compiler/runtime predicates and compiled predicates that resulted from the term-expansion mechanism may be exposed in some cases. This issue is shared with Prolog code and results from the non-availability of source code for the predicates being traced.

5.7.6 Known issues

Clause breakpoints require a Prolog backend compiler that supports accessing read term starting line but only some backends (B-Prolog, GNU Prolog, JIProlog, XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, and YAP) provide accurate line numbers. As a workaround, you can check the start line number for an entity predicate definition using a query such as:

```
| ?- object_property(Entity, defines(Functor/Arity, Properties)).
```

Check the returned `line_count/1` property to find if there's any offset to the source file number of the predicate clause that you want to trace. This issue, if present, usually only affects the first predicate clause.

Clause breakpoints are currently not available when using XSB as this backend doesn't provide line information.

Using the port command `p` (print) requires a backend supporting the user-defined `portray/1` hook predicate called via the `format/2-3` predicates `~p` control sequence.

5.8 `diagrams`

This tool generates *library*, *directory*, *file*, *entity*, and *predicate* diagrams for source files and for libraries of source files using the Logtalk reflection API to collect the relevant information and a graph language for representing the diagrams. Limited support is also available for generating diagrams for Prolog module applications. It's also possible in general to generate predicate cross-referencing diagrams for plain Prolog files.

Linking *library diagrams* to *entity diagrams* to *predicate cross-referencing diagrams* and linking *directory diagrams* to *file diagrams* is also supported when using SVG output. This feature allows using diagrams for understanding the architecture of applications by navigating complex code and zooming into details. SVG output can also easily link to both source code repositories and API documentation. This allows diagrams to be used for source code navigation.

Diagrams can also be used to uncover code issues. For example, comparing *loading diagrams* with *dependency diagrams* can reveal implicit dependencies. Loading diagrams can reveal circular dependencies that may warrant code refactoring. Entity diagrams can provide a good overview of code coupling. Predicate cross-referencing diagrams can be used to visually access entity code complexity, complementing the `code_metrics` tool.

All diagrams support a comprehensive set of options, discussed below, to customize the final contents and appearance.

Diagram generation can be easily automated using the `doclet` tool and the `logtalk_doclet` scripts. See the `doclet` tool examples and documentation for details. See also the `diagrams` tool own `lgt2svg` Bash and PowerShell scripts.

5.8.1 Requirements

Recent versions of d2 (0.6.9 or later) or Graphviz are required for generating diagrams in the final formats:

- https://d2lang.com
- https://www.graphviz.org/

These can be installed using their own installers or using operating-system package managers:

macOS - MacPorts

```
$ sudo port install d2 graphviz
```

macOS - Homebrew

```
$ brew install d2 graphviz
```

Windows - Chocolatey

```
> choco install d2 graphviz
```

Installers

- https://d2lang.com/tour/install
- https://www.graphviz.org/download/

5.8.2 API documentation

This tool API documentation is available at:

../../apis/library_index.html#diagrams

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.8.3 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(diagrams(loader)).
```

5.8.4 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(diagrams(tester)).
```

5.8.5 Supported diagrams

The following entity diagrams are supported:

- *entity diagrams* showing entity public interfaces, entity inheritance relations, and entity predicate cross-reference relations
- *predicate cross-reference diagrams* (between entities or within an entity)
- *inheritance diagrams* showing entity inheritance relations
- *uses diagrams* showing which entities use resources from other entities

The following library diagrams are supported:

- *library loading diagrams* showing which libraries load other libraries
- *library dependency diagrams* showing which libraries contain entities with references to entities defined in other libraries

The following file diagrams are supported:

- *file loading diagrams* showing which files load or include other files
- *file dependency diagrams* showing which files contain entities with references to entities defined in other files

File dependency diagrams are specially useful in revealing dependencies that are not represented in file loading diagrams due to files being loaded indirectly by files external to the libraries being documented.

The following directory diagrams are supported:

- *directory loading diagrams* showing which directories contain files that load files from other directories
- *directory dependency diagrams* showing which directories contain entities with references to entities defined in other directories

Comparing directory (or file) loading diagrams with directory (or file) dependency diagrams allows comparing what is explicitly loaded with the actual directory (or file) dependencies, which are inferred from the source code.

Library and directory dependency diagrams are specially useful for large applications where file diagrams would be too large and complex to be useful, notably when combined with the *zoom* option to link to, respectively, entity and file diagrams.

A utility object, `diagrams`, is provided for generating all supported diagrams in one step. This object provides an interface common to all diagrams, but note that some predicates that generate diagrams only make sense for some types of diagrams. For best results and fine-grained customization of each diagram, the individual diagram objects should be used with the intended set of options.

5.8.6 Graph elements

Limitations in both the graph languages and UML force the invention of a modeling language that can represent all kinds of Logtalk entities and entity relations. Currently we use the following Graphviz node shapes (libraries, entities, predicates, and files) and arrowheads (entity, predicate, and file relations):

- libraries
  `tab` (lightsalmon)
- library loading and dependency relations
  `normal` (arrow ending with a black triangle)
- objects (classes, instances, and prototypes)
  `box` (rectangle, yellow for instances/classes and beige for prototypes)
- protocols
  `note` (aqua marine rectangle with folded right-upper corners)
- categories
  `component` (light cyan rectangle with two small rectangles intercepting the left side)
- modules
  `box` (plum rectangle with small tab at top)
- public predicates
  `box` (springgreen)
- public, multifile, predicates
  `box` (skyblue)
- protected predicates
  `box` (yellow)
- private predicates
  `box` (indianred)
- external predicates
  `box` (beige)
- exported module predicates
  `box` (springgreen)
- directories
  `tab` (lightsalmon)
- directory loading and dependency relations

`normal` (arrow ending with a black triangle)

- files
  `box` (pale turquoise rectangle)
- file loading and dependency relations
  `normal` (arrow ending with a black triangle)
- specialization relation
  `onormal` (arrow ending with a white triangle)
- instantiation relation
  `normal` (arrow ending with a black triangle)
- extends relation
  `vee` (arrow ending with a "v")
- implements relation
  `dot` (arrow ending with a black circle)
- imports relation
  `box` (arrow ending with a black square)
- complements relation
  `obox` (arrow ending with a white square)
- uses and use module relations
  `rdiamond` (arrow ending with a black half diamond)
- predicate calls
  `normal` (arrow ending with a black triangle)
- dynamic predicate updates
  `diamond` (arrow ending with a black diamond)

When using the d2 graph language, we use similar node shapes and arrowheads when available. As d2 evolves, we hope that these graph elements will converge further.

The library, directory, file, entity, and predicate nodes that are not part of the predicates, entities, files, or libraries for which we are generating a diagram use a dashed border, a darker color, and are described as external.

Note that all the elements above can have captions. See below the diagrams `node_type_captions/1` and `relation_labels/1` output options.

5.8.7 Supported graph languages

Currently, both DOT and d2 graph languages support all the features of the `diagrams` tool. There's also preliminary support for Mermaid (not loaded by default, as its current version lacks several required features for parity with d2 and Graphviz).

The diagrams `.d2` and `.dot` files are created in the current directory by default. These files can be easily converted into a printable format such as SVG, PDF, or Postscript. Sample helper scripts are provided for converting a directory of `.d2` or `.dot` files to `.svg` files:

- `lgt2svg.sh` for POSIX systems
- `lgt2svg.ps1` for Windows systems

The scripts assume that the d2 and Graphviz command-line executables are available from the system path. For Graphviz, the default is the `dot` executable, but the scripts accept a command-line option to select in alternative the `circo`, `fdp`, or `neato` executables). For d2, the default layout engine is `elk`, but the scripts accept a command-line option to select in alternative the `dagre` or `tala` layout engines.

The recommended output format is SVG, as it supports tooltips and URL links, which can be used for showing, e.g., entity types, relation types, file paths, and for navigating to files and directories of files (libraries) or to API documentation. See the relevant diagram options below in order to take advantage of these features (see the discussion below on "linking diagrams").

To convert to formats other than SVG, you will need to use the d2 and Graphviz executables directly. For example, using the Graphviz `dot` executable, we can generate a PDF with the command:

```
dot -Tpdf diagram.dot > diagram.pdf
```

This usually works fine for entity and predicate call cross-referencing diagrams. For directory and file diagrams, the `fdp` and `circo` command-line executables may produce better results. For example:

```
fdp -Tsvg diagram.dot > diagram.svg
circo -Tsvg diagram.dot > diagram.svg
```

It's also worth experimenting with different layouts to find the one that produces the best results (see the `layout/1` option described below).

When generating diagrams for multiple libraries or directories, it's possible to split a diagram with several disconnected library or directory graphs using the `ccomps` command-line executable. For example:

```
ccomps -x -o subdiagram.dot diagram.dot
```

For more information on the DOT language and related tools, see:

```
http://www.graphviz.org/
```

When using Windows, there are known issues with some Prolog compilers due to the internal representation of paths. If you encounter problems with a specific backend Prolog compiler, try, if possible, to use another supported backend Prolog compiler when generating diagrams.

For printing large diagrams, you will need to either use a tool to slice the diagram into page-sized pieces or, preferably, use software capable of tiled printing (e.g., Adobe Reader). You can also hand-edit the generated `.dot` files and play with settings such as aspect ratio for fine-tuning the diagrams layout.

5.8.8 Customization

A set of options is available to specify the details to include in the generated diagrams. For entity diagrams, the options are:

- `layout(Layout)`
  diagram layout (one of the atoms `{top_to_bottom,bottom_to_top,left_to_right,right_to_left}`; default is `bottom_to_top`)
- `title(Title)`
  diagram title (an atom; default is `''`)
- `date(Boolean)`
  print current date and time (`true` or `false`; default is `true`)
- `versions(Boolean)`

print Logtalk and backend version data (`true` or `false`; default is `false`)

- `interface(Boolean)`
  print public predicates (`true` or `false`; default is `true`)
- `file_labels(Boolean)`
  print file labels (`true` or `false`; default is `true`)
- `file_extensions(Boolean)`
  print file name extensions (`true` or `false`; default is `true`)
- `relation_labels(Boolean)`
  print entity relation labels (`true` or `false`; default is `true`)
- `externals(Boolean)`
  print external nodes (`true` or `false`; default is `true`)
- `node_type_captions(Boolean)`
  print node type captions (`true` or `false`; default is `true`)
- `inheritance_relations(Boolean)`
  print inheritance relations (`true` or `false`; default is `true` for entity inheritance diagrams and `false` for other entity diagrams)
- `provide_relations(Boolean)`
  print provide relations (`true` or `false`; default is `false`)
- `xref_relations(Boolean)`
  print predicate call cross-reference relations (`true` or `false`; default depends on the specific diagram)
- `xref_calls(Boolean)`
  print predicate cross-reference calls (`true` or `false`; default depends on the specific diagram)
- `output_directory(Directory)`
  directory for the `.d2` and `.dot` files (an atom; default is `'./dot_dias'`)
- `exclude_directories(Directories)`
  list of directories to exclude except as external nodes (default is `[]`); all sub-directories of the excluded directories are also excluded; directories may be listed by full or relative path
- `exclude_files(Files)`
  list of source files to exclude except as external nodes (default is `[]`); files may be listed by full path or basename, with or without extension
- `exclude_libraries(Libraries)`
  list of libraries to exclude except as external nodes (default is `[startup, scratch_directory]`)
- `exclude_entities(Entities)`
  list of entities to exclude except as external nodes (default is `[]`)
- `path_url_prefixes(PathPrefix, CodeURLPrefix, DocURLPrefix)`
  code and documenting URL prefixes for a path prefix used when generating cluster, library, directory, file, and entity links (atoms; no default; can be specified multiple times)
- `url_prefixes(CodeURLPrefix, DocURLPrefix)`
  default URL code and documenting URL prefixes used when generating cluster, library, file, and entity links (atoms; no default)
- `entity_url_suffix_target(Suffix, Target)`

extension for entity documenting URLs (an atom; default is `'.html'`) and target separating symbols (an atom; default is `'#'`)

- `omit_path_prefixes(Prefixes)`
  omit common path prefixes when printing directory paths and when constructing URLs (a list of atoms; default is a list with the user home directory)
- `zoom(Boolean)`
  generate sub-diagrams and add links and zoom icons to library and entity nodes (`true` or `false`; default is `false`)
- `zoom_url_suffix(Suffix)`
  extension for linked diagrams (an atom; default is `'.svg'`)

In the particular case of cross-referencing diagrams, there are also the options:

- `recursive_relations(Boolean)`
  print recursive predicate relations (`true` or `false`; default is `false`)
- `url_line_references(Host)`
  syntax for the URL source file line part (an atom; possible values are `{github,gitlab,bitbucket}`; default is `github`); when using this option, the `CodeURLPrefix` should be a permanent link (i.e., it should include the commit SHA1)
- `predicate_url_target_format(Generator)`
  documentation final format generator (an atom; default is `sphinx`)

For directory and file diagrams, the options are:

- `layout(Layout)`
  diagram layout (one of the atoms `{top_to_bottom,bottom_to_top,left_to_right,right_to_left}`; default is `top_to_bottom`)
- `title(Title)`
  diagram title (an atom; default is `''`)
- `date(Boolean)`
  print current date and time (`true` or `false`; default is `true`)
- `versions(Boolean)`
  print Logtalk and backend version data (`true` or `false`; default is `false`)
- `directory_paths(Boolean)`
  print file directory paths (`true` or `false`; default is `false`)
- `file_extensions(Boolean)`
  print file name extensions (`true` or `false`; default is `true`)
- `path_url_prefixes(PathPrefix, CodeURLPrefix, DocURLPrefix)`
  code and documenting URL prefixes for a path prefix used when generating cluster, directory, file, and entity links (atoms; no default; can be specified multiple times)
- `url_prefixes(CodeURLPrefix, DocURLPrefix)`
  default URL code and documenting URL prefixes used when generating cluster, library, file, and entity links (atoms; no default)
- `omit_path_prefixes(Prefixes)`
  omit common path prefixes when printing directory paths and when constructing URLs (a list of atoms; default is a list with the user home directory)
- `relation_labels(Boolean)`

print entity relation labels (`true` or `false`; default is `false`)

- `externals(Boolean)`
  print external nodes (`true` or `false`; default is `true`)
- `node_type_captions(Boolean)`
  print node type captions (`true` or `false`; default is `false`)
- `output_directory(Directory)`
  directory for the `.d2` and `.dot` files (an atom; default is `'./dot_dias'`)
- `exclude_directories(Directories)`
  list of directories to exclude except as external nodes (default is `[]`)
- `exclude_files(Files)`
  list of source files to exclude except as external nodes (default is `[]`)
- `zoom(Boolean)`
  generate sub-diagrams and add links and zoom icons to library and entity nodes (`true` or `false`; default is `false`)
- `zoom_url_suffix(Suffix)`
  extension for linked diagrams (an atom; default is `'.svg'`)

For library diagrams, the options are:

- `layout(Layout)`
  diagram layout (one of the atoms `{top_to_bottom,bottom_to_top,left_to_right,right_to_left}`; default is `top_to_bottom`)
- `title(Title)`
  diagram title (an atom; default is `''`)
- `date(Boolean)`
  print current date and time (`true` or `false`; default is `true`)
- `versions(Boolean)`
  print Logtalk and backend version data (`true` or `false`; default is `false`)
- `directory_paths(Boolean)`
  print file directory paths (`true` or `false`; default is `false`)
- `path_url_prefixes(PathPrefix, CodeURLPrefix, DocURLPrefix)`
  code and documenting URL prefixes for a path prefix used when generating cluster, library, file, and entity links (atoms; no default; can be specified multiple times)
- `url_prefixes(CodeURLPrefix, DocURLPrefix)`
  default URL code and documenting URL prefixes used when generating cluster, library, file, and entity links (atoms; no default)
- `omit_path_prefixes(Prefixes)`
  omit common path prefixes when printing directory paths and when constructing URLs (a list of atoms; default is a list with the user home directory)
- `relation_labels(Boolean)`
  print entity relation labels (`true` or `false`; default is `false`)
- `externals(Boolean)`
  print external nodes (`true` or `false`; default is `true`)
- `node_type_captions(Boolean)`

print node type captions (`true` or `false`; default is `false`)

- `output_directory(Directory)`
  directory for the `.d2` and `.dot` files (an atom; default is `'./dot_dias'`)
- `exclude_directories(Directories)`
  list of directories to exclude except as external nodes (default is `[]`)
- `exclude_files(Files)`
  list of source files to exclude except as external nodes (default is `[]`)
- `exclude_libraries(Libraries)`
  list of libraries to exclude except as external nodes (default is `[startup, scratch_directory]`)
- `zoom(Boolean)`
  generate sub-diagrams and add links and zoom icons to library and entity nodes (`true` or `false`; default is `false`)
- `zoom_url_suffix(Suffix)`
  extension for linked diagrams (an atom; default is `'.svg'`)

When using the `zoom(true)` option, the `layout(Layout)` option applies only to the top diagram; sub-diagrams will use their own layout default.

The option `omit_path_prefixes(Prefixes)` with a non-empty list of prefixes should preferably be used together with the option `directory_paths(true)` when generating library or file diagrams that reference external libraries or files. To confirm the exact default options used by each type of diagram, send the `default_options/1` message to the diagram object.

Be sure to set the `source_data` flag `on` before compiling the libraries or files for which you want to generate diagrams.

Support for displaying Prolog modules and Prolog module files in diagrams of Logtalk applications:

- ECLiPSe
  file diagrams don't display module files
- SICStus Prolog
  file diagrams don't display module files
- SWI-Prolog
  full support (uses the SWI-Prolog `prolog_xref` library)
- YAP
  full support (uses the YAP `prolog_xref` library)

5.8.9 Linking diagrams

When using SVG output, it's possible to generate diagrams that link to other diagrams, to API documentation, to local files and directories, and to source code repositories.

For generating links between diagrams, use the `zoom(true)` option. This option allows (1) linking library diagrams to entity diagrams to predicate cross-referencing diagrams and (2) linking directory diagrams to file diagrams to entity diagrams to predicate cross-referencing diagrams. The sub-diagrams are automatically generated. For example, using the predicates that generate library diagrams will also automatically generate the entity and predicate cross-referencing diagrams.

To generate local links for opening directories, files, and file locations in selected text editors, set the URL code prefix:

- VSCode: `url_prefixes('vscode://file/', DocPrefix)`
- VSCodium: `url_prefixes('vscodium://file/', DocPrefix)`
- Cursor: `url_prefixes('cursor://file/', DocPrefix)`
- PearAI: `url_prefixes('pearai://file/', DocPrefix)`
- Windsurf: `url_prefixes('windsurf://file/', DocPrefix)`
- Zed: `url_prefixes('zed://file/', DocPrefix)`
- BBEdit: `url_prefixes('x-bbedit://open?url=file://', DocPrefix)`
- MacVim: `url_prefixes('mvim://open?url=file://', DocPrefix)`
- TextMate: `url_prefixes('txmt://open?url=file://', DocPrefix)`
- IDEA: `url_prefixes('idea://open?file=', DocPrefix)`
- PyCharm: `url_prefixes('pycharm://open?file=', DocPrefix)`

In this case, the `DocPrefix` argument should be the path to the directory containing the HTML version of the application APIs.

As most of the text editor URL scheme handlers require local links to use absolute paths, the `omit_path_prefixes/1` option is ignored. Note that local links require text editor support for URL schemes that can handle both file and directory links.

To generate links to API documentation and source code repositories, use the options `path_url_prefixes/3` (or `url_prefixes/2` for simpler cases) and `omit_path_prefixes/1`. The idea is that the `omit_path_prefixes/1` option specifies local file prefixes that will be cut and replaced by the URL prefixes (which can be path prefix specific when addressing multiple code repositories). To generate local file system URLs, define the empty atom, `''`, as a prefix. As an example, consider the Logtalk library. Its source code is available from a GitHub repository, and its documentation is published on the Logtalk website. The relevant URLs in this case are:

- https://github.com/LogtalkDotOrg/logtalk3/ (source code)
- https://logtalk.org/library/ (API documentation)

Git source code URLs should include the commit SHA1 to ensure that entity and predicate file line information in the URLs remain valid if the code changes in later commits. Assuming a `GitHub` variable bound to the SHA1 commit URL we want to reference, an inheritance diagram can be generated using the goal:

```
| ?- GitHub  = 'https://github.com/LogtalkDotOrg/logtalk3/commit/
↪eb156d46e135ac47ef23adcc5d20d49dd8b66abb',
     APIDocs = 'https://logtalk.org/library/',
     logtalk_load(diagrams(loader)),
     set_logtalk_flag(source_data, on),
     logtalk_load(library(all_loader)),
     inheritance_diagram::rlibrary(library, [
         title('Logtalk library'),
         node_type_captions(true),
         zoom(true),
         path_url_prefixes('$LOGTALKUSER/', GitHub, APIDocs),
         path_url_prefixes('$LOGTALKHOME/', GitHub, APIDocs),
         omit_path_prefixes(['$LOGTALKUSER/', '$LOGTALKHOME/', '$HOME/'])
     ]).
```

The two `path_url_prefixes/3` options take care of source code and API documentation for entities loaded either from the Logtalk installation directory (whose location is given by the `LOGTALKHOME` environment

variable) or from the Logtalk user directory (whose location is given by the `LOGTALKUSER` environment variable). As we also don't want any local operating-system paths to be exposed in the diagram, we use the `omit_path_prefixes/1` option to suppress those path prefixes. Note that all the paths and URLs must end with a slash for proper handling. The `git` library may be useful to retrieve the commit SHA1 from a local repo directory.

For both `path_url_prefixes/3` and `omit_path_prefixes/1` options, when a path prefix is itself a prefix of another path, the shorter path must come last to ensure correct links.

See the `SCRIPT.txt` file in the tool directory for additional examples. To avoid retyping such complex goals when updating diagrams, use the `doclet` tool to save and reapply them easily (e.g., by using the `make` tool with the `documentation` target).

5.8.10 Creating diagrams for Prolog module applications

Currently limited to SWI-Prolog and YAP Prolog module applications due to the lack of a comprehensive reflection API in other Prolog systems.

Simply load your Prolog module application and its dependencies and then use diagram entity, directory, or file predicates. Library diagram predicates are not supported. See the `SCRIPT.txt` file in the tool directory for some usage examples. Note that support for diagrams with links to API documentation is quite limited, however, due to the lack of Prolog standards.

5.8.11 Creating diagrams for plain Prolog files

This tool can also be used to create predicate cross-referencing diagrams for plain Prolog files. For example, if the Prolog file is named `code.pl`, simply define an object including its code:

```
:- object(code).
    :- include('code.pl').
:- end_object.
```

Save the object to an e.g. `code.lgt` file in the same directory as the Prolog file and then load it and create the diagram:

```
| ?- logtalk_load(code),
     xref_diagram::entity(code).
```

An alternative is to use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(code, [hook(object_wrapper_hook)]),
     xref_diagram::entity(code).
```

5.8.12 Other notes

Generating complete diagrams requires that all referenced entities are loaded. When that is not the case, notably when generating cross-referencing diagrams, missing entities can result in incomplete diagrams.

For complex applications, diagrams can often be made simpler and more readable by omitting external nodes (see the `externals/1` option) and/or using one of the alternatives to `dot` provided by Graphviz depending on the type of the diagram (see the section above on supported graph languages for more details).

When generating entity predicate call cross-reference diagrams, caller nodes are not created for auxiliary predicates. For example, if the `meta_compiler` library is used to optimize meta-predicate calls, the diagrams may show predicates that are not apparently called by any other predicate when the callers are from the optimized meta-predicate goals (which are called via library generated auxiliary predicates). A workaround in this case would be creating a dedicated loader file that doesn't load (and apply) the `meta_compiler` library when generating the diagrams.

When generating diagrams in SVG format, a copy of the `diagrams.css` file must exist in any directory used for publishing diagrams using it. The `lgt2svg` scripts also take care of copying this file.

The Graphviz command-line utilities, e.g., `dot`, are notorious for random crashes (segmentation faults usually), often requiring retrying conversions from `.dot` files to other formats. A possible workaround is to repeat the command until it completes without error. See, for example, the `lgt2svg.sh` script.

The conversion by the `d2` command-line executable of `.d2` files to `.svg` files can be quite slow (as of its 0.6.8 version) with the default `elk` layout engine. The `dagre` layout engine is much faster but doesn't support a node referencing itself (notably, a node representing a metaclass that instantiates itself).

Using the default d2 layout engine (`elk`) works fine with graphs with a relatively small number of nodes and edges. When that's not the case, it's a good idea to experiment with other layout engines.

5.9 `doclet`

This folder provides a simple tool for (re)generating documentation for a project. The tool defines a `doclet` object that is expected to be extended by the user to specify a sequence of goals and a sequence of shell commands that load the application and (re)generate its documentation.

Doclet source files are preferably named `doclet.lgt` (or `doclet.logtalk`) and the doclet objects are usually named after the application or library to be documented with a `_doclet` suffix. By using an `initialization/1` directive to automatically send the `update/0` message that generates the documentation upon doclet loading, we can abstract the name of the doclet object. The usual query to load and run a doclet is therefore:

```
| ?- logtalk_load([doclet(loader), doclet]).
```

For usage examples, see the `sample_doclet.lgt`, `doclet1.lgt`, `zoom_doclet.lgt`, and `tools_doclet.lgt` source files.

5.9.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#doclet

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.9.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(doclet(loader)).
```

5.9.3 Automating running doclets

You can use the `scripts/logtalk_doclet.sh` Bash shell script for automating running doclets. The script expects the doclet source files to be named either `doclet.lgt` or `doclet.logtalk`. See the `scripts/NOTES.md` file or the script man page for details.

5.9.4 Integration with the make tool

Loading this tool adds a definition for the `logtalk_make_target_action/1` hook predicate for the target `documentation`. The hook definition sends an `update/0` message to each loaded doclet.

5.10 `help`

This tool provides basic online help for Logtalk features and libraries when running in most operating-systems. For help on the Logtalk compiler error and warning messages, see the `tutor` tool.

5.10.1 Requirements

On Windows, the `start` command must be available. On Linux, the `xdg-open` command must be available. On macOS, the command `open` is used.

Experimental features for browsing the Handbook and APIs documentation at the top-level require Texinfo to be installed. See the `tools/NOTES.md` file for per operating-system installation instructions.

5.10.2 API documentation

This tool API documentation is available at:

../../apis/library_index.html#help

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.10.3 Loading

```
| ?- logtalk_load(help(loader)).
```

5.10.4 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(help(tester)).
```

5.10.5 Supported operating-systems

Currently, support is limited to Linux, macOS, and Windows.

This tool relies on the library portable operating-system access abstraction.

5.10.6 Usage

After loading the tool, use the query `help::help` to get started.

5.10.7 Experimental features

On POSIX systems, when using Ciao Prolog, ECLiPSe, GNU Prolog (1.5.1 or later version), XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, or XSB as the backend, `apis/1` and `handbook/0-1` predicates are made available. These predicates open inline at the top-level interpreter the Texinfo versions of the Handbook and the APIs documentation. The optional argument is a starting node, which can be an atom, a predicate indicator, or a non-terminal indicator. When there are several nodes for the same argument (e.g., multiple implementations of the `member/2` predicate), one of them will be displayed. Some examples:

```
| ?- help::handbook.

| ?- help::handbook(base64).

| ?- help::handbook(logtalk_load/2).

| ?- help::apis.

| ?- help::apis(check/2).

| ?- help::apis(message_tokens//2).
```

Although less useful, you can also browse the `man` pages of Logtalk scripts. For example:

```
| ?- help::man(logtalk_tester).
```

When you finish consulting the documentation and quit the `info` process, you will be back to the top-level prompt (if you find that the top-level have scrolled from its last position, try to set your terminal terminfo to `xterm-256colour`).

If you're running Logtalk from a git clone of its repo, you will need to run the `docs/apis/sources/build.sh` or `docs/apis/sources/build.ps1` scripts to generate the APIs documentation `.info` file and also run the

`docs/handbook/sources/build.sh` or `docs/handbook/sources/build.ps1` scripts to generate the Handbook `.info` file. Alternatively, you can download the `.info` files for the latest stable release from the Logtalk website and save them to the `docs` and `manuals` directories.

The required `info` command is provided by the third-party `texinfo` package (tested with version 6.8). On macOS, this package can be installed with either MacPorts:

```
$ sudo port install texinfo
```

Or using Homebrew:

```
$ brew install texinfo
```

On Linux systems, use the distribution's own package manager to install the `texinfo` package. For example, in Ubuntu systems:

```
$ sudo apt install info
```

5.10.8 Known issues

The open commands used to open documentation URLs drops the fragment part, thus preventing navigation to the specified position on the documentation page.

When browsing the Texinfo versions of the Handbook and the APIs documentation generated with a recent version of Sphinx and using a recent version of Texinfo, the Texinfo search feature often displays the previous nodes of the searched nodes.

ECLiPSe defines a `help` prefix operator that forces wrapping this atom between parentheses when sending messages to the tool. E.g. use `(help)::help` instead of `help::help`.

5.11 `issue_creator`

This is a complementary tool for the `lgtunit` tool for automatically creating bug report issues for failed tests in GitHub or GitLab servers.

5.11.1 Requirements

This tool requires that the GitHub `gh` and GitLab `glab` CLIs be installed. For the installation instructions see:

- GitHub: https://cli.github.com
- GitLab: https://glab.readthedocs.io

5.11.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(issue_creator(loader)).
```

But, in the most common usage scenario, this tool is automatically loaded by the `logtalk_tester` automation script.

5.11.3 Usage

The `logtalk_tester` automation script accepts a `-b` option for automatically using this tool (see the script man page for details). In the most simple case, this option possible values are `github` and `gitlab`. For example:

```
$ logtalk_tester \
    -p gnu \
    -b github \
    -s "/home/jdoe/foo/" \
    -u https://github.com/jdoe/foo/tree/55aa900775befa135e0d5b48ea63098df8b97f5c/
```

The `logtalk_tester` script **must** be called from a git repo directory or one of its sub-directories, which is a common setup in CI/CD pipelines. Moreover, prior to running the tests, the CLI must be used, if required, to authenticate and login to the server where the bug report issues will be created:

- GitHub: `gh auth login --hostname <string> --with-token < token.txt`
- GitLab: `glab auth login --hostname <string> --token <string>`

The access token must have the necessary scopes that allow bug reports to be created. See the CLIs documentation for details. Typically, the `auth` command is called from the CI/CD pipeline definition scripts. However, depending on the CI/CD workflow, the authentication may be done implicitly.

The bug reports are created using by default the label `bug` and assigned to the author of the latest commit of the git repo. The `-b` option can also be used to override the label with a comma separated set of labels. For example, to use both `bug` and `auto` labels:

```
$ logtalk_tester \
    -p gnu \
    -b github:bug,auto \
    -s "/home/jdoe/foo/" \
    -u https://github.com/jdoe/foo/tree/55aa900775befa135e0d5b48ea63098df8b97f5c/
```

Note that the labels **must** be predefined in the issue tracker server for the bug report to be successfully created. The `auto` label can be used to simplify filtering of auto-generated bug reports when browsing the issue tracker.

The bug reports use Markdown formatting, which is the default in GitHub and GitLab issue trackers.

But reports are only created for non-flaky tests. The bug report title and labels are used to prevent creating duplicated bug reports. Therefore, the title should not be manually edited and the same labels should be used for multiple runs of the same tests and preserved when editing the bug reports.

There are cases where we may want to postpone or temporarily disable the automatic creation of bug reports. E.g. a WIP branch that's known to break multiple tests. A solution is to define a pull/merge request label, e.g. `NO_AUTO_BUG_REPORTS`, that can then be checked by the CI/CD workflow. For example, we can test the

presence of that label to set a `AUTO_BUG_REPORTS` environment variable to either an empty string or a `-b` option and use:

```
logtalk_tester.sh $AUTO_BUG_REPORTS -p ...
```

5.11.4 Known issues

GitLab creates CI/CD pipelines in a detached HEAD state by default. As a consequence, the git branch would be reported as HEAD. To workaround this issue, this tool uses the value of the GitLab CI/CD pipeline variable `CI_COMMIT_REF_NAME` when defined as the branch name.

This tool is in a beta stage of development. Your feedback is most appreciated.

5.12 `lgtdoc`

This is the default Logtalk documenting tool for generating API documentation for libraries and applications. It uses the structural reflection API to extract and output in XML format relevant documentation about a source file, a library or directory of source files, or all loaded source files. The tool predicates accept several options for generating the XML files, including the output directory.

The `lgtdoc/xml` directory contains several ready-to-use Bash and PowerShell scripts for converting the XML documenting files into final formats, including (X)HTML, PDF, Markdown, and reStructuredText (for use with Sphinx), and plain text files. The scripts are described in their `man` pages and made available in the system path by default.

5.12.1 Requirements

This tool requirements for converting the XML files it generates to a final format are as follows:

1. Converting XML files to (X)HTML, reStructuredText, Markdown, or plain text files requires a XSLT processor. The supported XSLT processors for Bash conversion scripts are:

- xsltproc: https://gitlab.gnome.org/GNOME/libxslt/-/wikis/home
- Xalan: https://xml.apache.org/xalan-c/index.html
- Saxon: https://www.saxonica.com

On Windows, the PowerShell scripts use the .Net XSLT classes.

The reStructuredText files output is usually used as an intermediate step to generate Sphinx HTML, PDF, ePub, and Texinfo files. The additional requirements are:

- Sphinx: https://www.sphinx-doc.org/
- Pygments: https://pygments.org/
- Read the Docs theme: https://github.com/readthedocs/sphinx_rtd_theme

2. Converting XML files to PDF files require a XSL-FO processor. The supported XSL-FO processors for Bash and PowerShell conversion scripts are:

- Apache FOP processor: https://xmlgraphics.apache.org/fop/
- RenderX XEP processor: https://www.renderx.com/

For additional details, including compatible dependency versions and available conversion scripts, see the `xml/NOTES.md` file. See the `tools/NOTES.md` file for per operating-system installation instructions for the above dependencies.

5.12.2 API documentation

This tool API documentation is available at:

../../apis/library_index.html#lgtdoc

5.12.3 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(lgtdoc(loader)).
```

5.12.4 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(lgtdoc(tester)).
```

5.12.5 Documenting source code

For information on documenting your source code, notably on documenting directives, consult the documenting section of the User Manual:

../../handbook/userman/documenting.html

Extracting documenting information from your source code using this tool requires compiling the source files using the `source_data(on)` compiler flag. For example:

```
| ?- logtalk_load(source_file, [source_data(on)]).
```

Usually, this flag is set for all application source files in the corresponding loader file. In alternative, you may also turn on the `source_data` flag globally by typing:

```
| ?- set_logtalk_flag(source_data, on).
```

The tool API allows generating documentation for libraries, directories, and files, complemented with library, directory, entity, and predicate indexes. Note that the source files to be documented **must** be loaded prior to using this tool predicates to generate the documentation.

5.12.6 Generating documentation

For a simple application, assuming a library alias is defined for it (e.g. `my_app`), and at the top-level interpreter, we can generate the application documentation by typing:

```
| ?- {my_app(loader)}.
...

| ?- {lgtdoc(loader)}.
...

| ?- lgtdoc::library(my_app).
...
```

By default, the documenting XML files are created in a `xml_docs` directory in the current working directory. But usually all documenting files are collected for both the application and the libraries it uses in a common directory so that all documentation links resolved properly. The `lgtdoc` predicates can take a list of options to customize the generated XML documenting files. See the remarks section in the lgtdocp protocol documentation for details on the available options.

After generating the XML documenting files, these can be easily converted into final formats using the provided scripts. For example, assuming that we want to generate HTML documentation:

```
$ cd xml_docs
$ lgt2html -t "My app"
```

To generate the documentation in Sphinx format instead (as used by Logtalk itself for its APIs):

```
$ cd xml_docs
$ lgt2rst -s -- -q -p "Application name" -a "Author name" -v "Version X.YZ.P"
$ make html
```

In this case, the generated documentation will be in the `xml_docs/_build/html/` directory. See the scripts man pages or call them using the `-h` option to learn more about their supported options.

For more complex applications, you can use the `doclet` tool to define a *doclet* to automate all the steps required to generate documentation. The *doclet* message that triggers the process can also be sent automatically when the `make` tool is used with the `documentation` target.

5.12.7 Documentation linter checks

When the `lgtdoc_missing_directives` flag is set to `warning` (its usual default value), the `lgtdoc` tool prints warnings on missing entity `info/1` directives and missing predicate `info/2` and `mode/2` directives.

When the `lgtdoc_missing_info_key` flag is set to `warning` (its usual default value), the `lgtdoc` tool prints warnings on entity `info/1` directive and predicate `info/2` directive missing de facto required keys (e.g., `comment`, `parameters` or `parnames` for parametric entities, `arguments` or `argnames` for predicates/non-terminals with arguments).

When the `lgtdoc_invalid_dates` flag is set to `warning` (its usual default value), the `lgtdoc` tool prints warnings on invalid dates (including dates in the future) in `info/1` directives.

When the `lgtdoc_non_standard_exceptions` flag is set to `warning` (its usual default value), the `lgtdoc` tool prints warnings on non-standard exceptions. This linter check is particularly effective in detecting typos when specifying standard exceptions.

When the `lgtdoc_missing_punctuation` flag is set to `warning` (its usual default value), the `lgtdoc` tool prints warnings on missing ending periods (full stops), exclamation marks, or question marks in `info/1-2` directives (in comments, remarks, parameter descriptions, and argument descriptions).

Set a flag value to `silent` to turn off the corresponding linter warnings.

5.13 `lgtunit`

The `lgtunit` tool provides testing support for Logtalk. It can also be used for testing plain Prolog code and Prolog module code.

This tool is inspired by the xUnit frameworks architecture and by the works of Joachim Schimpf (ECLiPSe library `test_util`) and Jan Wielemaker (SWI-Prolog `plunit` package).

Tests are defined in objects, which represent a *test set* or *test suite*. In simple cases, we usually define a single object containing the tests. But it is also possible to use parametric test objects or multiple objects defining parametrizable tests or test subsets for testing more complex units and facilitating tests maintenance. Parametric test objects are specially useful to test multiple implementations of the same protocol using a single set of tests by passing the implementation object as a parameter value.

5.13.1 Main files

The `lgtunit.lgt` source file implements a framework for defining and running unit tests in Logtalk. The `lgtunit_messages.lgt` source file defines the default translations for the messages printed when running unit tests. These messages can be intercepted to customize the output, e.g. to make it less verbose or for integration with e.g. GUI IDEs and continuous integration servers.

Other files that are part of this tool provide support for alternative output formats of test results and are discussed below.

5.13.2 API documentation

This tool API documentation is available at:

../../apis/library_index.html#lgtunit

5.13.3 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(lgtunit(loader)).
```

5.13.4 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(lgtunit(tester)).
```

5.13.5 Writing and running tests

In order to write your own unit tests, define objects extending the `lgtunit` object. You may start by copying the `samples/tests-sample.lgt` file (at the root of the Logtalk distribution) to a `tests.lgt` file in your project directory and editing it to add your tests:

```
:- object(tests,
    extends(lgtunit)).

    % test definitions
    ...

:- end_object.
```

The section on *test dialects* below describes in detail how to write tests. See the `tests` top directory for examples of actual unit tests. Other sources of examples are the `library` and `examples` directories.

The tests must be term-expanded by the `lgtunit` object by compiling the source files defining the test objects using the option `hook(lgtunit)`. For example:

```
| ?- logtalk_load(lgtunit(loader)),
     logtalk_load(tests, [hook(lgtunit)]).
```

As the term-expansion mechanism applies to all the contents of a source file, the source files defining the test objects should preferably not contain entities other than the test objects. Additional code necessary for the tests should go to separate files. In general, the tests themselves can be compiled in *optimized* mode. Assuming that's the case, also use the `optimize(on)` compiler option for faster execution.

The term-expansion performed by the `lgtunit` object sets the test object `source_data` flag to `on` and the `context_switching_calls` flag to `allow` for code coverage and debugging support. But these settings can always be overridden in the test objects.

The `samples/tester-sample.lgt` file (at the root of the Logtalk distribution) exemplifies how to compile and load `lgtunit` tool, the source code under testing, the unit tests, and how to automatically run all the tests after loading:

```
:- initialization((
    % minimize compilation reports to the essential ones (errors and warnings)
    set_logtalk_flag(report, warnings),
    % load any necessary library files for your application; for example
    logtalk_load(basic_types(loader)),
    % load the unit test tool
    logtalk_load(lgtunit(loader)),
    % load your application files (e.g., "source.lgt") enabling support for
    % code coverage, which requires compilation in debug mode and collecting
    % source data information; if code coverage is not required, remove the
    % "debug(on)" option for faster execution
    logtalk_load(source, [source_data(on), debug(on)]),
```

(continues on next page)

(continued from previous page)

```
    % compile the unit tests file expanding it using "lgtunit" as the hook
    % object to preprocess the tests; if you have failing tests, add the
    % option debug(on) to debug them (see "tools/lgtunit/NOTES.md" for
    % debugging advice); tests should be loaded after the code being tested
    % is loaded to avoid warnings such as references to unknown entities
    logtalk_load(tests, [hook(lgtunit)]),
    % run all the unit tests; assuming your tests object is named "tests"
    tests::run
)).
```

You may copy this sample file to a `tester.lgt` file in your project directory and edit it to load your project and test files. The `logtalk_tester` testing automation script defaults to looking for test driver files named `tester.lgt` or `tester.logtalk` (if you have work-in-progress test sets that you don't want to run by default, simply use a different file name such as `tester_wip.lgt`; you can still run them automated by using `logtalk_tester -n tester_wip`).

Debugged test sets should preferably be compiled in optimal mode, specially when containing deterministic tests and when using the utility benchmarking predicates.

Assuming a `tester.lgt` driver file as exemplified above, the tests can be run by simply loading this file:

```
| ?- logtalk_load(tester).
```

Assuming your test object is named `tests`, you can re-run the tests by typing:

```
| ?- tests::run.
```

You can also re-run a single test (or a list of tests) using the `run/1` predicate:

```
| ?- tests::run(test_identifier).
```

When testing complex *units*, it is often desirable to split the tests between several test objects or use parametric test objects to be able to run the same tests using different parameters (e.g., different data sets or alternative implementations of the same protocol). In this case, you can run all test subsets using the goal:

```
| ?- lgtunit::run_test_sets([test_set_1, test_set_2, ...]).
```

where the `run_test_sets/1` predicate argument is a list of two or more test object identifiers. This predicate makes it possible to get a single code coverage report that takes into account all the tests.

It's also possible to automatically run loaded tests when using the `make` tool by calling the goal that runs the tests from a definition of the hook predicate `logtalk_make_target_action/1`. For example, by adding to the tests `tester.lgt` driver file the following code:

```
% integrate the tests with logtalk_make/1
:- multifile(logtalk_make_target_action/1).
:- dynamic(logtalk_make_target_action/1).

logtalk_make_target_action(check) :-
    tests::run.
```

Alternatively, you can define the predicate `make/1` inside the test set object. For example:

```
:- object(tests, extends(lgtunit)).

```

(continues on next page)

(continued from previous page)

```
    make(check).
    ...

:- end_object.
```

This clause will cause all tests to be run when calling the `logtalk_make/1` predicate with the target `check` (or its top-level shortcut, `{?}`). The other possible target is `all` (with top-level shortcut `{*}`).

Note that you can have multiple test driver files. For example, one driver file that runs the tests collecting code coverage data and a quicker driver file that skips code coverage and compiles the code to be tested in optimized mode.

5.13.6 Automating running tests

You can use the `scripts/logtalk_tester.sh` Bash shell script or the `scripts/logtalk_tester.ps1` PowerShell script for automating running unit tests (e.g., from a CI/CD pipeline). When using one of the Logtalk installers, the `.sh` extension can usually be omitted. For example, assuming your current directory (or subdirectories) contains one or more `tester.lgt` files:

```
$ logtalk_tester -p gnu
```

The only required argument is the identifier of the backend Prolog system. For other options, see the `scripts/NOTES.md` file or type:

```
$ logtalk_tester -h
```

On POSIX systems, you can also access extended documentation by consulting the script man page:

```
$ man logtalk_tester
```

The scripts support the same set of options. But the option for passing additional arguments to the tests uses different syntax. For example:

```
$ logtalk_tester -p gnu -- foo bar baz

PS> logtalk_tester -p gnu -a foo,bar,baz
```

On POSIX systems, assuming Logtalk was installed using one of the provided installers or installation scripts, there is also a `man` page for the script:

```
$ man logtalk_tester
```

Alternatively, an HTML version of this man page can be found at:

https://logtalk.org/man/logtalk_tester.html

On POSIX systems, the `logtalk_tester.sh` Bash script timeout option requires either a `timeout` or a `gtimeout` command (provided by the GNU coreutils package). The `logtalk_tester.ps1` PowerShell script timeout option requires that Git for Windows is also installed, as it requires the GNU timeout command bundled with it.

In addition to using the `logtalk_tester.ps1` PowerShell script, the Bash shell version of the automation script can also be used in Windows operating-systems with selected backends by using the Bash shell included in the Git for Windows installer. That requires defining a `.profile` file setting the paths to the Logtalk scripts and the Prolog backend executables. For example:

```
$ cat ~/.profile
# YAP
export PATH="/C/Program Files/Yap64/bin":$PATH
# GNU Prolog
export PATH="/C/GNU-Prolog/bin":$PATH
# SWI/Prolog
export PATH="/C/Program Files/swipl/bin":$PATH
# ECLiPSe
export PATH="/C/Program Files/ECLiPSe 7.0/lib/x86_64_nt":$PATH
# SICStus Prolog
export PATH="/C/Program Files/SICStus Prolog VC16 4.6.0/bin":$PATH
# Logtalk
export PATH="$LOGTALKHOME/scripts":"$LOGTALKHOME/integration":$PATH
```

The Git for Windows installer also includes GNU `coreutils` and its `timeout` command, which is used by the `logtalk_tester` script `-t` option.

Note that some tests may give different results when run from within the Bash shell compared with running the tests manually using a Windows GUI version of the Prolog backend. Some backends may also not be usable for automated testing due to the way they are made available as Windows applications.

Additional advice on testing and on automating testing using continuous integration servers can be found at:

https://logtalk.org/testing.html

5.13.7 Parametric test objects

Parameterized unit tests can be easily defined by using parametric test objects. A typical example is testing multiple implementations of the same protocol. In this case, we can use a parameter to pass the specific implementation being tested. For example, assume that we want to run the same set of tests for the library `random_protocol` protocol. We can write:

```
:- object(tests(_RandomObject_),
    extends(lgtunit)).

    :- uses(_RandomObject_, [
        random/1, between/3, member/2,
        ...
    ]).

    test(between_3_in_interval) :-
        between(1, 10, Random),
        1 =< Random, Random =< 10.

    ...

:- end_object.
```

We can then test a specific implementation by instantiating the parameter. For example:

```
| ?- tests(fast_random)::run.
```

Or use the `lgtunit::run_test_sets/1` predicate to test all the implementations:

```
| ?- lgtunit::run_test_sets([
       tests(backend_random),
       tests(fast_random),
       tests(random)
    ]).
```

5.13.8 Test dialects

Multiple test *dialects* are supported by default. See the next section on how to define your own test dialects. In all dialects, a **ground callable term**, usually an atom, is used to uniquely identify a test. This simplifies reporting failed tests and running tests selectively. An error message is printed if invalid or duplicated test identifiers are found. These errors must be corrected; otherwise the reported test results can be misleading. Ideally, tests should have descriptive names that clearly state the purpose of the test and what is being tested.

Unit tests can be written using any of the following predefined dialects:

```
test(Test) :- Goal.
```

This is the most simple dialect, allowing the specification of tests that are expected to succeed. The argument of the `test/1` predicate is the test identifier, which must be unique. A more versatile dialect is:

```
succeeds(Test) :- Goal.
deterministic(Test) :- Goal.
fails(Test) :- Goal.
throws(Test, Ball) :- Goal.
throws(Test, Balls) :- Goal.
```

This is a straightforward dialect. For `succeeds/1` tests, `Goal` is expected to succeed. For `deterministic/1` tests, `Goal` is expected to succeed once without leaving a choice-point. For `fails/1` tests, `Goal` is expected to fail. For `throws/2` tests, `Goal` is expected to throw the exception term `Ball` or one of the exception terms in the list `Balls`. The specified exception must subsume the actual exception for the test to succeed.

An alternative test dialect that can be used with more expressive power is:

```
test(Test, Outcome) :- Goal.
```

The possible values of the outcome argument are:

- `true`
  The test is expected to succeed.
- `true(Assertion)`
  The test is expected to succeed and satisfy the `Assertion` goal.
- `deterministic`
  The test is expected to succeed once without leaving a choice-point.
- `deterministic(Assertion)`
  The test is expected to succeed once without leaving a choice-point and satisfy the `Assertion` goal.
- `subsumes(Expected, Result)`
  The test is expected to succeed by binding `Result` to a term that is subsumed by the `Expected` term.
- `variant(Term1, Term2)`
  The test is expected to succeed by binding `Term1` to a term that is a variant of the `Term2` term.

- `exists(Assertion)`
  A solution exists for the test goal that satisfies the `Assertion` goal.
- `all(Assertion)`
  All test goal solutions satisfy the `Assertion` goal.
- `fail`
  The test is expected to fail.
- `false`
  The test is expected to fail.
- `error(Error)`
  The test is expected to throw the exception term `error(ActualError, _)` where `ActualError` is subsumed `Error`.
- `errors(Errors)`
  The test is expected to throw an exception term `error(ActualError, _)` where `ActualError` is subsumed by an element of the list `Errors`.
- `ball(Ball)`
  The test is expected to throw the exception term `ActualBall` where `ActualBall` is subsumed `Ball`.
- `balls(Balls)`
  The test is expected to throw an exception term `ActualBall` where `ActualBall` is subsumed by an element of the list `Balls`.

In the case of the `true(Assertion)`, `deterministic(Assertion)`, and `all(Assertion)` outcomes, a message that includes the assertion goal is printed for assertion failures and errors to help to debug failed unit tests. Same for the `subsumes(Expected, Result)` and `variant(Term1, Term2)` assertions. Note that this message is only printed when the test goal succeeds, as its failure will prevent the assertion goal from being called. This allows distinguishing between test goal failure and assertion failure.

Note that the `all(Assertion)` outcome simplifies pinpointing which test goal solution failed the assertion. See also the section below on testing non-deterministic predicates.

The `fail` and `false` outcomes are better reserved for cases where there is a single test goal. With multiple test goals, the test will succeed when *any* of those goals fail.

Some tests may require individual condition, setup, or cleanup goals. In this case, the following alternative test dialect can be used:

```
test(Test, Outcome, Options) :- Goal.
```

The currently supported options are (non-recognized options are ignored):

- `condition(Goal)`
  Condition for deciding if the test should be run or skipped (default goal is `true`).
- `setup(Goal)`
  Setup goal for the test (default goal is `true`).
- `cleanup(Goal)`
  Cleanup goal for the test (default goal is `true`).
- `flaky`
  Declare the test as a flaky test.
- `note(Term)`

Annotation to print (between parentheses by default) after the test result (default is `''`); the annotation term can share variables with the test goal, which can be used to pass additional information about the test result.

Also supported is QuickCheck testing, where random tests are automatically generated and run given a predicate mode template with type information for each argument (see the section below for more details):

```
quick_check(Test, Template, Options).
quick_check(Test, Template).
```

The valid options are the same as for the `test/3` dialect plus all the supported QuickCheck specific options (see the QuickCheck section below for details).

For examples of how to write unit tests, check the `tests` folder or the `testing` example in the `examples` folder in the Logtalk distribution. Most of the provided examples also include unit tests, some of them with code coverage.

5.13.9 User-defined test dialects

Additional test dialects can be easily defined by extending the `lgtunit` object and by term-expanding the new dialect into one of the default dialects. As an example, suppose that you want a dialect where you can simply write a file with tests defined by clauses using the format:

```
test_identifier :-
    test_goal.
```

First, we define an expansion for this file into a test object:

```
:- object(simple_dialect,
    implements(expanding)).

    term_expansion(begin_of_file, [(:- object(tests,extends(lgtunit)))]).
    term_expansion((Head :- Body), [test(Head) :- Body]).
    term_expansion(end_of_file, [(:- end_object)]).

:- end_object.
```

Then we can use this hook object to expand and run tests written in this dialect by using a `tester.lgt` driver file with contents such as:

```
:- initialization((
    set_logtalk_flag(report, warnings),
    logtalk_load(lgtunit(loader)),
    logtalk_load(hook_flows(loader)),
    logtalk_load(simple_dialect),
    logtalk_load(tests, [hook(hook_pipeline([simple_dialect,lgtunit]))]),
    tests::run
)).
```

The hook pipeline first applies our `simple_dialect` expansion, followed by the default `lgtunit` expansion. This solution allows other hook objects (e.g., required by the code being tested) to also be used by updating the pipeline.

5.13.10 QuickCheck

QuickCheck was originally developed for Haskell. Implementations for several other programming languages soon followed. QuickCheck provides support for *property-based testing*. The idea is to express properties that predicates must comply with and automatically generate tests for those properties. The `lgtunit` tool supports both `quick_check/2-3` test dialects, as described above, and `quick_check/1-3` public predicates for interactive use:

```
quick_check(Template, Result, Options).
quick_check(Template, Options).
quick_check(Template).
```

The following options are supported:

- `n/1`: number of random tests that will be generated and run (default is 100).
- `s/1`: maximum number of shrink operations when a counter-example is found (default is 64).
- `ec/1`: boolean option deciding if type edge cases are tested before generating random tests (default is `true`).
- `rs/1`: starting seed to be used when generating the random tests (no default).
- `pc/1`: pre-condition closure for generated tests (extended with the test arguments; no default).
- `l/1`: label closure for classifying the generated tests (extended with the test arguments plus the label argument; no default).
- `v/1`: boolean option for verbose reporting of generated random tests (default is `false`).
- `pb/2`: progress bar option for executed random tests when the verbose option is false (first argument is a boolean, default is `false`; second argument is the tick number, a positive integer).

The `quick_check/1` predicate uses the default option values. The `quick_check/1-2` predicates print the test results and are thus better reserved for testing at the top-level interpreter. The `quick_check/3` predicate returns results in reified form:

- `passed(SequenceSeed, Discarded, Labels)`
- `failed(Goal, SequenceSeed, TestSeed)`
- `error(Error, Goal, SequenceSeed, TestSeed)`
- `broken(Why, Culprit)`

The `broken(Why, Culprit)` result only occurs when the user-defined testing setup is broken. For example, a non-callable template (e.g., a non-existing predicate), an invalid option, a problem with the pre-condition closure or with the label closure (e.g., a pre-condition that always fails or a label that fails to classify a generated test), or errors/failures when generating tests (e.g., due to an unknown type being used in the template or a broken custom type arbitrary value generator).

The `Goal` argument is the random test that failed.

The `SequenceSeed` argument is the starting seed used to generate the sequence of random tests. The `TestSeed` is the seed used to generate the test that failed. Both seeds should be regarded as opaque terms. When the test seed is equal to the sequence seed, this means that the failure or error occurred while using only type edge cases. See below how to use the seeds when testing bug fixes.

The `Discarded` argument returns the number of generated tests that were discarded for failing to comply with a pre-condition specified using the `pc/1` option. This option is specially useful when constraining or enforcing a relation between the generated arguments and is often used as an alternative to define a custom type. For example, if we define the following predicate:

```
condition(I) :-
    between(0, 127, I).
```

We can then use it to filter the generated tests:

```
| ?- lgtunit::quick_check(integer(+byte), [pc(condition)]).
% 100 random tests passed, 94 discarded
% starting seed: seed(416,18610,17023)
yes
```

The `Labels` argument returns a list of pairs `Label-N` where `N` is the number of generated tests that are classified as `Label` by a closure specified using the `l/1` option. For example, assuming the following predicate definition:

```
label(I, Label) :-
    (   I mod 2 =:= 0 ->
        Label = even
    ;   Label = odd
    ).
```

We can try:

```
| ?- lgtunit::quick_check(integer(+byte), [l(label), n(10000)]).
% 10000 random tests passed, 0 discarded
% starting seed: seed(25513,20881,16407)
% even: 5037/10000 (50.370000%)
% odd: 4963/10000 (49.630000%)
yes
```

The label statistics are key to verifying that the generated tests provide the necessary coverage. The labeling predicates can return a single test label or a list of test labels. Labels should be ground and are typically atoms. To examine the generated tests themselves, you can use the verbose option, `v/1`. For example:

```
| ?- lgtunit::quick_check(integer(+integer), [v(true), n(7), pc([I]>>(I>5))]).
% Discarded: integer(0)
% Passed:    integer(786)
% Passed:    integer(590)
% Passed:    integer(165)
% Discarded: integer(-412)
% Passed:    integer(440)
% Discarded: integer(-199)
% Passed:    integer(588)
% Discarded: integer(-852)
% Discarded: integer(-214)
% Passed:    integer(196)
% Passed:    integer(353)
% 7 random tests passed, 5 discarded
% starting seed: seed(23671,3853,29824)
yes
```

When a counter-example is found, the verbose option also prints the shrink steps. For example:

```
| ?- lgtunit::quick_check(atom(+atomic), [v(true), ec(false)]).
% Passed:    atom('dyO=Xv_MX-3b/U4KH U')
```

(continues on next page)

(continued from previous page)

```
*     Failure:   atom(-198)
*     Shrinked:  atom(-99)
*     Shrinked:  atom(-49)
*     Shrinked:  atom(-24)
*     Shrinked:  atom(-12)
*     Shrinked:  atom(-6)
*     Shrinked:  atom(-3)
*     Shrinked:  atom(-1)
*     Shrinked:  atom(0)
*     quick check test failure (at test 2 after 8 shrinks):
*       atom(0)
*     starting seed: seed(3172,9814,20125)
*     test seed:     seed(7035,19506,18186)
no
```

The template can be a `(::)/2`, `(<<)/2`, or `(:)/2` qualified callable term. When the template is an unqualified callable term, it will be used to construct a goal to be called in the context of the *sender* using the `(<<)/2` debugging control construct. Another simple example is passing a template that will trigger a failed test (as the `random::random/1` predicate always returns non-negative floats):

```
| ?- lgtunit::quick_check(random::random(-negative_float)).
*     quick check test failure (at test 1 after 0 shrinks):
*       random::random(0.09230089279334841)
*     starting seed: seed(3172,9814,20125)
*     test seed:     seed(3172,9814,20125)
no
```

When QuickCheck exposes a bug in the tested code, we can use the reported counter-example to help diagnose it and fix it. As tests are randomly generated, we can use the starting seed reported with the counter-example to confirm the bug fix by calling the `quick_check/2-3` predicates with the `rs(Seed)` option. For example, assume the following broken predicate definition:

```
every_other([], []).
every_other([_, X| L], [X | R]) :-
    every_other(L, R).
```

The predicate is supposed to construct a list by taking every other element of an input list. Cursory testing may fail to notice the bug:

```
| ?- every_other([1,2,3,4,5,6], List).
List = [2, 4, 6]
yes
```

But QuickCheck will report a bug with lists with an odd number of elements with a simple property that verifies that the predicate always succeeds and returns a list of integers:

```
| ?- lgtunit::quick_check(every_other(+list(integer), -list(integer))).
*     quick check test failure (at test 2 after 0 shrinks):
*       every_other([0],A)
*     starting seed: seed(3172,9814,20125)
*     test seed:     seed(3172,9814,20125)
no
```

We could fix this particular bug by rewriting the predicate:

```
every_other([], []).
every_other([H| T], L) :-
    every_other(T, H, L).

every_other([], X, [X]).
every_other([_| T], X, [X| L]) :-
    every_other(T, L).
```

By retesting with the same test seed that uncovered the bug, the same random test that found the bug will be generated and run again:

```
| ?- lgtunit::quick_check(
        every_other(+list(integer), -list(integer)),
        [rs(seed(3172,9814,20125))]
    ).
% 100 random tests passed, 0 discarded
% starting seed: seed(3172,9814,20125)
yes
```

Still, after verifying the bug fix, is also a good idea to re-run the tests using the sequence seed instead, as bug fixes sometimes cause regressions elsewhere.

When retesting using the `logtalk_tester` automation script, the starting seed can be set using the `-r` option. For example:

```
$ logtalk_tester -r "seed(3172,9814,20125)"
```

We could now move to other properties that the predicate should comply with (e.g., all elements in the output list being present in the input list). Often, both traditional unit tests and QuickCheck tests are used, complementing each other to ensure the required code coverage.

Another example using a Prolog module predicate:

```
| ?- lgtunit::quick_check(
        pairs:pairs_keys_values(
            +list(pair(atom,integer)),
            -list(atom),
            -list(integer)
        )
    ).
% 100 random tests passed, 0 discarded
% starting seed: seed(3172,9814,20125)
yes
```

As illustrated by the examples above, properties are expressed using predicates. In the most simple cases, that can be the predicate that we are testing itself. But, in general, it will be an auxiliary predicate calling the predicate or predicates being tested and checking properties that the results must comply with.

The QuickCheck test dialects and predicates take as argument the mode template for a property, generate random values for each input argument based on the type information, and check each output argument. For common types, the implementation tries first (by default) common edge cases (e.g., empty atom, empty list, or zero) before generating arbitrary values. When the output arguments check fails, the QuickCheck implementation tries (by default) up to 64 shrink operations of the counter-example to report a simpler case to help debugging the failed test. Edge cases, generating arbitrary terms, and shrinking terms make use of the library `arbitrary` category via the `type` object (both entities can be extended by the user by defining clauses for multifile predicates).

The mode template syntax is the same as that used in the `info/2` predicate directives with an additional notation, `{}/1`, for passing argument values as-is instead of generating random values for these arguments. For example, assume that we want to verify the `type::valid/2` predicate, which takes as its first argument a type. Randomly generating random types would be cumbersome at best but the main problem is that we need to generate random values for the second argument according to the first argument. Using the `{}/1` notation, we can solve this problem for any specific type, e.g. integer, by writing:

```
| ?- lgtunit::quick_check(type::valid({integer}, +integer)).
```

We can also test all (ground, i.e. non-parametric) types with arbitrary value generators by writing:

```
| ?- forall(
        (type::type(Type), ground(Type), type::arbitrary(Type)),
        lgtunit::quick_check(type::valid({Type}, +Type))
    ).
```

You can find the list of the basic supported types for use in the template in the API documentation for the library entities `type` and `arbitrary`. Note that other library entities, including third-party or your own, can contribute with additional type definitions, as both `type` and `arbitrary` entities are user-extensible by defining clauses for their multifile predicates.

The user can define new types to use in the property mode templates to use with its QuickCheck tests by defining clauses for the `type` library object and the `arbitrary` library category multifile predicates. QuickCheck will use the later to generate arbitrary input arguments and the former to verify output arguments. As a toy example, assume that the property mode template has an argument of type `bit` with possible values `0` and `1`. We would then need to define:

```
:- multifile(type::type/1).
type::type(bit).

:- multifile(type::check/2).
type::check(bit, Term) :-
    once((Term == 0; Term == 1)).

:- multifile(arbitrary::arbitrary/1).
arbitrary::arbitrary(bit).

:- multifile(arbitrary::arbitrary/2).
arbitrary::arbitrary(bit, Arbitrary) :-
    random::member(Arbitrary, [0, 1]).
```

5.13.11 Skipping tests

A test object can define the `condition/0` predicate (which defaults to `true`) to test if some necessary condition for running the tests holds. The tests are skipped if the call to this predicate fails or generates an error.

Individual tests that for some reason should be unconditionally skipped can have the test clause head prefixed with the `(-)/1` operator. For example:

```
- test(not_yet_ready) :-
    ...
```

In this case, it's a good idea to use the `test/3` dialect with a `note/1` option that briefly explains why the test is being skipped. For example:

```
- test(xyz_reset, true, [note('Feature xyz reset not yet implemented')]) :-
    ...
```

The number of skipped tests is reported together with the numbers of passed and failed tests. To skip a test depending on some condition, use the `test/3` dialect and the `condition/1` option. For example:

```
test(test_id, true, [condition(current_prolog_flag(bounded,true))]) :-
    ...
```

The test is skipped if the condition goal fails or generates an error. The conditional compilation directives can also be used in alternative, but note that in this case there will be no report on the number of skipped tests.

5.13.12 Selecting tests

While debugging an application, we often want to temporarily run just a selection of relevant tests. This is specially useful when running all the tests slows down and distracts from testing fixes for a specific issue. This can be accomplished by prefixing the clause heads of the selected tests with the `(+)/1` operator. For example:

```
:- object(tests,
    extends(lgtunit)).

    cover(ack).

    test(ack_1, true(Result == 11)) :-
        ack::ack(2, 4, Result).

    + test(ack_2, true(Result == 61)) :-
        ack::ack(3, 3, Result).

    test(ack_3, true(Result == 125)) :-
        ack::ack(3, 4, Result).

:- end_object.
```

In this case, only the `ack_2` would run. Just be careful to remove all `(+)/1` test prefixes when done debugging the issue that prompted you to run just the selected tests. After, be sure to run all the tests to ensure there are no regressions introduced by your fixes.

5.13.13 Checking test goal results

Checking test goal results can be performed using the `test/2-3` supported outcomes such as `true(Assertion)` and `deterministic(Assertion)`. For example:

```
test(compare_3_order_less, deterministic(Order == (<))) :-
    compare(Order, 1, 2).
```

For the other test dialects, checking test goal results can be performed by calling the `assertion/1-2` utility predicates or by writing the checking goals directly in the test body. For example:

```
test(compare_3_order_less) :-
    compare(Order, 1, 2),
    ^^assertion(Order == (<)).
```

or:

```
succeeds(compare_3_order_less) :-
    compare(Order, 1, 2),
    Order == (<).
```

Using assertions is, however, preferable to directly checking test results in the test body as it facilitates debugging by printing the unexpected results when the assertions fail.

The `assertion/1-2` utility predicates are also useful for the `test/2-3` dialects when we want to check multiple assertions in the same test. For example:

```
test(dictionary_clone_4_01, true) :-
    as_dictionary([], Dictionary),
    clone(Dictionary, DictionaryPairs, Clone, ClonePairs),
    empty(Clone),
    ^^assertion(original_pairs, DictionaryPairs == []),
    ^^assertion(clone_pairs, ClonePairs == []).
```

Ground results can be compared using the standard `==/2` term equality built-in predicate. Non-ground results can be compared using the `variant/2` predicate provided by `lgtunit`. The standard `subsumes_term/2` built-in predicate can be used when testing a compound term structure while abstracting some of its arguments. Floating-point numbers can be compared using the `=~=/2`, `approximately_equal/3`, `essentially_equal/3`, and `tolerance_equal/4` predicates provided by `lgtunit`. Using the `=/2` term unification built-in predicate is almost always an error, as it would mask test goals failing to bind output arguments. The `lgtunit` tool implements a linter check for the use of unification goals in test outcome assertions. In the rare cases that a unification goal is intended, wrapping the `(=)/2` goal using the `{}/1` control construct avoids the linter warning.

When the meta-argument of the `assertion/1-2` predicates is call to a local predicate (in the tests object), you need to call them using the `(::)/2` message-sending control construct instead of the `(^^)/2` *super* call control construct. This is necessary as *super* calls preserve the *sender*, and the tests are implicitly run by the `lgtunit` object sending a message to the tests object. For example:

```
:- uses(lgtunit, [
    assertion/1
]).

test(my_test_id, true) :-
    foo(X, Y),
    assertion(consistent(X, Y)).

consistent(X, Y) :-
    ...
```

In this case, the *sender* is the tests object, and the `assertion/1` meta-predicate will call the local `consistent/2` predicate in the expected context.

5.13.14 Testing local predicates

The `(<<)/2` debugging control construct can be used to access and test object local predicates (i.e., predicates without a scope directive). In this case, make sure that the `context_switching_calls` compiler flag is set to `allow` for those objects. This is seldom required, however, as local predicates are usually auxiliary predicates called by public predicates and thus tested when testing those public predicates. The code coverage support can pinpoint any local predicate clause that is not being exercised by the tests.

5.13.15 Testing non-deterministic predicates

For testing non-deterministic predicates (with a finite and manageable number of solutions), you can wrap the test goal using the standard `findall/3` predicate to collect all solutions and check against the list of expected solutions. When the expected solutions are a set, use in alternative the standard `setof/3` predicate.

If you want to check that all solutions of a non-deterministic predicate satisfy an assertion, use the `test/2` or `test/3` test dialect with the `all(Assertion)` outcome. For example:

```
test(atom_list, all(atom(Item))) :-
    member(Item, [a, b, c]).
```

See also the next section on testing *generators*.

If you want to check that a solution exists for a non-deterministic predicate that satisfies an assertion, use the `test/2` or `test/3` test dialect with the `exists(Assertion)` outcome. For example:

```
test(at_least_one_atom, exists(atom(Item))) :-
    member(Item, [1, foo(2), 3.14, abc, 42]).
```

5.13.16 Testing generators

To test all solutions of a predicate that acts as a *generator*, we can use either the `all/1` outcome or the `forall/2` predicate as the test goal with the `assertion/2` predicate called to report details on any solution that fails the test. For example:

```
test(test_solution_generator, all(test(X,Y,Z))) :-
    generator(X, Y, Z).
```

or:

```
:- uses(lgtunit, [assertion/2]).
...

test(test_solution_generator_2) :-
    forall(
        generator(X, Y, Z),
        assertion(generator(X), test(X,Y,Z))
    ).
```

While using the `all/1` outcome results in a more compact test definition, using the `forall/2` predicate allows customizing the assertion description. In the example above, we use the `generator(X)` description instead of the `test(X,Y,Z)` description implicit when we use the `all/1` outcome.

5.13.17 Testing input/output predicates

Extensive support for testing input/output predicates is provided, based on similar support found on the Prolog conformance testing framework written by Péter Szabó and Péter Szeredi.

Two sets of predicates are provided, one for testing text input/output and one for testing binary input/output. In both cases, temporary files (possibly referenced by a user-defined alias) are used. The predicates allow setting, checking, and cleaning text/binary input/output. These predicates are declared as protected and thus called using the `(^^/1)` control construct.

As an example of testing an input predicate, consider the standard `get_char/1` predicate. This predicate reads a single character (atom) from the current input stream. Some test for basic functionality could be:

```
test(get_char_1_01, true(Char == 'q')) :-
    ^^set_text_input('qwerty'),
    get_char(Char).

test(get_char_1_02, true(Assertion)) :-
    ^^set_text_input('qwerty'),
    get_char(_Char),
    ^^text_input_assertion('werty', Assertion).
```

As you can see in the above example, the testing pattern consists on setting the input for the predicate being tested, calling it, and then checking the results. It is also possible to work with streams other than the current input/output streams by using the `lgtunit` predicate variants that take a stream alias as argument. For example, when testing the standard `get_char/2` predicate, we could write:

```
test(get_char_2_01, true(Char == 'q')) :-
    ^^set_text_input(in, 'qwerty'),
    get_char(in, Char).

test(get_char_2_02, true(Assertion)) :-
    ^^set_text_input(in, 'qwerty'),
    get_char(in, _Char),
    ^^text_input_assertion(in, 'werty', Assertion).
```

Testing output predicates follows a similar pattern by using instead the `set_text_output/1-2` and `text_output_assertion/2-3` predicates. For example:

```
test(put_char_2_02, true(Assertion)) :-
    ^^set_text_output(out, 'qwert'),
    put_char(out, y),
    ^^text_output_assertion(out, 'qwerty', Assertion).
```

The `set_text_output/1` predicate diverts only the standard output stream (to a temporary file) using the standard `set_output/1` predicate. Most backend Prolog systems also support writing to the de facto standard error stream. But there's no standard solution to divert this stream. However, several systems provide a `set_stream/2` or similar predicate that can be used for stream redirection. For example, assume that you wanted to test a backend Prolog system warning when an `initialization/1` directive fails that is written to `user_error`. An hypothetical test could be:

```
test(singletons_warning, true(Assertion)) :-
    ^^set_text_output(''),
    current_output(Stream),
    set_stream(Stream, alias(user_error)),
```

(continues on next page)

(continued from previous page)

```
    consult(broken_file),
    ^^text_output_assertion('WARNING: initialization/1 directive failed', Assertion).
```

For testing binary input/output predicates, equivalent testing predicates are provided. There is also a small set of helper predicates for dealing with stream handles and stream positions. For testing with files instead of streams, testing predicates are provided that allow creating text and binary files with given contents and check text and binary files for expected contents.

For more practical examples, check the included tests for Prolog standard conformance of built-in input/output predicates.

5.13.18 Suppressing tested predicates output

Sometimes predicates being tested output text or binary data that, at best, clutters testing logs and, at worst, can interfere with parsing of test logs. If that output itself is not under testing, you can suppress it by using the goals `^^suppress_text_output` or `^^suppress_binary_output` at the beginning of the tests. For example:

```
test(proxies_04, true(Color == yellow)) :-
    ^^suppress_text_output,
    {circle('#2', Color)}::print.
```

The `suppress_text_output/0` and `suppress_binary_output/0` predicates work by redirecting standard output to the operating-system null device. But the application may also output to e.g. `user_error` and other streams. If this output must also be suppressed, several alternatives are described next.

Output of expected warnings can be suppressed by turning off the corresponding linter flags. In this case, it is advisable to restrict the scope of the flag value changes as much as possible.

Output of expected compiler errors can be suppressed by defining suitable clauses for the `logtalk::message_hook/4` hook predicate. For example:

```
:- multifile(logtalk::message_hook/4).
:- dynamic(logtalk::message_hook/4).

% ignore expected domain error
logtalk::message_hook(compiler_error(_,_,error(domain_error(foo,bar),_)), error, core, _).
```

In this case, it is advisable to restrict the scope of the clauses as much as possible to exact exception terms. For the exact message terms, see the `core_messages` category source file. Defining this hook predicate can also be used to suppress all messages from a given component. For example:

```
:- multifile(logtalk::message_hook/4).
:- dynamic(logtalk::message_hook/4).

logtalk::message_hook(_Message, _Kind, code_metrics, _Tokens).
```

Note that there's no portable solution to suppress *all* output. However, several systems provide a `set_stream/2` or similar predicate that can be used for stream redirection. Check the documentation of the backend Prolog systems you're using for details.

5.13.19 Tests with timeout limits

There's no portable way to call a goal with a timeout limit. However, some backend Prolog compilers provide this functionality:

- B-Prolog: `time_out/3` built-in predicate
- ECLiPSe: `timeout/3` and `timeout/7` library predicates
- XVM: `call_with_timeout/2-3` built-in predicates
- SICStus Prolog: `time_out/3` library predicate
- SWI-Prolog: `call_with_time_limit/2` library predicate
- Trealla Prolog: `call_with_time_limit/2` and `time_out/3` library predicates
- XSB: `timed_call/2` built-in predicate
- YAP: `time_out/3` library predicate

Logtalk provides a `timeout` portability library implementing a simple abstraction for those backend Prolog compilers.

The `logtalk_tester` automation script accepts a timeout option that can be used to set a limit per-test set.

5.13.20 Setup and cleanup goals

A test object can define `setup/0` and `cleanup/0` goals. The `setup/0` predicate is called, when defined, before running the object unit tests. The `cleanup/0` predicate is called, when defined, after running all the object unit tests. The tests are skipped when the setup goal fails or throws an error. For example:

```
cleanup :-
    this(This),
    object_property(This, file(_,Directory)),
    atom_concat(Directory, serialized_objects, File),
    catch(ignore(os::delete_file(File)), _, true).
```

Per test setup and cleanup goals can be defined using the `test/3` dialect and the `setup/1` and `cleanup/1` options. The test is skipped when the setup goal fails or throws an error. Note that a broken test cleanup goal doesn't affect the test but may adversely affect any following tests. Variables in the setup and cleanup goals are shared with the test body.

5.13.21 Test annotations

It's possible to define per-unit and per-test annotations to be printed after the test results or when tests are skipped. This is particularly useful when some units or some unit tests may be run while still being developed. Annotations can be used to pass additional information to a user reviewing test results. By intercepting the unit test framework message printing calls (using the `message_hook/4` hook predicate), test automation scripts and integrating tools can also access these annotations.

Units can define a global annotation using the predicate `note/1`. To define per-test annotations, use the `test/3` dialect and the `note/1` option. For example, you can inform why a test is being skipped by writing:

```
- test(foo_1, true, [note('Waiting for Deep Thought answer')]) :-
    ...
```

Another common use is to return the execution time of one of the test sub-goals. For example:

```
test(foobar, true, [note(bar(seconds-Time))]) :-
    foo(...),
    benchmark(bar(...), Time).
```

Annotations are written, by default, between parentheses after and in the same line as the test results.

5.13.22 Test execution times and memory usage

Individual test CPU and wall execution times (in seconds) are reported by default when running the tests. Total CPU and wall execution times for passed and failed tests are reported after the tests complete. Starting and ending date and time when running a set of tests is also reported by default. The `lgtunit` object also provides several public benchmarking predicates that can be useful for e.g. reporting test sub-goals execution times using either CPU or wall clocks. When running multi-threaded code, the CPU time may or may not include all threads CPU time depending on the backend.

Be aware that the accuracy of CPU and wall time depends of the backend. Accuracy can also be different between CPU and wall time (e.g. CPU time can have nanosecond accuracy with wall time only having millisecond accuracy).

Test memory usage is not reported by default due to the lack of a portable solution to access memory data. However, several backend Prolog systems provide a `statistics/2` or similar predicate that can be used for a custom solution. Depending on the system, individual keys may be provided for each memory area (heap, trail, atom table, ...). Aggregating keys may also be provided. As a hypothetical example, assume you're running Logtalk with a backend providing a `statistics/2` predicate with a `memory_used` key:

```
test(ack_3, true(Result == 125), [note(memory-Memory)]) :-
    statistics(memory_used, Memory0),
    ack::ack(3, 4, Result),
    statistics(memory_used, Memory1),
    Memory is Memory1 - Memory0.
```

Consult the documentation of the backend Prolog systems for actual details.

5.13.23 Working with test data files

Frequently, tests make use of test data files that are usually stored in the test set directory or in sub-directories. These data files are referenced using their relative paths. But to allow the tests to run independently of the Logtalk process current directory, the relative paths often must be expanded into an absolute path before being passed to the predicates being tested. The `file_path/2` protected predicate can be used in the test definitions to expand the relative paths. For example:

```
% check that the encoding/1 option is accepted
test(lgt_unicode_open_4_01, true) :-
    ^^file_path(sample_utf_8, Path),
    open(Path, write, Stream, [encoding('UTF-8')]),
    close(Stream).
```

The absolute path is computed relative to the path of *self*, i.e. relative to the path of the test object that received the message that runs the tests.

It's also common for tests to create temporary files and directories that should be deleted after the tests completion. The `clean_file/1` and `clean_directory/1` protected predicates can be used for this purpose. For example, assuming that the tests create a `foo.txt` text file and a `tmp` directory in the same directory as the tests object:

```
cleanup :-
    ^^clean_file('foo.txt'),
    ^^clean_directory('tmp').
```

Similar to the `file_path/2` predicate, relative paths are interpreted as relative to the path of the test object. This predicate also closes any open stream connected to the file before deleting it.

5.13.24 Flaky tests

Flaky tests are tests that pass or fail non-deterministically, usually due to external conditions (e.g., computer or network load). Thus, flaky tests often don't result from bugs in the code being tested itself but from test execution conditions that are not predictable. The `flaky/0` test option declares a test to be flaky. For example:

```
test(foo, true, [flaky]) :-
    ...
```

For backwards compatibility, the `note/1` annotation can also be used to alert that a test failure is for a flaky test when its argument is an atom containing the sub-atom `flaky`.

The testing automation support outputs the text `[flaky]` when reporting failed flaky tests. Moreover, the `logtalk_tester` automation script will ignore failed flaky tests when setting its exit status.

5.13.25 Mocking

Sometimes the code being tested performs complex tasks that are not feasible or desirable when running tests. For example, the code may perform a login operation requiring the user to provide a username and a password using some GUI widget. In this case, the tests may require the login operation to still be performed but using canned data (also simplifying testing automation). I.e. we want to *mock* (as in *imitate*) the login procedure. Ideally, this should be accomplished without requiring any changes to the code being tested. Logtalk provides two solutions that can be used for mocking: *term-expansion* and *hot patching*. A third solution is possible if the code we want to mock uses the *message printing mechanism*.

Using the term-expansion mechanism, we would define a *hook object* that expands the login predicate into a fact:

```
:- object(mock_login,
    implements(expanding)).

    term_expansion((login(_, _) :- _), login(jdoe, test123)).

:- end_object.
```

The tests driver file would then load the application object responsible for user management using this hook object:

```
:- initialization((
    ...,
    logtalk_load(mock_login),
    logtalk_load(user_management, [hook(mock_login)]),
    ...
)).
```

Using hot patching, we would define a *complementing category* patching the object that defines the login predicate:

```
:- category(mock_login,
    complements(user_management)).

    login(jdoe, test123).

:- end_category.
```

The tests driver file would then set the `complements` flag to `allow` and load the patch after loading application code:

```
:- initialization((
    ...,
    set_logtalk_flag(complements, allow),
    logtalk_load(application),
    logtalk_load(mock_login),
    ...
)).
```

There are pros and cons for each solution. Term-expansion works by defining hook objects that are used at compile-time, while hot patching happens at runtime. Complementing categories can also be dynamically created, stacked, and abolished. Hot patching disables static binding optimizations, but that's usually not a problem as the code being tested is often compiled in debug mode to collect code coverage data. Two advantages of the term-expansion solution are that it allows defining conditions for expanding terms and goals and can replace both predicate definitions and predicate calls. Limitations in the current Prolog standards prevent patching callers to local predicates being patched. But often both solutions can be used, with the choice depending on code clarity and user preference. See the Handbook sections on term-expansion and hot patching for more details on these mechanisms.

In those cases where the code we want to mock uses the message printing mechanism, the solution is to intercept and rewrite the messages being printed and/or the questions being asked using the `logtalk::message_hook/4` and `logtalk::question_hook/6` hook predicates.

5.13.26 Debugging messages in tests

Sometimes it is useful to write debugging or logging messages from tests when running them manually. But those messages are better suppressed when running the tests automated. A common solution is to use debug *meta-messages*. For example:

```
:- uses(logtalk, [
    print_message(debug, my_app, Message) as dbg(Message)
]).

test(some_test_id, ...) :-
    ...,
    dbg('Some intermediate value'-Value),
    ...,
    dbg([Stream]>>custom_print_goal(Stream, ...)),
    ...
```

The messages are only printed (and the user-defined printing goals are only called) when the `debug` flag is turned on. Note that this doesn't require compiling the tests in debug mode: you simply toggle the flag to

toggle the debug messages. Also note that the `print_message/3` goals are suppressed by the compiler when compiling with the `optimize` flag turned on.

5.13.27 Debugging failed tests

Debugging of failed unit tests is simplified by using test assertions as the reason for the assertion failures is printed out. Thus, use preferably the `test/2-3` dialects with `true(Assertion)`, `deterministic(Assertion)`, `subsumes(Expected, Result)`, or `variant(Term1, Term2)` outcomes. If a test checks multiple assertions, you can use the predicate `assertion/2` in the test body. In the case of QuickCheck tests, the `v(true)` verbose option can be used to print the generated test case that failed if necessary.

If the assertion failures don't provide enough information, you can use the `debugger` tool to debug failed unit tests. Start by compiling the unit test objects and the code being tested in debug mode. Load the debugger and trace the test that you want to debug. For example, assuming your tests are defined in a `tests` object and that the identifier of the test to be debugged is `test_foo`:

```
| ?- logtalk_load(debugger(loader)).
...

| ?- debugger::trace.
...

| ?- tests::run(test_foo).
...
```

You can also compile the code and the tests in debug mode but without using the `hook/1` compiler option for the tests compilation. Assuming that the `context_switching_calls` flag is set to `allow`, you can then use the `(<<)/2` debugging control construct to debug the tests. For example, assuming that the identifier of the test to be debugged is `test_foo` and that you used the `test/1` dialect:

```
| ?- logtalk_load(debugger(loader)).
...

| ?- debugger::trace.
...

| ?- tests<<test(test_foo).
...
```

In the more complicated cases, it may be worth defining `loader_debug.lgt` and `tester_debug.lgt` driver files that load code and tests in debug mode and also load the debugger.

5.13.28 Code coverage

If you want entity predicate clause coverage information to be collected and printed, you will need to compile the entities that you're testing using the flags `debug(on)` and `source_data(on)`. Be aware, however, that compiling in debug mode results in a performance penalty.

A single test object may include tests for one or more entities (objects, protocols, and categories). The entities being tested by a unit test object for which code coverage information should be collected must be declared using the `cover/1` predicate. For example, to collect code coverage data for the objects `foo` and `bar` include in the tests object the two clauses:

```
cover(foo).
cover(bar).
```

Code coverage is listed using the predicates clause indexes (counting from one). For example, using the `points` example in the Logtalk distribution:

```
% point: default_init_option/1 - 2/2 - (all)
% point: instance_base_name/1 - 1/1 - (all)
% point: move/2 - 1/1 - (all)
% point: position/2 - 1/1 - (all)
% point: print/0 - 1/1 - (all)
% point: process_init_option/1 - 1/2 - [1]
% point: position_/2 - 0/0 - (all)
% point: 7 out of 8 clauses covered, 87.500000% coverage
```

The numbers after the predicate indicators represent the clauses covered and the total number of clauses. E.g. for the `process_init_option/1` predicate, the tests cover 1 out of 2 clauses. After these numbers, we either get `(all)` telling us that all clauses are covered or a list of indexes for the covered clauses. E.g. only the first clause for the `process_init_option/1` predicate, `[1]`. Summary clause coverage numbers are also printed for entities and for clauses across all entities.

In the printed predicate clause coverage information, you may get a total number of clauses smaller than the covered clauses. This results from the use of dynamic predicates with clauses asserted at runtime. You may easily identify dynamic predicates in the results as their clauses often have an initial count equal to zero.

The list of indexes of the covered predicate clauses can be quite long. Some backend Prolog compilers provide a flag or a predicate to control the depth of printed terms that can be useful:

- CxProlog: `write_depth/2` predicate
- ECLiPSe: `print_depth` flag
- XVM 3.2.0 or later: `answer_write_options` flag
- SICStus Prolog: `toplevel_print_options` flag
- SWI-Prolog 7.1.10 or earlier: `toplevel_print_options` flag
- SWI-Prolog 7.1.11 or later: `answer_write_options` flag
- Trealla Prolog: `answer_write_options` flag
- XSB: `set_file_write_depth/1` predicate
- YAP: `write_depth/2-3` predicates

Code coverage is only available when testing Logtalk code. But Prolog modules can often be compiled as Logtalk objects and plain Prolog code may be wrapped in a Logtalk object. For example, assuming a `module.pl` module file, we can compile and load the module as an object by simply calling:

```
| ?- logtalk_load(module).
...
```

The module exported predicates become object public predicates. For a plain Prolog file, say `plain.pl`, we can define a Logtalk object that wraps the code using an `include/1` directive:

```
:- object(plain).

    :- include('plain.pl').
```

(continues on next page)

(continued from previous page)

```
:- end_object.
```

The object can also declare as public the top Prolog predicates to simplify writing the tests. In alternative, we can use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(plain, [hook(object_wrapper_hook)]).
...
```

These workarounds may thus allow generating code coverage data also for Prolog code by defining tests that use the `(<<)/2` debugging control construct to call the Prolog predicates.

See also the section below on exporting code coverage results to XML files, which can be easily converted and published as e.g. HTML reports.

5.13.29 Utility predicates

The `lgtunit` tool provides several public utility predicates to simplify writing unit tests and for general use:

- `variant(Term1, Term2)`
  To check when two terms are a variant of each other (e.g., to check expected test results against actual results when they contain variables).
- `assertion(Goal)`
  To generate an exception in case the goal argument fails or throws an error.
- `assertion(Description, Goal)`
  To generate an exception in case the goal argument fails or throws an error (the first argument allows assertion failures to be distinguished when using multiple assertions).
- `approximately_equal(Number1, Number2)`
  For number approximate equality using the `epsilon` arithmetic constant value.
- `approximately_equal(Number1, Number2, Epsilon)`
  For number approximate equality. Weaker equality than essential equality.
- `essentially_equal(Number1, Number2, Epsilon)`
  For number essential equality. Stronger equality than approximate equality.
- `tolerance_equal(Number1, Number2, RelativeTolerance, AbsoluteTolerance)`
  For number equality within tolerances.
- `Number1 =~= Number2`
  For number (or list of numbers) close equality (usually floating-point numbers).
- `benchmark(Goal, Time)`
  For timing a goal.
- `benchmark_reified(Goal, Time, Result)`
  Reified version of `benchmark/2`.
- `benchmark(Goal, Repetitions, Time)`
  For finding the average time to prove a goal.

- `benchmark(Goal, Repetitions, Clock, Time)`
  For finding the average time to prove a goal using a `cpu` or a `wall` clock.
- `deterministic(Goal)`
  For checking that a predicate succeeds without leaving a choice-point.
- `deterministic(Goal, Deterministic)`
  Reified version of the `deterministic/1` predicate.

The `assertion/1-2` predicates can be used in the body of tests where using two or more assertions is convenient or in the body of tests written using the `test/1`, `succeeds/1`, and `deterministic/1` dialects to help differentiate between the test goal and checking the test goal results and to provide more informative test failure messages.

When the assertion, benchmarking, and deterministic meta-predicates call a local predicate of the tests object, you must call them using an implicit or explicit message instead of using a *super* call. For example, to use an implicit message to call the `assertion/1-2` meta-predicates, add the following directive to the tests object:

```
:- uses(lgtunit, [assertion/1, assertion/2]).
```

The reason this is required is that meta-predicates goals arguments are always called in the context of the *sender*, which would be the `lgtunit` object in the case of a `(^^)/2` call (as it preserves both *self* and *sender* and the tests are internally run by a message sent from the `lgtunit` object to the tests object).

As the `benchmark/2-4` predicates are meta-predicates, turning on the `optimize` compiler flag is advised to avoid runtime compilation of the meta-argument, which would add an overhead to the timing results. But this advice conflicts with collecting code coverage data, which requires compilation in debug mode. The solution is to use separate test objects for benchmarking and for code coverage. Note that the CPU and wall execution times (in seconds) for each individual test are reported by default when running the tests.

The `(=~=)/2` predicate is typically used by adding the following directive to the object (or category) calling it:

```
:- uses(lgtunit, [
    op(700, xfx, =~=), (=~=)/2
]).
```

Consult the `lgtunit` object API documentation for more details on these predicates.

5.13.30 Exporting test results in xUnit XML format

To output test results in the xUnit XML format (from JUnit; see e.g. https://github.com/windyroad/JUnit-Schema or https://llg.cubic.org/docs/junit/), simply load the `xunit_output.lgt` file before running the tests. This file defines an object, `xunit_output`, that intercepts and rewrites unit test execution messages, converting them to the xUnit XML format.

To export the test results to a file using the xUnit XML format, simply load the `xunit_report.lgt` file before running the tests. A file named `xunit_report.xml` will be created in the same directory as the object defining the tests. When running a set of test suites as a single unified suite (using the `run_test_sets/1` predicate), the single xUnit report is created in the directory of the first test suite object in the set.

To use the xUnit.net v2 XML format (https://xunit.net/docs/format-xml-v2), load either the `xunit_net_v2_output.lgt` file or the `xunit_net_v2_report.lgt` file.

When using the `logtalk_tester` automation script, use either the `-f xunit` option or the `-f xunit_net_v2` option to generate the `xunit_report.xml` files in the test set directories.

There are several third-party xUnit report converters that can generate HTML files for easy browsing. For example:

- https://allurereport.org/docs/ (supports multiple reports)
- https://github.com/Zir0-93/xunit-to-html (supports multiple test sets in a single report)
- https://www.npmjs.com/package/xunit-viewer
- https://github.com/JatechUK/NUnit-HTML-Report-Generator
- https://plugins.jenkins.io/xunit

5.13.31 Exporting test results in the TAP output format

To output test results in the TAP (Test Anything Protocol) format, simply load the `tap_output.lgt` file before running the tests. This file defines an object, `tap_output`, that intercepts and rewrites unit test execution messages, converting them to the TAP output format.

To export the test results to a file using the TAP (Test Anything Protocol) output format, load instead the `tap_report.lgt` file before running the tests. A file named `tap_report.txt` will be created in the same directory as the object defining the tests.

When using the `logtalk_tester` automation script, use the `-f tap` option to generate the `tap_report.xml` files in the test set directories.

When using the `test/3` dialect with the TAP format, a `note/1` option whose argument is an atom starting with a `TODO` or `todo` word results in a test report with a TAP TODO directive.

When running a set of test suites as a single unified suite, the single TAP report is created in the directory of the first test suite object in the set.

There are several third-party TAP report converters that can generate HTML files for easy browsing. For example:

- https://github.com/Quobject/tap-to-html
- https://plugins.jenkins.io/tap/

5.13.32 Generating Allure reports

A `logtalk_allure_report.pl` Bash shell script and a `logtalk_allure_report.ps1` PowerShell script are provided for generating Allure reports (version 2.26.0 or later required). This requires exporting test results in xUnit XML format. A simple usage example (assuming a current directory containing tests):

```
$ logtalk_tester -p gnu -f xunit
$ logtalk_allure_report
$ allure open
```

The `allure open` command accepts `--host` and `--port` arguments in case their default values are not suitable (e.g., when running Logtalk in a remote host over a SSH connection).

The `logtalk_allure_report` script supports command-line options to pass the tests directory (i.e., the directory where the `logtalk_tester` script was run), the directory where to collect all the xUnit report files for generating the report, the directory where the report is to be saved, and the report title (see the script man page or type `logtalk_allure_report -h`). The script also supports saving the history of past test runs. In this case, a persistent location for both the results and report directories must be used.

It's also possible to use the script just to collect the xUnit report files generated by `lgtunit` and delegate the actual generation of the report to e.g. an Allure Docker container or to a Jenkins plug-in. Two examples are:

- https://github.com/fescobar/allure-docker-service
- https://plugins.jenkins.io/allure-jenkins-plugin/

In this case, we would use the `logtalk_allure_report` script option to only perform the preprocessing step:

```
$ logtalk_allure_report -p
```

The script also supports passing *environment pairs*, which are displayed in the generated Allure reports in the environment pane. This feature can be used to pass e.g. the backend name and the backend version or git commit hash. The option syntax differs, however, between the two scripts. For example, using the Bash script:

```
$ logtalk_allure_report -- Backend='GNU Prolog' Version=1.5.0
```

Or:

```
$ logtalk_allure_report -- Project='Deep Thought' Commit=`git rev-parse --short HEAD`
```

In the case of the PowerShell script, the pairs are passed comma separated inside a string:

```
PS> logtalk_allure_report -e "Backend='GNU Prolog',Version=1.5.0"
```

Or:

```
PS> logtalk_allure_report -e "Project='Deep Thought',Commit=bf166b6"
```

To show test run trends in the report (e.g., when running the tests for each application source code commit), save the processed test results and the report data to permanent directories. For example:

```
$ logtalk_allure_report \
  -i "$HOME/my_project/allure-results" \
  -o "$HOME/my_project/allure-report"
$ allure open "$HOME/my_project/allure-report"
```

Note that Allure cleans the report directory when generating a new report. Be careful to always specify a dedicated directory to prevent accidental data loss.

The generated reports can include links to the tests source code. This requires using the `logtalk_tester` shell script option that allows passing the base URL for those links. This option needs to be used together with the option to suppress the tests directory prefix so that the links can be constructed by appending the tests file relative path to the base URL. For example, assuming that you want to generate a report for the tests included in the Logtalk distribution when using the GNU Prolog backend:

```
$ cd $LOGTALKUSER
$ logtalk_tester \
  -p gnu \
  -f xunit \
  -s "$LOGTALKUSER" \
  -u "https://github.com/LogtalkDotOrg/logtalk3/tree/3e4ea295986fb09d0d4aade1f3b4968e29ef594e
↪"
```

The use of a git hash in the base URL ensures that the generated links will always show the exact versions of the tests that were run. The links include the line number for the tests in the test files (assuming that the git repo is stored in a BitBucket, GitHub, or GitLab server). But note that not all supported backends provide accurate line numbers.

It's also possible to generate single-file reports. For example:

```
$ logtalk_allure_report -s -t "My Amazing Tests Report"
```

There are some caveats when generating Allure reports that users must be aware of. First, Allure expects test names to be unique across different tests sets. If there are two tests with the same name in two different test sets, only one of them will be reported. Second, when using the `xunit` format, dates are reported as MM/DD/YYYY. Finally, when using the `xunit_net_v2` format, tests are reported in a random order instead of their run order, and dates are displayed as "unknown" in the overview page.

5.13.33 Exporting code coverage results in XML format

To export code coverage results in XML format, load the `coverage_report.lgt` file before running the tests. A file named `coverage_report.xml` will be created in the same directory as the object defining the tests.

The XML file can be opened in most web browsers (with the notorious exception of Google Chrome) by copying to the same directory the `coverage_report.dtd` and `coverage_report.xsl` files found in the `tools/lgtunit` directory (when using the `logtalk_tester` script, these two files are copied automatically). In alternative, an XSLT processor can be used to generate an XHTML file instead of relying on a web browser for the transformation. For example, using the popular `xsltproc` processor:

```
$ xsltproc -o coverage_report.html coverage_report.xml
```

On Windows operating-systems, this processor can be installed using e.g. Chocolatey. On a POSIX operating-systems (e.g., Linux, macOS, ...) use the system package manager to install it if necessary.

The coverage report can include links to the source code when hosted on Bitbucket, GitHub, or GitLab. This requires passing the base URL as the value for the `url` XSLT parameter. The exact syntax depends on the XSLT processor, however. For example:

```
$ xsltproc \
  --stringparam url https://github.com/LogtalkDotOrg/logtalk3/blob/master \
  -o coverage_report.html coverage_report.xml
```

Note that the base URL should preferably be a permanent link (i.e., it should include the commit SHA1) so that the links to source code files and lines remain valid if the source code is later updated. It's also necessary to suppress the local path prefix in the generated `coverage_report.xml` file. For example:

```
$ logtalk_tester -c xml -s $HOME/logtalk/
```

Alternatively, you can pass the local path prefix to be suppressed to the XSLT processor (note that the `logtalk_tester` script suppresses the `$HOME` prefix by default):

```
$ xsltproc \
  --stringparam prefix logtalk/ \
  --stringparam url https://github.com/LogtalkDotOrg/logtalk3/blob/master \
  -o coverage_report.html coverage_report.xml
```

If you are using Bitbucket, GitHub, or GitLab hosted on your own servers, the `url` parameter may not contain a `bitbucket`, `github`, or `gitlab` string. In this case, you can use the XSLT parameter `host` to indicate which service you are running.

5.13.34 Automatically creating bug reports at issue trackers

To automatically create bug report issues for failed tests in GitHub or GitLab servers, see the `issue_creator` tool.

5.13.35 Minimizing test results output

To minimize the test results output, simply load the `minimal_output.lgt` file before running the tests. This file defines an object, `minimal_output`, that intercepts and summarizes the unit test execution messages.

5.13.36 Help with warnings

Load the `tutor` tool to get help with selected warnings printed by the `lgtunit` tool.

5.13.37 Known issues

Deterministic unit tests are currently not available when using Quintus Prolog as it lacks built-in support that cannot be sensibly defined in Prolog.

Parameter variables (`_VariableName_`) cannot currently be used in the definition of the `condition/1`, `setup/1`, and `cleanup/1` test options when using the `test/3` dialect. For example, the following condition will not work:

```
test(some_id, true, [condition(_ParVar_ == 42)]) :-
    ...
```

The workaround is to define an auxiliary predicate called from those options. For example:

```
test(check_xyz, true, [condition(xyz_condition)]) :-
    ...

xyz_condition :-
    _ParVar_ == 42.
```

5.14 `linter`

Logtalk provides a built-in linter tool that runs automatically when compiling and loading source files. The lint warnings are controlled by a set of flags. The default values for these flags are defined in the backend Prolog compiler adapter files and can be overridden from a settings file, from a source file (e.g., a loader file), or from an entity. These flags can be set globally using the set_logtalk_flag/2 built-in predicate. For (source file or entity) local scope, use instead the set_logtalk_flag/2 directive.

The linter flags can be managed as a group using the linter meta-flag. See the documentation for details.

Some lint checks are turned off by default, specially when computationally expensive. Still, it's a good idea to turn them on to check your code on a regular basis (e.g., in CI/CD pipelines).

Note that, in some cases, the linter may generate false warnings due to source code analysis limitations or special cases that, while valid when intended, usually result from programming issues. When a code rewrite is not a sensible solution to avoid the warning, the workaround is to turn off as locally as possible the flag that controls the warning.

5.14.1 Main linter checks

Lint checks include:

- Missing directives (including scope, meta-predicate, dynamic, discontiguous, and multifile directives)
- Duplicated directives, clauses, and grammar rules
- Missing predicates (unknown messages plus calls to non-declared and non-defined predicates)
- Calls to declared but not defined static predicates
- Non-terminals called as predicates (instead of via the `phrase/2-3` built-in methods)
- Predicates called as non-terminals (instead of via the `call//1` built-in method)
- Non-portable predicate calls, predicate options, arithmetic function calls, directives, flags, and flag values
- Missing arithmetic functions (with selected backends)
- Suspicious calls (syntactically valid calls that are likely semantic errors; e.g. float comparisons using the standard arithmetic comparison operators or comparing numbers using unification)
- Deprecated directives, predicates, arithmetic functions, control constructs, and flags
- References to unknown entities (objects, protocols, categories, or modules)
- Top-level shortcuts used as directives
- Unification goals that will succeed without binding any variables
- Unification goals that will succeed by creating a cyclic term
- Goals that are always true or always false
- Trivial goal failures (due to no matching predicate clause)
- Redefined built-in predicates
- Redefined standard operators
- Lambda expression unclassified variables and mixed up variables
- Lambda expression with parameter variables used elsewhere in a clause
- Singleton variables
- If-then-else and soft cut control constructs without an else part
- If-then-else and soft cut control constructs where the test is a unification between a variable and a ground term
- Missing parentheses around if-then-else and disjunction control constructs in the presence of cuts in the first argument
- Cuts in clauses for multifile predicates
- Missing cut in repeat loops
- Possible non-steadfast predicate definitions
- Non-tail recursive predicate definitions
- Redundant calls to control constructs and built-in predicates
- Calls to all-solutions predicates with existentially qualified variables not occurring in the qualified goal
- Calls to all-solutions predicates with no shared variables between template and goal

- Calls to `bagof/3` and `setof/3` where the goal argument contains singleton variables
- Calls to `findall/3` used to backtrack over all solutions of a goal without collecting them
- Calls to `catch/3` that catch all exceptions
- Calls to standard predicates that have more efficient alternatives
- Unsound calls in grammar rules
- File, entity, predicate, and variable names not following official coding guidelines
- Variable names that differ only on case
- Clauses whose body is a disjunction (and that can be rewritten as multiple clauses per coding guidelines)
- Naked meta-variables in cut-transparent control constructs
- Left-recursion in clauses and grammar rules

Additional lint checks are provided by the `lgtunit`, `lgtdoc`, `make`, and `dead_code_scanner` tools. For large projects, the data generated by the `code_metrics` tool may also be relevant in accessing code quality and suggesting code refactoring candidates.

5.14.2 Help on linter warnings

By loading the `tutor` tool, most lint warnings are expanded with explanations and suggestions on how to fix the reported issues. See also the coding guidelines for additional explanations.

5.14.3 Extending the linter

Experimental support for extending the linter with user-defined warnings is available using the logtalk_linter_hook/7 multifile hook predicate. For example, the `format` and `list` library objects define this hook predicate to lint calls to the `format/2-3` and `append/3` predicates for common errors and misuses.

5.14.4 Linting Prolog modules

This tool can also be applied to Prolog modules that Logtalk is able to compile as objects. For example, if the Prolog module file is named `module.pl`, try:

```
| ?- logtalk_load(module, [source_data(on)]).
```

Due to the lack of standardization of module systems and the abundance of proprietary extensions, this solution is not expected to work for all cases.

5.14.5 Linting plain Prolog files

This tool can also be applied to plain Prolog code. For example, if the Prolog file is named `code.pl`, simply define an object including its code:

```
:- object(code).
    :- include('code.pl').
:- end_object.
```

Save the object to an e.g. `code.lgt` file in the same directory as the Prolog file and then load it:

```
| ?- logtalk_load(code, [source_data(on)]).
```

In alternative, use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(code, [hook(object_wrapper_hook), source_data(on)]).
```

With either wrapping solution, pay special attention to any compilation warnings that may signal issues that could prevent the plain Prolog from being fully checked when wrapped by an object.

5.15 make

Logtalk provides a make tool supporting several targets using the `logtalk_make/0-1` built-in predicates. Top-level shortcuts for the targets are also provided.

5.15.1 API documentation

To consult the documentation of the `logtalk_make/0-1` built-in predicates, open in a web browser the links:

- ../refman/predicates/logtalk_make_0.html
- ../refman/predicates/logtalk_make_1.html

There is also a user-defined hook predicate that supports defining additional actions for the make targets (e.g., running tests automatically on make check or regenerating API documentation on make documentation):

- ../refman/predicates/logtalk_make_target_action_1.html

5.15.2 Help with warnings

Load the `tutor` tool to get help with selected warnings printed by the `make` tool.

5.16 packs

This tool provides predicates for downloading, installing, upgrading, and uninstalling third-party libraries and applications, generically known as *packs*. Collections of pack specifications are made available using *registries*. Registries can be local to a system, publicly shared, or private to a company (e.g., only available in a VPN). There is no concept of a central registry. Users can decide which registries they trust and want to use and add them using their published URLs. The tool supports both pack checksums and signatures and takes several steps to sanitize registry and pack specifications. As with other Logtalk developer tools, portability is a main goal. This tool can be used with any supported Prolog backend and run on both POSIX and Windows systems. Moreover, this tool can be used not only for handling Logtalk packs but also for Prolog only packs, thus providing a solution for sharing portable resources between multiple systems.

A list of public Logtalk and Prolog pack registries is available at:

https://github.com/LogtalkDotOrg/pack-registries

This tool is in the beta stage of development. Feedback is most welcome.

5.16.1 Requirements

On POSIX systems (Linux, macOS, ...), the following shell commands are required:

- `sha256sum` (provided by GNU `coreutils`)
- `curl` (default file downloader)
- `wget` (alternative file downloader)
- `bsdtar` (provided by `libarchive` or `libarchive-tools`)
- `gpg` (provided by `gnupg2`)
- `git`
- `direnv` (when using virtual environments)

The tool uses `bsdtar` instead of GNU `tar` so that it can uncompress `.zip` archives (`unzip` doesn't provide the desired options that allow a simple and reliable solution for ignoring the non-predictable name of the wrapper directory).

On Windows systems, the following shell commands are required:

- `certutil.exe`
- `curl.exe` (default file downloader)
- `wget.exe` (alternative file downloader)
- `tar.exe`
- `gpg.exe`
- `git.exe`
- `Set-PsEnv` (when using virtual environments)

In recent Windows 10 builds, only `wget`, `gpg`, `git`, and `Set-PsEnv` should require installation. You can download the GnuPG software from:

https://www.gnupg.org/

You can download Git from:

https://gitforwindows.org

You can download Wget from:

https://eternallybored.org/misc/wget/

You can also use Chocolatey to install the commands above:

```
> choco install gnupg git wget
```

To install Set-PsEnv from the PowerShell Gallery:

```
PS> Install-Module -Name Set-PsEnv
```

On macOS systems, Apple bundles both `curl` and BSD `tar` (under the name `tar`; you can simply create a `bsdtar` alias or install a more recent version). The required commands can be easily installed using MacPorts:

```
$ sudo port install coreutils wget libarchive gnupg2 git direnv
```

Or using Homebrew:

```
$ brew install coreutils wget libarchive gnupg2 git direnv
```

On Linux systems, use the distribution's own package manager to install any missing command. For example, in recent Ubuntu versions:

```
$ sudo apt update
$ sudo apt install coreutils curl wget libarchive-tools gnupg2 git direnv
```

5.16.2 API documentation

This tool API documentation is available at:

../../apis/library_index.html#packs

5.16.3 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(packs(loader)).
```

5.16.4 Testing

To run the tool tests, use the query:

```
| ?- logtalk_load(packs(tester)).
```

The tests can be run without interfering with the user packs setup.

5.16.5 Usage

The `packs` tool loads at startup all the currently defined registry and pack specifications (from the registries/packs storage directory; see below). When no registry/pack setup exists, a new one is automatically created.

The tool provides two main objects, `registries` and `packs`, for handling, respectively, registries and packs. Both objects accept a `help/0` message that describes the most common queries.

5.16.6 Registries and packs storage

The tool uses a directory specified using the `logtalk_packs` library alias when defined (in a settings file or in a backend Prolog initialization file). When this library alias is not defined, the tool uses the value of the `LOGTALKPACKS` environment variable when defined. Otherwise, it defaults to the `~/logtalk_packs` directory. The actual directory can be retrieved by the query:

```
| ?- packs::logtalk_packs(Directory).
...
```

This directory holds sub-directories for registries, packs, and archives. These sub-directories are automatically created when loading the `packs` tool if they don't exist. Users should not manually modify the contents of these directories. Multiple and independent registry/pack setups are possible using *virtual environments* as explained next.

Your registries and packs setup can be saved and restored (e.g., in a different system) by using the `packs::save/1-2` and `packs::restore/1-2` predicates, as explained in the next section about virtual environments. If necessary, before restoring, the `packs::reset/0` predicate can be called to delete any defined registries and installed packs.

5.16.7 Virtual environments

An application may require specific pack versions. These requirements may differ between applications. Different applications may also have conflicting requirements. Therefore, a *virtual environment* where an application requirements are fulfilled may be required to develop and/or run it. A virtual environment is essentially a registries/packs storage directory.

Defining the `logtalk_packs` library alias in a settings file or defining the `LOGTALKPACKS` environment variable before starting Logtalk allows easy creation and switching between virtual environments. By using a per-application settings file (or a per-application environment variable definition), each application can thus use its own virtual environment. The `settings.lgt` file can define the `logtalk_packs` library alias using code such as:

```
:- initialization((
    logtalk_load_context(directory, Directory),
    assertz(logtalk_library_path(logtalk_packs, Directory))
)).
```

The definition of the `logtalk_packs` library alias **must** always be an atom and thus never use library notation (i.e., it must never depend on other library aliases).

When a virtual environment also requires a specific Logtalk version (e.g., the version used to test and certify it), this can be installed as a pack from the official talkshow registry and used by (re)defining the `LOGTALKHOME` and `LOGTALKUSER` environment variables to point to its pack directory (which can be queried by using the `packs::directory/2` message).

Experimental `lgtenv.sh` and `lgtenv.ps1` scripts are included to simplify creating virtual environments. For example:

```
$ lgtenv -d ~/my_venv -c -p logtalk_packs
$ cd ~/my_venv
direnv: loading ~/my_venv/.envrc
direnv: export +LOGTALKPACKS
```

Type `lgtenv -h` for details on the script options.

These scripts require, respectively, direnv and Set-PsEnv to be installed. These utilities load and unload environment variables when changing the current directory. On Windows systems, when using the `lgtenv.ps1` script, you also need to redefine the PowerShell prompt in a profile file (e.g., `$HOME\Documents\PowerShell\Profile.ps1`) to mimic the functionality of `direnv` of automatically loading an existing `.env` file when changing to its directory. For example:

```
function prompt {
    Set-PsEnv
    'PS ' + $(Get-Location) + '> '
}
```

A virtual environment setup (i.e., the currently defined registries and installed packs) can be saved into a file (e.g., `requirements.lgt`) using the `packs::save/1` predicate:

```
| ?- packs::save('requirements.lgt').
...
```

This query saves a listing of all the installed packs and their registries. Using the saved file, the virtual environment setup can then be restored using the `packs::restore/1-2` predicates. The file uses a simple format with `registry/2`, `pack/3`, `pinned_registry/1`, and `pinned_pack/1` facts (in this order) and can be manually created or edited if necessary. For example:

```
registry(talkshow, 'https://github.com/LogtalkDotOrg/talkshow.git').
pack(talkshow, logtalk, 3:45:0).
pack(talkshow, lflat, 2:1:0).
```

These files can be distributed with applications so that users can easily fulfill application requirements by running the query once:

```
| ?- packs::restore('requirements.lgt').
```

Subsequently, the application `loader.lgt` file can then load the required packs using their loader files:

```
:- initialization((
    % load required packs
    logtalk_load(foo(loader)),
    logtalk_load(bar(loader)),
    ...
    % load application files
    ...
)).
```

Note that restoring encrypted registries or encrypted packs requires entering the required passphrases. Although the `restore/2` predicate accepts a list of options that include the `gpg/1` option, this only allows specifying a single and common passphrase when interactive entering of passphrases is not convenient or possible.

5.16.8 Registry specification

A registry is a git remote repo that can be cloned, a downloadable or local archive, or a local directory containing a Logtalk loader file that loads source files defining the registry itself and the packs it provides. The registry name is ideally a valid unquoted atom. The registry directory must contain at least two Logtalk source files:

- A file defining an object named after the registry with a `_registry` suffix, implementing the `registry_protocol`. This naming convention helps prevent name conflicts.
- A loader file (named `loader.lgt` or `loader.logtalk`) that loads the registry object file and all pack object files.

An example of a registry specification object would be:

```
:- object(jdoe_awesome_packs_registry,
    implements(registry_protocol)).

    :- info([
```

(continues on next page)

(continued from previous page)

```
        version is 1:0:0,
        author is 'John Doe',
        date is 2021-10-18,
        comment is 'John Doe awesome packs registry spec.'
    ]).

    name(jdoe_awesome_packs).

    description('John Doe awesome packs').

    home('https://example.com/jdoe_awesome_packs').

    clone('https://github.com/jdoe/jdoe_awesome_packs.git').

    archive('https://github.com/jdoe/jdoe_awesome_packs/archive/main.zip').

:- end_object.
```

Optionally, the registry object can also define a `note(Action, Note)` predicate. The `Action` argument is an atom: `add`, `update`, or `delete`. The `Note` argument is also an atom. The tool will print any available notes when executing one of the registry actions. See the `registry_protocol` documentation for more details.

The registry directory should also contain `LICENSE` and `README.md` files (individual packs can use a different license, however). The path to the `README.md` file is printed when the registry is added. It can also be queried using the `registries::directory/2` predicate. The `NOTES.md` file name can also be used in alternative to the recommended `README.md` file name.

Summarizing the required directory structure using the above example (note that the registry and pack specification files are named after the objects):

```
jdoe_awesome_packs
    LICENSE
    README.md
    jdoe_awesome_packs_registry.lgt
    loader.lgt
    foo_pack.lgt
    bar_pack.lgt
    ...
```

With the contents of the `loader.lgt` file being:

```
:- initialization((
    logtalk_load(jdoe_awesome_packs_registry),
    logtalk_load(foo_pack),
    logtalk_load(bar_pack),
    ...
)).
```

It would be, of course, possible to have all objects in a single source file. But having a file per-object and a loader file helps maintenance, and it's also a tool requirement for applying safety procedures to the source file contents and thus successfully loading the registry and pack specs.

As registries are git repos in the most common case, and thus adding them performs a git repo cloning, they should only contain the strictly required files.

5.16.9 Registry handling

Registries can be added using the `registries::add/1-3` predicates, which take a registry URL. Using the example above:

```
| ?- registries::add('https://github.com/jdoe/jdoe_awesome_packs.git').
```

HTTPS URLs must end with either a `.git` extension or an archive extension (same valid extensions as for pack archives, including gpg encrypted). Git cloning URLs are preferred as they simplify updating registries. But a registry can also be made available via a local directory (using a `file://` URL) or a downloadable archive (using a `https://` URL).

For registries made available using an archive, the `registries::add/2-3` predicates **must** be used as the registry name cannot in general be inferred from the URL basename or from the archived directory name. The registry argument must also be the declared registry name in the registry specification object. For example:

```
| ?- registries::add(
        jdoe_awesome_packs,
        'https://github.com/jdoe/jdoe_awesome_packs/archive/main.zip'
    ).
```

When a registry may be already defined, you can use the `update(true)` option to ensure that the registry will be updated to its latest definition:

```
| ?- registries::add(
        jdoe_awesome_packs,
        'https://github.com/jdoe/jdoe_awesome_packs/archive/main.zip',
        [update(true)]
    ).
```

The added registries can be listed using the `registries::list/0` predicate:

```
| ?- registries::list.

% Defined registries:
%   jdoe_awesome_packs (git)
%   ...
```

The `registries::describe/1` predicate can be used to print the details of a registry:

```
| ?- registries::describe(jdoe_awesome_packs).

% Registry:    jdoe_awesome_packs
% Description: John Doe awesome packs
% Home:        https://example.com/jdoe_awesome_packs
% Cloning URL: https://github.com/jdoe/jdoe_awesome_packs.git
% Archive URL: https://github.com/jdoe/jdoe_awesome_packs/archive/main.zip
```

To update all registries, use the `registries::update/0` predicate. To update a single registry, use the `registries::update/1-2` predicates. After updating, you can use the `packs::outdated/0-1` predicates to list any outdated packs.

Registries can also be deleted using the `registries::delete/1-2` predicate. By default, any registries with installed packs cannot be deleted. If you force deletion (by using the `force(true)` option), you can use the `packs::orphaned/0` predicate to list any orphaned packs that are installed.

See the tool API documentation on the registries object for other useful predicates.

5.16.10 Registry development

To simplify registry development and testing, use a local directory and a `file://` URL when calling the `registries::add/1` predicate. For example:

```
| ?- registries::add('file:///home/jdoe/work/my_pack_collection').
```

If the directory is a git repo, the tool will clone it when adding it. Otherwise, the files in the directory are copied to the registry definition directory. This allows the registry to be added and deleted without consequences for the original registry source files.

To check your registry specifications, use the `registries::lint/0-1` predicates after adding the registry.

5.16.11 Pack specification

A pack is specified using a Logtalk source file defining an object that implements the `pack_protocol`. The source file should be named after the pack with a `_pack` suffix. This naming convention helps prevent name conflicts, notably with the pack's own objects. The file must be available from a declared pack registry (by having the registry loader file loading it). The pack name is preferably a valid unquoted atom. An example of a pack specification object would be:

```
:- object(lflat_pack,
    implements(pack_protocol)).

    :- info([
        version is 1:0:0,
        author is 'Paulo Moura',
        date is 2021-10-18,
        comment is 'L-FLAT - Logtalk Formal Language and Automata Toolkit pack spec.'
    ]).

    name(lflat).

    description('L-FLAT - Logtalk Formal Language and Automata Toolkit').

    license('MIT').

    home('https://github.com/l-flat/lflat').

    version(
        2:1:0,
        stable,
        'https://github.com/l-flat/lflat/archive/refs/tags/v2.1.0.tar.gz',
        sha256 - '9c298c2a08c4e2a1972c14720ef1498e7f116c7cd8bf7702c8d22d8ff549b6a1',
        [logtalk @>= 3:42:0],
        all
    ).

    version(
        2:0:2,
```

(continues on next page)

(continued from previous page)

```
        stable,
        'https://github.com/l-flat/lflat/archive/refs/tags/v2.0.2.tar.gz',
        sha256 - '8774b3863efc03bb6c284935885dcf34f69f115656d2496a33a446b6199f3e19',
        [logtalk @>= 3:36:0],
        all
    ).

:- end_object.
```

The `license/1` argument must be an atom and should, whenever possible, be a license identifier as specified in the SPDX standard.

Optionally, the pack object can also define a `note(Action, Version, Note)` predicate. The `Action` argument is an atom: `install`, `update`, or `uninstall`. The `Note` argument is also an atom. The tool will print any available notes when executing one of the registry actions. See the `pack_protocol` documentation for more details.

The pack sources must be available either as a local directory (when using a `file://` URL) or for downloading as a supported archive. The checksum for the archive must use the SHA-256 hash algorithm (`sha256`). The pack may optionally be signed. Supported archive formats and extensions are:

- `.zip`
- `.tgz`, `.tar.gz`
- `.tbz2`, `.tar.bz2`

Also, for encrypted packs, all the extensions above with a `.gpg` suffix (e.g., `.zip.gpg`).

The pack sources should contain `LICENSE`, `README.md` (or `NOTES.md`), and `loader.lgt` (or `loader.logtalk`) files. Ideally, it should also contain a `tester.lgt` (`tester.logtalk`) file. The path to the `README.md` file is printed when the pack is installed or updated. It can also be queried using the `packs::directory/2` predicate.

5.16.12 Encrypted packs

Packs can be gpg encrypted, with a choice of passphrase-based encryption, key-based encryption, or both. Encrypted pack archives must always have a `.gpg` extension. For example, to encrypt a pack archive with a symmetric cipher using a passphrase:

```
$ tar -cvzf - my_pack | gpg -c --cipher-algo AES256 > v1.2.1.tar.gz.gpg
```

In this case, the passphrase would need to be securely communicated to any users installing or updating the pack.

See the gpg documentation for full details on encrypting and decrypting archives. If you get a "gpg: problem with the agent: Inappropriate ioctl for device" error message with the command above, try:

```
$ export GPG_TTY=$(tty)
```

5.16.13 Signed packs

Packs can be gpg signed. Detached signature files are assumed and expected to share the name of the archive and use `.asc` or `.sig` extensions. For example, if the pack archive name is `v1.0.0.tar.gz`, the signature file must be named `v1.0.0.tar.gz.asc` or `v1.0.0.tar.gz.sig`. When the `checksig(true)` option is used, the signature file is automatically downloaded using a URL constructed from the pack archive URL. When both `.asc` and `.sig` files exist, the `.asc` file is used. An example of signing a pack and creating the `.asc` file (assuming the default signing key) is:

```
$ gpg --armor --detach-sign v1.0.0.tar.gz
```

To create instead a `.sig` file:

```
$ gpg --detach-sign v1.0.0.tar.gz
```

See the gpg documentation for full details on signing archives and sharing the public keys required to verify the signatures.

5.16.14 Pack URLs and Single Sign-On

Typically, pack archive download URLs are HTTPS URLs and handled using `curl`. It's also possible to use `git archive` to download pack archives, provided that the server supports it (as of this writing, Bitbucket and GitLab public hosting services support it but not GitHub). Using `git archive` is specially useful when the packs registry is hosted on a server using Single Sign-On (SSO) for authentication. In this case, HTTPS URLs can only be handled by `curl` by passing a token (see below for an example). When the user has setup SSH keys to authenticate to the packs registry server, `git archive` simplifies pack installation, providing a better user experience. For example:

```
version(
    1:0:1,
    stable,
    'git@gitlab.com:me/foo.git/v1.0.1.zip',
    sha256 - '0894c7cdb8968b6bbcf00e3673c1c16cfa98232573af30ceddda207b20a7a207',
    [logtalk @>= 3:36:0],
    all
).
```

The pseudo-URL must be the concatenation of the SSH repo cloning URL with the archive name. The archive name must be the concatenation of a valid tag with a supported archive extension. SSH repo cloning URLs use the format:

```
git@<hostname>:path/to/project.git
```

They can usually be easily copied from the hosting service repo webpage. To compute the checksum, you must first download the archive. For example:

```
$ git archive --output=foo-v1.0.1.zip --remote=git@gitlab.com:me/foo.git v1.0.1
$ openssl sha256 foo-v1.0.1.zip
```

Be sure to use a format that is supported by both the `packs` tool and the `git archive` command (the format is inferred from the `--output` option). Do not download the archive from the web interface of the git hosting service in order to compute the checksum. Different implementations of the archiving and compressing algorithms may be used, resulting in mismatched checksums.

Users installing packs available using `git archive` URLs are advised to run a SSH agent to avoid being prompted for passwords when installing or updating packs. They must also upload their SSH public keys to the pack provider hosts.

5.16.15 Multiple pack versions

A pack may specify multiple versions. Each version is described using a `version/6` predicate clause as illustrated in the example above. The versions must be listed in order from newest to oldest. For details, see the `pack_protocol` API documentation.

Listing multiple versions allows the pack specification to be updated (by updating its registry) without forcing existing users into installing (or updating to) the latest version of the pack. It allows different applications depending on different pack versions to continue to be built and deployed.

The pack version is complemented by the pack status. Valid values are `stable`, `rc`, `beta`, `alpha`, `experimental`, and `deprecated`. Packs with a `experimental` or `deprecated` status are **never** installed by default when using the install and update predicates unless their version is explicitly specified. When updating packs, we can restrict the valid status of the updates using the `status/1` option. For example, we can ensure that we only update to new stable pack versions by using the option `status([stable])`.

5.16.16 Pack dependencies

Pack dependencies on other packs can be specified using a list of `Registry::Pack Operator Version` terms where `Operator` is a standard term comparison operator:

- `Registry::Pack @>= Version` - the pack requires a dependency with a version equal or above the specified one. For example, `logtalk @>= 3:36:0` means that the pack requires Logtalk 3.36.0 or a later version.
- `Registry::Pack @=< Version` - the pack requires a dependency with a version up to the specified one. For example, `common::bits @=< 2:1` means that the pack requires a `common::bits` pack up to 2.1. This includes all previous versions and also all patches for version 2.1 (e.g., 2.1.7, 2.1.8, ...) but not version 2.2 or newer.
- `Registry::Pack @< Version` - the pack requires a dependency with version older than the specified one. For example, `common::bits @< 3` means that the pack requires a `common::bits` 2.x or older version.
- `Registry::Pack @> Version` - the pack requires a dependency with version newer than the specified one. For example, `common::bits @> 2:4` means that the pack requires a `common::bits` 2.5 or newer version.
- `Registry::Pack == Version` - the pack requires a dependency with a specific version. For example, `common::bits == 2:1` means that the pack requires a `common::bits` pack version 2.1.x (thus, from version 2.1.0 to the latest patch for version 2.1).
- `Registry::Pack \== Version` - the pack requires a dependency with any version other than the one specified. For example, `common::bits \== 2.1` means that the pack requires a `common::bits` pack version other than any 2.1.x version.

To specify *range* dependencies by using two consecutive elements with the lower bound followed by the upper bound. For example, `common::bits @>= 2, common::bits @< 3` means all `common::bits` 2.x versions but not older or newer major versions.

It's also possible to specify *alternative* dependencies using the `(;)/2` operator. For example, `(common::bits == 1:9; common::bits @>= 2:3)` means either `common::bits` 1.9.x versions or 2.3.x and later versions. Alternatives should be listed in decreasing order of preference.

When a pack also depends on a Logtalk or backend version, the name `logtalk` or the backend identifier atom can be used in place of `Registry::Pack` (see below for the table of backend specifiers). For example, `logtalk @>= 3.36.0`.

When a pack also depends on an operating-system version (e.g., a pack containing shared libraries with executable code), the `os(Name,Machine)` compound term can also be used in place of `Registry::Pack`. For example, `os('Darwin',x86_64) @>= '23.0.0'`. Note that, in this case, the release is an atom. The operating-system data (name, machine, and release) is queried using the corresponding `os` library predicates (see the library documentation for details).

5.16.17 Pack portability

Ideally, packs are fully portable and can be used with all Logtalk-supported Prolog backends. This can be declared by using the atom `all` in the last argument of the `version/6` predicate (see example above).

When a pack can only be used with a subset of the Prolog backends, the last argument of the `version/6` predicate is a list of backend identifiers (atoms):

- B-Prolog: `b`
- Ciao Prolog: `ciao`
- CxProlog: `cx`
- ECLiPSe: `eclipse`
- GNU Prolog: `gnu`
- JIProlog: `ji`
- XVM: `xvm`
- Quintus Prolog: `quintus`
- SICStus Prolog: `sicstus`
- SWI-Prolog: `swi`
- Tau Prolog: `tau`
- Trealla Prolog: `trealla`
- XSB: `xsb`
- YAP: `yap`

5.16.18 Pack development

To simplify pack development and testing, define a local registry and add to it a pack specification with the development version available from a local directory. For example:

```
version(
    0:11:0,
    beta,
    'file:///home/jdoe/work/my_awesome_library',
    none,
    [],
    all
).
```

If the directory is a git repo, the tool will clone it when installing the pack. Otherwise, the files in the directory are copied to the pack installation directory. This allows the pack to be installed, updated, and uninstalled without consequences for the pack source files.

You can also use a local archive instead of a directory. For example:

```
version(
    1:0:0,
    stable,
    'file:///home/jdoe/work/my_awesome_library/v1.0.0.tar.gz',
    sha256 - '1944773afba1908cc6194297ff6b5ac649a844ef69a69b2bcdf267cfa8bfce1e',
    [],
    all
).
```

Packs that are expected to be fully portable should always be checked by loading them with the `portability` flag set to `warning`.

To check your pack manifest files, use the `packs::lint/0-2` predicates after adding the registry that provides the packs.

5.16.19 Pack handling

Packs must be available from a defined registry. To list all packs that are available for installation, use the `packs::available/0` predicate:

```
| ?- packs::available.
```

To list all installed packs, call the `packs::installed/0` predicate:

```
| ?- packs::installed.
```

To list only the installed packs from a specific registry, call instead the `packs::installed/1` predicate. For example:

```
| ?- packs::installed(talkshow).
```

To know more about a specific pack, use the `packs::describe/1-2` predicates. For example:

```
| ?- packs::describe(bar).
```

The `packs::describe/2` predicate can be used when two or more registries provide packs with the same name. For example:

```
| ?- packs::describe(reg, bar).
```

To install the latest version of a pack, we can use the `packs::install/1-4` predicates. In the most simple case, when a pack name is unique among registries, we can use the `packs::install/1` predicate. For example:

```
| ?- packs::install(bar).
```

Any pack dependencies are also checked and installed or updated if necessary. Other install predicates are available to disambiguate between registries and to install a specific pack version.

Packs become available for loading immediately after successful installation (no restarting of the Logtalk session is required). For example, after the pack `bar` is installed, you can load it at the top-level by typing:

```
| ?- {bar(loader)}.
```

or load it from a loader file using the goal `logtalk_load(bar(loader))`.

After updating the defined registries, outdated packs can be listed using the `packs::outdated/0` predicate. You can update all outdated packs by calling the `packs::update/0` predicate or update a single pack using the `packs::update/1-2` predicates. For example:

```
| ?- packs::update(bar).
```

By default, updating a pack fails if it would break any dependent pack (the `force(true)` option, described below, can be used to force updating in this case).

The tool provides versions of the pack install, update, and uninstall predicates that accept a list of options:

- `verbose(Boolean)` (default is `false`)
- `clean(Boolean)` (default is `false`)
- `update(Boolean)` (default is `false`)
- `force(Boolean)` (default is `false`)
- `compatible(Boolean)` (default is `true`)
- `checksum(Boolean)` (default is `true`)
- `checksig(Boolean)` (default is `false`)
- `git(Atom)` (extra command-line options; default is `''`)
- `downloader(Atom)` (downloader utility; default is `curl`)
- `curl(Atom)` (extra command-line options; default is `''`)
- `wget(Atom)` (extra command-line options; default is `''`)
- `gpg(Atom)` (extra command-line options; default is `''`)
- `tar(Atom)` (extra command-line options; default is `''`)

Note that, by default, only compatible packs can be installed. To install a pack that is incompatible with the current Logtalk version, backend version, or operating-system version, use the `install/4` or `update/3` predicates with the option `compatible(false)`.

When installing large packs over unreliable network conditions, you may try switching the default downloader utility from `curl` to `wget`.

When a pack may already be installed, you can use the `update(true)` option to ensure that the installation will be updated to the specified version:

```
| ?- packs::install(reg, bar, 1:1:2, [update(true)]).
```

When using a `checksig(true)` option to check a pack signature, it is strongly advised that you also use the `verbose(true)` option. For example:

```
| ?- packs::install(reg, bar, 1:1:2, [verbose(true), checksig(true)]).
```

Note that the public key used to sign the pack archive must already be present in your local system.

Downloading pack archives may require passing extra command-line options to `curl` for authentication. A common solution is to use a personal access token. The details depend on the server software. An example when using GitHub:

```
| ?- packs::install(reg, bar, 1:1:2, [curl('--header "Authorization: token foo42"')]).
```

Another example when using GitLab:

```
| ?- packs::install(reg, bar, 1:1:2, [curl('--header "PRIVATE-TOKEN: foo42"')]).
```

Pack archives may be gpg encrypted. Encryption can be passphrase-based, key-based, or both. When using only passphrase-based encryption, the archive passphrase must be entered (if not cached) when installing or updating a pack. In this case, the passphrase can be entered interactively or using the `gpg/1` option. For example:

```
| ?- packs::install(reg, bar, 1:1:2, [gpg('--batch --passphrase test123')]).
```

See the gpg documentation for details. When using the `gpg/1` option, you should be careful to not leak passphrases in, e.g., the query history.

To uninstall a pack that you no longer need, use the `packs::uninstall/1-2` predicates. By default, only packs with no dependent packs can be uninstalled. You can print or get a list of the packs that depend on a given pack by using the `packs::dependents/1-3` predicates. For example:

```
| ?- packs::dependents(reg, bar, Dependents).
```

See the tool API documentation on the packs object for other useful predicates.

5.16.20 Pack documentation

The path to the pack `README.md` file is printed when the pack is installed or updated. It can also be retrieved at any time by using the `readme/2` predicate. For example:

```
| ?- packs::readme(lflat, Path).
```

Additional documentation may also be available from the pack home page, which can be printed by using the `describe/1-2` predicates. For example:

```
| ?- packs::describe(lflat).

% Registry:    ...
% Pack:        lflat
% Description: L-FLAT - Logtalk Formal Language and Automata Toolkit
% License:     MIT
% Home:        https://github.com/l-flat/lflat
% Versions:
...
```

The pack API documentation can be generated using the `lgtdoc` tool library and directory predicates (depending on the pack source files organization). For example:

```
| ?- {lflat(loader)},
     {lgtdoc(loader)},
     logtalk::expand_library_path(lflat, Path),
     lgtdoc::rdirectory(Path).
...
```

This query creates a `xml_docs` directory in the current directory. The XML documentation files can then be converted into a final format, e.g., HTML, using one of the scripts provided by the `lgtdoc` tool. For example:

```
$ cd xml_docs
$ lgt2html
```

For more details and alternatives, see the `lgtdoc` tool documentation.

It is also possible to add API documentation and diagrams for all the installed packs to the Logtalk distribution API documentation and diagrams by calling the `build` and `update_svg_diagrams` scripts in the `docs/apis/sources` directory with the `-i` option. See the scripts documentation for more details.

5.16.21 Pinning registries and packs

Registries and packs can be *pinned* after installation to prevent accidental updating or deleting, e.g., when using the batch `update/0` predicate. This is useful when your application requires a specific version or for security considerations (see below). For example, if we want the `bar` pack to stay at its current installed version:

```
| ?- packs::pin(bar).
yes
```

After, any attempt to update or uninstall the pack will fail with an error message:

```
| ?- packs::update(bar).
!     Cannot update pinned pack: bar
no

| ?- packs::uninstall(bar).
!     Cannot uninstall pinned pack: bar
no
```

To enable the pack to be updated or uninstalled, the pack must first be unpinned. Alternatively, the `force(true)` option can be used. Note that if you force update a pinned pack, the new version will be unpinned.

It's also possible to pin (or unpin) all defined registries or installed packs at once by using the `pin/0` (or `unpin/0`) predicates. But note that registries added after or packs installed after will not be automatically pinned.

5.16.22 Testing packs

Logtalk packs (as most Logtalk libraries, tools, and examples) are expected to have a `tester.lgt` or `tester.logtalk` tests driver file at the root of their directory, which can be used for both automated and manual testing. For example, after installing the `foo` pack:

```
| ?- {foo(tester)}.
```

To test all installed packs, you can use the `logtalk_tester` automation script from the installed packs directory, which you can query using the goal:

```
| ?- packs::prefix(Directory).
```

Note that running the packs tests, like simply loading the pack, can result in calling arbitrary code, which can potentially harm your system. Always take into account the security considerations discussed below.

5.16.23 Security considerations

New pack registries should be examined before being added, specially if public and from a previously unknown source. The same precautions should be taken when adding or updating a pack. Note that a registry can always index third-party packs.

Pack checksums are checked by default. But pack signatures are only checked if requested, as packs are often unsigned. Care should be taken when adding public keys for pack signers to your local system.

Registry and pack spec files, plus the registry loader file, are compiled by term-expanding them so that only expected terms are actually loaded and only expected `logtalk_load/2` goals with expected relative file paths are allowed. Predicates defining URLs are discarded if the URLs are neither `https://` nor `file://` URLs or if they contain non-allowed characters (currently, only alpha-numeric ASCII characters plus the ASCII `/`, `.`, `-`, and `_` characters are accepted). But note that this tool makes no attempt to audit pack source files themselves.

Registries and packs can always be pinned so that they are not accidentally updated to a version that you may not have had the chance to audit.

5.16.24 Best practices

- Make available a new pack registry as a git repo. This simplifies updating the registry and rolling back to a previous version.
- Use registry and pack names that are valid unquoted atoms, thus simplifying usage. Use descriptive names with underscores if necessary to link words.
- Name the registry and pack specification objects after their names with a `_registry` or `_pack` suffix. Save the objects in files named after the objects.
- Create new pack versions from git tags.
- If the sources of a pack are available from a git repo, consider using signed commits and signed tags for increased security.
- When a new pack version breaks backwards compatibility, list both the old and the new versions on the pack specification file.
- Pin registries and packs when specific versions are critical for your work so that you can still easily batch update the remaining packs and registries.
- Include the `$LOGTALKPACKS` directory (or the default `~/logtalk_packs` directory) on your regular backups.

5.16.25 Installing Prolog packs

This tool can also be used to install Prolog packs that don't use Logtalk. After installing a `pl_pack` Prolog pack from a `pl_reg` registry, it can be found in the `$LOGTALKPACKS/packs/pl_reg/pl_pack` directory. When the `LOGTALKPACKS` environment variable is not defined, the pack directory is by default `~/logtalk_packs/packs/pl_reg/pl_pack`.

Different Prolog systems provide different solutions for locating Prolog code. For example, several Prolog systems adopted the Quintus Prolog `file_search_path/2` hook predicate. For these systems, a solution could be to add a fact to this predicate for each installed Prolog pack. For example, assuming a `pl_pack` Prolog pack:

```
:- multifile(file_search_path/2).
:- dynamic(file_search_path/2).

file_search_path(library, '$LOGTALKPACKS/packs/pl_pack').
```

If the Prolog system also supports reading an initialization file at startup, the above definition could be added there.

5.16.26 Help with warnings

Load the `tutor` tool to get help with selected warnings printed by the `packs` tool.

5.16.27 Known issues

Using the `verbose(true)` option on Windows systems may not provide the shell commands output depending on the backend.

On Windows systems, the reset, delete, and uninstall predicates may fail to delete all affected folders and files due to a operating-system bug. Depending on the backend, this bug may cause some of the tests to fail. For details on this bug, see:

https://github.com/microsoft/terminal/issues/309

The workaround is to use the Windows File Explorer to delete the leftover folders and files.

When using Ciao Prolog 1.20.0, a workaround is used for this system non-standard support for multifile predicates.

When using GNU Prolog 1.5.0 as the backend on Windows, you may get an error on `directory_files/2` calls. For details and a workaround, see:

https://github.com/didoudiaz/gprolog/issues/4

This issue is fixed in the latest GNU Prolog git version.

Using SICStus Prolog as the backend on Windows doesn't currently work in version 4.7.0 and earlier versions. The underlying issues are fixed in the SICStus Prolog 4.7.1 version.

XSB has an odd bug (likely in its parser) when reading files that may cause a pack installed version to be reported as the `end_of_file` atom.

Some tests fail on Windows when using ECLiPSe or XSB due to file path representation issues.

5.17 `ports_profiler`

This tool counts and reports the number of times each port in the *procedure box model* is traversed during the execution of queries. It can also report the number of times each clause (or grammar rule) is used. It is inspired by the ECLiPSe `port_profiler` tool.

The procedure box model is the same as the one used in the debugger tool. This is an extended version of the original Byrd's four-port model. Besides the standard `call`, `exit`, `fail`, and `redo` ports, Logtalk also defines two post-unification ports, `fact` and `rule`, and an `exception` port. This tool can also distinguish between deterministic exits (reported in the `exit` column in the profiling result tables) and exits that leave choice-points (reported in the `*exit` column).

5.17.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#ports-profiler

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.17.2 Loading

```
| ?- logtalk_load(ports_profiler(loader)).
```

5.17.3 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(ports_profiler(tester)).
```

5.17.4 Compiling source files for port profiling

To compile source files for port profiling, simply compile them in debug mode and with the `source_data` flag turned on. For example:

```
| ?- logtalk_load(my_source_file, [debug(on), source_data(on)]).
```

Alternatively, you can also simply turn on the `debug` and `source_data` flags globally before compiling your source files:

```
| ?- set_logtalk_flag(debug, on), set_logtalk_flag(source_data, on).
```

Be aware, however, that loader files (e.g., library loader files) may override the default flag values, and thus the loaded files may not be compiled in debug mode. In this case, you will need to modify the loader files themselves.

5.17.5 Generating profiling data

After loading this tool and compiling the source files that you want to profile in debug mode, simply call the `ports_profiler::start` goal followed by the goals to be profiled. Use the `ports_profiler::stop` goal to stop profiling.

Note that the `ports_profiler::start/0` predicate implicitly selects the `ports_profiler` tool as the active debug handler. If you have additional debug handlers loaded (e.g., the `debugger` tool), those would no longer be active (there can be only one active debug handler at any given time).

5.17.6 Printing profiling data reports

After calling the goals that you want to profile, you can print a table with all profile data by typing:

```
| ?- ports_profiler::data.
```

To print a table with data for a single entity, use the query:

```
| ?- ports_profiler::data(Entity).
```

To print a table with data for a single entity predicate, use the query:

```
| ?- ports_profiler::data(Entity, Predicate).
```

In this case, the second argument must be either a predicate indicator, `Name/Arity`, or a non-terminal indicator, `Name//Arity`.

The profiling data can be reset using the query:

```
| ?- ports_profiler::reset.
```

To reset only the data for a specific entity, use the query:

```
| ?- ports_profiler::reset(Entity).
```

To illustrate the tool output, consider the `family` example in the Logtalk distribution:

```
| ?- {ports_profiler(loader)}.
...
yes

| ?- set_logtalk_flag(debug, on).
yes

| ?- logtalk_load(family(loader)).
...
yes

| ?- ports_profiler::start.
yes

| ?- addams::sister(Sister, Sibling).
Sister = wednesday,
Sibling = pubert ;
Sister = wednesday,
Sibling = pugsley ;
Sister = wednesday,
Sibling = pubert ;
Sister = wednesday,
Sibling = pugsley ;
no

| ?- ports_profiler::data.
------------------------------------------------------------------------
Entity     Predicate    Fact  Rule  Call  Exit *Exit  Fail  Redo Error
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------------------
addams     female/1        2     0     1     1     1     0     1     0
addams     parent/2        8     0     4     3     5     1     5     0
relations  sister/2        0     1     1     0     4     1     4     0
------------------------------------------------------------------------
yes

| ?- ports_profiler::data(addams).
------------------------------------------------------------
Predicate    Fact  Rule  Call  Exit *Exit  Fail  Redo Error
------------------------------------------------------------
female/1        2     0     1     1     1     0     1     0
parent/2        8     0     4     3     5     1     5     0
------------------------------------------------------------
yes

| ?- ports_profiler::data(addams, parent/2).
-------------
Clause  Count
-------------
     1      1
     2      1
     3      2
     4      1
     5      1
     6      2
-------------
yes
```

5.17.7 Interpreting profiling data

Some useful information that can be inferred from the profiling data include:

- which predicates are called more often (from the `call` port)
- unexpected failures (from the `fail` port)
- unwanted non-determinism (from the `*exit` port)
- performance issues due to backtracking (from the `*exit` and `redo` ports)
- predicates acting like a generator of possible solutions (from the `*exit` and `redo` ports)
- inefficient indexing of predicate clauses (from the `fact`, `rule`, and `call` ports)
- clauses that are never used or seldom used

The profiling data should be analyzed by taking into account the expected behavior for the profiled predicates.

5.17.8 Profiling Prolog modules

This tool can also be applied to Prolog modules that Logtalk is able to compile as objects. For example, if the Prolog module file is named `module.pl`, try:

```
| ?- logtalk_load(module, [debug(on), source_data(on)]).
```

Due to the lack of standardization of module systems and the abundance of proprietary extensions, this solution is not expected to work for all cases.

5.17.9 Profiling plain Prolog code

This tool can also be applied to plain Prolog code. For example, if the Prolog file is named `code.pl`, simply define an object including its code and declaring as public any predicates that you want to use as messages to the object. For example:

```
:- object(code).

    :- public(foo/2).
    :- include('code.pl').

:- end_object.
```

Save the object to an e.g. `code.lgt` file in the same directory as the Prolog file and then load it in debug mode:

```
| ?- logtalk_load(code, [debug(on), source_data(on)]).
```

In alternative, use the `object_wrapper_hook` provided by the `hook_objects` library:

```
| ?- logtalk_load(hook_objects(loader)).
...

| ?- logtalk_load(
        code,
        [hook(object_wrapper_hook), debug(on),
         source_data(on), context_switching_calls(allow)]
    ).
```

In this second alternative, you can then use the `(<<)/2` context switch control construct to call the wrapped predicates. E.g.

```
| ?- code<<foo(X, Y).
```

With either wrapping solution, pay special attention to any compilation warnings that may signal issues that could prevent the plain Prolog code from working as-is when wrapped by an object. Often any required changes are straightforward (e.g., adding `use_module/2` directives for called module library predicates).

5.17.10 Known issues

Determinism information is currently not available when using Quintus Prolog as the backend compiler.

5.18 `profiler`

This tool contains simple wrappers for selected Prolog profiler tools.

5.18.1 Loading

This tool can be loaded using the query:

```
?- logtalk_load(profiler(loader)).
```

For sample queries, please see the `SCRIPT.txt` file in the tool directory.

5.18.2 Testing

To test this tool, load the `tester.lgt` file:

```
| ?- logtalk_load(profiler(tester)).
```

5.18.3 Supported backend Prolog compilers

Currently, this tool supports the profilers provided with SICStus Prolog 4, SWI-Prolog, and YAP. The tool includes two files:

- `yap_profiler.lgt`
  simple wrapper for the YAP count profiler
- `sicstus_profiler.lgt`
  simple wrapper for the SICStus Prolog 4 profiler

Logtalk also supports the YAP tick profiler (using the latest YAP development version) and the SWI-Prolog XPCE profiler. When using the XPCE profiler, you can avoid profiling the Logtalk compiler (which is invoked, e.g., when you use the `(::)/2` message-sending operator at the top-level interpreter) by compiling your code with the `optimize` flag turned on:

```
?- set_logtalk_flag(optimize, on).
true.

?- use_module(library(statistics)).
true.

?- profile(... :: ...).
...
```

Given that `prolog_statistics:profile/1` is a meta-predicate, Logtalk will compile its argument before calling it thanks to the `goal_expansion/2` hook predicate definitions in the adapter file. Without this hook definition, you would need to use instead (to avoid profiling the compiler itself):

```
?- logtalk << (prolog_statistics:profile(... :: ...)).
...
```

In either case, don't forget, however, to load the `prolog_statistics` module *before* using or compiling calls to the `profile/1` to allow the Logtalk compiler to access its meta-predicate template.

The profiler support attempts to conceal internal Logtalk compiler/runtime predicates and the generated entity predicates that implement predicate inheritance. Calls to internal compiler and runtime predicates have functors starting with `$lgt_`. Calls to predicates with functors such as `_def`, `_dcl`, or `_super`, used to implement inheritance, may still be listed in a few cases. Note that the time and the number of calls/redos of concealed predicates are added to the caller predicates.

5.18.4 Compiling source code for profiling

To get the user-level object and predicate names instead of the compiler-generated internal names when using the SWI-Prolog and YAP profilers, you must set `code_prefix` flag to a character other than the default `$` before compiling your source code. For example:

```
?- set_logtalk_flag(code_prefix, '.').
```

See also the `samples/settings-sample.lgt` file for automating the necessary setup at Logtalk startup.

5.19 `tutor`

This tool adds explanations and suggestions for selected warning and error messages from the compiler/runtime and the developer tools. It's specially useful for new users not yet familiar with the warning and error messages and looking for advice on how to solve the reported issues.

5.19.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#tutor

5.19.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(tutor(loader)).
```

5.19.3 Usage

Simply load the tool at startup (e.g., from a settings file). As an example, with this tool loaded, instead of terse compiler warnings such as:

```
*     No matching clause for goal: baz(a)
*       while compiling object main_include_compiler_warning
*       in file logtalk/examples/errors/include_compiler_warning.lgt between lines 37-38
*
*     Duplicated clause: b(one)
*       first found at or above line 45
*       while compiling object main_include_compiler_warning
*       in file logtalk/examples/errors/include_compiler_warning.lgt at or above line 48
```

the user will get:

```
*     No matching clause for goal: baz(a)
*       while compiling object main_include_compiler_warning
*       in file logtalk/examples/errors/include_compiler_warning.lgt between lines 37-38
*     Calls to locally defined predicates without a clause with a matching head
*     fail. Typo in a predicate argument? Predicate definition incomplete?
*
*     Duplicated clause: b(one)
*       first found at or above line 45
*       while compiling object main_include_compiler_warning
*       in file logtalk/examples/errors/include_compiler_warning.lgt at or above line 48
*     Duplicated clauses are usually a source code editing error and can
*     result in spurious choice-points, degrading performance. Delete or
*     correct the duplicated clause to fix this warning.
```

5.20 `wrapper`

This is a prototype tool to help port a plain Prolog application to Logtalk. It can also be used to enable applying other Logtalk developer tools, such as the documenting and diagramming tools, to plain Prolog code.

The tool takes a directory of Prolog files or a list of Prolog files, loads and wraps the code in each file using an object wrapper, and advises on missing directives to be added to those objects by using the compiler lint checker and the reflection API. The user can then either save the generated wrapper objects or copy and paste the printed advice into the Prolog files (updating them to Logtalk files by adding the object opening and closing directives to the Prolog files). The wrapper objects use `include/1` directives to include the Prolog files and can be loaded for testing and for use with other tools. The wrapped Prolog files are not modified and thus require only read permission.

5.20.1 API documentation

This tool API documentation is available at:

../../apis/library_index.html#wrapper

5.20.2 Loading

This tool can be loaded using the query:

```
| ?- logtalk_load(wrapper(loader)).
```

5.20.3 Workflows

The typical porting workflow is:

```
| ?- wrapper::rdirectory(root_directory_of_prolog_code).
...
| ?- wrapper::save.
...
```

See the next section on how to customize the API calls for more flexible processing.

5.20.4 Customization

The tool can be customized by extending the `wrapper` object. A common scenario is when wrapping plain Prolog code just to take advantage, for example, of the documenting tool or for generating cross-referencing diagrams. In this case, we can workaround any compiler errors by specializing the inherited definitions for the `term_expansion/2` and `goal_expansion/2` predicates and then load the wrapper objects for further processing by using the `include_wrapped_files(false)` option described below.

The API predicates also accept a set of options for customization:

- `prolog_extensions(Extensions)`
  list of file name extensions used to recognize Prolog source files (default is `['.pl', '.pro', '.prolog']`)
- `logtalk_extension(Extension)`
  Logtalk file name extension to be used for the generated wrapper files (default is `'.lgt'`)
- `exclude_files(Files)`
  list of Prolog source files to exclude (default is `[]`)
- `exclude_directories(Files)`
  list of sub-directories to exclude (default is `[]`)
- `include_wrapped_files(Boolean)`
  generate `include/1` directives for the wrapped Prolog source files (default is `true`)

The use, by default, of `include/1` directives to wrap the code in Prolog source files facilitates running in parallel the Logtalk port and the original Prolog code. This is specially useful when the Prolog code being ported lacks a comprehensive set of tests that could be adapted to verify the Logtalk port. The generated Logtalk files can also take advantage of `uses/2` directives, including predicate aliases and predicate shorthands to minimize the changes to the original Prolog code and help verify if replacement calls to Logtalk library predicates provide the same semantics as the original calls.

5.20.5 Current limitations

- The tool cannot deal with syntax errors in the Prolog files. These errors usually occur when using a backend Prolog system different from the one used to compile the original plain Prolog code. A common cause of syntax errors is operator definitions. These can often be solved by defining those operators for the Prolog backend used to run Logtalk and this tool. An alternative is to preload the Prolog files where those operators are declared. Preloading the plain Prolog application can also help in wrapping it by ensuring that its dependencies are also loaded.
- The tool assumes that all files to be wrapped have different names (even if found in different directories). If that is not the case, the name conflicts must be manually solved before using the tool.
- There's only preliminary support for dealing with meta-predicates and advise on missing meta-predicate directives.

CHAPTER

SIX

LIBRARIES

The documentation of each library can also be found in the library directory in the `NOTES.md` file.

6.1 Overview

This folder contains libraries of useful objects, categories, and protocols. Specific notes about individual libraries can be found in the corresponding library directory `NOTES.md` files.

A plain Prolog version of the Unicode 6.2 standard is also included in the `unicode_data` folder. See its `README.md` file for details.

A `parallel_logtalk_processes_setup.pl` Prolog file is also provided with sample code for selected backend Prolog compilers for initializing Logtalk processes such that each process uses a unique scratch directory, therefore allowing parallel process execution (e.g., for usage at continuous integration servers). Starting with Logtalk 3.48.0, this setup is only required in general when running with the `clean` flag turned off. See the comments in the file itself for usage instructions.

6.1.1 Library documentation

Specific notes about each library can be found in the corresponding `NOTES.md` files. HTML documentation for each library API can be found on the `docs` directory (open the `../docs/handbook/index.html` file with your web browser).

6.1.2 Loading libraries

All the individual libraries can be loaded using the `<library name>(loader)` notation as argument for the compiling and loading predicates. For example:

```
| ?- logtalk_load(random(loader)).
```

There is also a file named `all_loader.lgt` that will load all libraries. Simply type the goal:

```
| ?- logtalk_load(library(all_loader)).
```

As a general rule, always use the corresponding loader file to load a library. Most library entities are part of small hierarchies or depend on other libraries and thus cannot be loaded and compiled separately (e.g., the `list` object implements the `listp` protocol and is part of a basic types hierarchy). Using the loader files takes care of all dependencies and also ensures compilation in optimized mode.

6.1.3 Testing libraries

Most of the libraries include unit tests in their directory, together with a `tester.lgt` file for running them. For example, to run the tests for the `random` library, we can use the goal:

```
| ?- logtalk_load(random(tester)).
```

To run all library tests, we can use the `logtalk_tester` automation script from the `library` directory at the root of the Logtalk distribution. For example, assuming the Logtalk user directory is `~/logtalk` and that we want to run the tests using ECLiPSe as the backend Prolog compiler:

```
$ cd ~/logtalk/library
$ logtalk_tester -p eclipse
```

6.1.4 Credits

Some code in this library is based on public domain Prolog code, in particular, code adopted from the Edinburgh Prolog library. The definition of the predicate `reverse/2` in the `list` object is from Richard O'Keefe and can be found in its book "The Craft of Prolog".

Some elements of this library are inspired by Richard O'Keefe library proposal available at:

http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm

Some libraries, or parts of libraries, are either ports of Prolog system libraries or inspired by Prolog system libraries. See the individual library notes for details. See also the `NOTICE.txt` file at the root of the Logtalk distribution for copyright information on third-party source code.

6.1.5 Other notes

Some files contained in this directory represent work in progress and are not loaded by default by any loader utility file.

6.2 `arbitrary`

The `arbitrary` library defines an `arbitrary` category providing predicates for generating random values for selected types to the `type` object, complementing its type checking predicates. Both the object and the category predicates can be extended by the user with definitions for new types by defining clauses for multifile predicates. This library is notably used in the QuickCheck implementation by the `lgtunit` tool. See also the documentation of the `mutations` library for related functionality.

6.2.1 API documentation

Open the ../../apis/library_index.html#arbitrary link in a web browser.

6.2.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(arbitrary(loader)).
```

6.2.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(arbitrary(tester)).
```

Several of the provided tests are generic and verify the correct behavior of all pre-defined and loaded user-defined generators and shrinkers for all ground types.

6.2.4 Pre-defined types

This library defines random generators for the most common Logtalk and Prolog types. See the API documentation for a listing of all the pre-defined types.

6.2.5 Usage

The `arbitrary` category complements the `type` object and thus its predicates are accessed via this object. For example:

```
| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = -816
yes
```

6.2.6 Defining new generators and shrinkers

To define a generator of arbitrary values for a type, define a clause for the `arbitrary::arbitrary/1` multifile predicate specifying the type and a clause for the `arbitrary::arbitrary/2` multifile predicate generating an arbitrary term of the specified type. As a simple example, assume that we want to define an "odd integer type". We start by defining both the type checker and the arbitrary generator:

```
:- multifile(type::type/1).
type::type(odd).

:- multifile(type::check/2).
type::check(odd, Term) :-
    (   var(Term) ->
        throw(instantiation_error)
    ;   integer(Term),
        Term mod 2 =:= 1 ->
        true
    ;   throw(type_error(odd, Term))
    ).
```

(continues on next page)

(continued from previous page)

```
:- multifile(arbitrary::arbitrary/1).
arbitrary::arbitrary(odd).

:- multifile(arbitrary::arbitrary/2).
arbitrary::arbitrary(odd, Arbitrary) :-
    type::arbitrary(integer, Arbitrary0),
    (   Arbitrary0 mod 2 =:= 1 ->
        Arbitrary = Arbitrary0
    ;   Arbitrary is Arbitrary0 + 1
    ).
```

The `arbitrary` library also provides *meta-types* that can simplify the definition of new generators. For example, the odd type above can also be defined using the `constrain/2` meta-type to only generate values that satisfy a closure:

```
arbitrary::arbitrary(odd, Arbitrary) :-
    arbitrary(constrain(integer, [Arbitrary]>>(Arbitrary mod 2 =:= 1)), Arbitrary).
```

Another example is using the `transform/2` meta-type to transform generated values for a base type using a closure. Assuming that we want to generate a sorted list of random integers, we can write:

```
arbitrary::arbitrary(sorted_integer_list, Arbitrary) :-
    arbitrary(transform(list(integer), sort), Arbitrary).
```

We can also define a clause for the `arbitrary::shrinker/1` multifile predicate to declare a new shrinker and a `arbitrary::shrink/3` multifile predicate for shrinking arbitrary values for QuickCheck usage:

```
:- multifile(arbitrary::shrinker/1).
arbitrary::shrinker(odd).

:- multifile(arbitrary::shrink/3).
arbitrary::shrink(odd, Large, Small) :-
    integer(Large),
    (   Large < -1 ->
        Small is Large + 2
    ;   Large > 1,
        Small is Large - 2
    ).
```

Definitions for the `shrink/3` predicate should either succeed or fail but never throw an exception. The `shrink_sequence/3` predicate can be used to help test that shrinking a value results in a finite sequence of values.

It is also possible to define edge cases for a given type for use with QuickCheck implementations. For example:

```
:- multifile(arbitrary::edge_case/2).
arbitrary::edge_case(odd,  1).
arbitrary::edge_case(odd, -1).
```

Edge cases are tried before resorting to generating arbitrary values for a type.

A more complex example is generating arbitrary values for a recursive type. A simple example of a recursive type is a binary tree. Assuming that we are working with a binary tree holding integers where each node is

represented by a `node(Left, Right)` compound term, we can define a `node(Depth)` type where `Depth` is the maximum depth of the tree. This argument allows us to prevent excessively deep trees:

```
:- category(binary_tree).

    :- multifile(type::type/1).
    type::type(node(_)).

    :- multifile(type::check/2).
    type::check(node(_), Term) :-
        (   check(Term) ->
            true
        ;   var(Term) ->
            throw(instantiation_error)
        ;   throw(type_error(node(_), Term))
        ).

    check(Term) :-
        (   integer(Term) ->
            true
        ;   compound(Term),
            Term = node(Left, Right),
            check(Left),
            check(Right)
        ).

    :- multifile(arbitrary::arbitrary/1).
    arbitrary::arbitrary(node(_)).

    :- multifile(arbitrary::arbitrary/2).
    arbitrary::arbitrary(node(Depth), Arbitrary) :-
    (   Depth > 1 ->
        NewDepth is Depth - 1,
        type::arbitrary(
            types_frequency([
                integer - 1,
                compound(
                    node,
                    [
                        types([node(NewDepth), integer]),
                        types([node(NewDepth), integer])
                    ]
                ) - 3
            ]),
            Arbitrary
        )
    ;   type::arbitrary(
    integer, Arbitrary)
    ).

:- end_category.
```

In this second example, we use some of the pre-defined types provided by the library. The `types_frequency(Pairs)` type supports generating random terms for a type in the `Type-Frequency` pairs

list where the type is randomly chosen after the types relative frequency. The `compound(Name, Types)` type supports generating compound terms with a given name and random arguments after the given types:

```
| ?- type::arbitrary(node(4), Arbitrary).
Arbitrary = 907
yes

| ?- type::arbitrary(node(4), Arbitrary).
Arbitrary = node(node(node(522, 509), node(83, 453)), node(454, -197))
yes

| ?- type::arbitrary(node(4), Arbitrary).
Arbitrary = node(node(-875, -866), -254)
yes

| ?- type::arbitrary(node(4), Arbitrary).
Arbitrary = node(-133, -831)
yes
```

The source code of these examples can be found in the `test_files` directory. Other examples of arbitrary term generators can be found in the implementation of the `optionals` and `expecteds` libraries.

6.2.7 Scoped generators and shrinkers

Declaring a new generator and possibly a shrinker for a custom type raises the possibility of a conflict with third-party-defined generators and shrinkers. An alternative is to use the `(::)/2` meta-type to define scoped generators and shrinkers. For example:

```
:- object(scoped).

    % the same predicate is used for both generating and validating
    :- public(custom/1).
    custom(Term) :-
        (   var(Term) ->
            % assume predicate used as a generator
            random::random(Term)
        ;   % assume predicate used as a validator
            float(Term)
        ).

    % a predicate with the same name is used for shrinking
    :- public(custom/2).
    custom(Larger, Small) :-
        Small is Larger / 2.

:- end_object.
```

Some sample calls:

```
| ?- type::arbitrary(scoped::custom, Arbitrary).
Arbitrary = 0.5788130906607927
yes
```

(continues on next page)

(continued from previous page)

```
| ?- type::valid(scoped::custom, foo).
no

| ?- type::check(scoped::custom, _).
ERROR: type_error(instantiation_error)

| ?- type::check(scoped::custom, foo).
ERROR: type_error(scoped::custom, foo)

| ?- type::shrink(scoped::custom, 0.42, Smaller).
Smaller = 0.21
yes
```

The source code of this example can be found in the `test_files` directory.

6.2.8 Reproducing sequences of arbitrary terms

The `arbitrary` category provides access to the pseudo-random generator it uses via the `get_seed/1` and `set_seed/1`. This allows sequences of arbitrary values to be reproduced. For example:

```
| ?- type::get_seed(Seed).
Seed = seed(3172, 9814, 20125)
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = -816
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = -113
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = 446

| ?- type::set_seed(seed(3172, 9814, 20125)).
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = -816
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = -113
yes

| ?- type::arbitrary(integer, Arbitrary).
Arbitrary = 446
yes
```

The seed should be regarded as an opaque term and handled using the `get_seed/1` and `set_seed/1` predicates. These predicates are notably used in the QuickCheck implementation provided by the `lgtunit` tool.

6.2.9 Default size of generated terms

The library uses the value 42 for the default size of generated terms for types where size is meaningful and implicit. To override this default value, define a clause for the `arbitrary::max_size/1` multifile predicate. The new default size must be a positive integer. For example:

```
:- multifile(arbitrary::max_size/1).
arbitrary::max_size(7).
```

When multiple definitions exist, the first valid one found is used. When no definition is valid, the default value of 42 is used.

6.2.10 Known issues

Some Prolog systems either don't support the null character or provide buggy results when calling `char_code/2` with a code of zero. When that's the case, the null character is excluded when generating arbitrary characters or character codes.

Generating arbitrary Unicode characters (instead of Unicode codepoints) is inherently problematic as the process first generates codepoints and then tries to use the standard `char_code/2` to convert them to characters. But, depending on the backend Prolog system and its internal (if any) Unicode normalization, it may not be possible to convert a codepoint to a single character.

6.3 assignvars

The `assignvarsp` protocol declares the predicates used for logical assignment of Prolog terms developed by Nobukuni Kino.

The `assignvars` object provides a declarative implementation of the `assignvarsp` protocol. It can be used with any backend Prolog compiler.

The `nd_assignvars` object provides a non-declarative but faster implementation of the `assignvarsp` protocol. It can be used with the following backend Prolog compilers: B-Prolog, CxProlog, ECLiPSe, GNU Prolog, SICStus Prolog, SWI-Prolog, and YAP.

For more information on assignvars, please consult the URL:

https://web.archive.org/web/20160818050049/http://www.kprolog.com/en/logical_assignment/

The representation of assignable terms should be regarded as an opaque term and only accessed using the library predicates.

6.3.1 API documentation

Open the ../../apis/library_index.html#assignvars link in a web browser.

6.3.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(assignvars(loader)).
```

6.3.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(assignvars(tester)).
```

6.4 base64

The `base64` library provides predicates for parsing and generating data in the Base64 and Base64URL formats as per the specification found at:

https://www.rfc-editor.org/rfc/rfc4648

6.4.1 API documentation

Open the ../../apis/library_index.html#base64 link in a web browser.

6.4.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(base64(loader)).
```

6.4.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(base64(tester)).
```

6.4.4 Encoding

Encoding a list of bytes in Base64 format is accomplished by the `base64::generate/2` predicate. For example:

```
| ?- atom_codes('Hello world!', Bytes),
     base64::generate(atom(Base64), Bytes).
Base64 = 'SGVsbG8gd29ybGQh'
Bytes = [72,101,108,108,111,32,119,111,114,108,100,33]
yes

| ?- atom_codes('Hello world!', Bytes),
```

(continues on next page)

(continued from previous page)

```
    base64::generate(codes(Base64), Bytes).
Base64 = [83,71,86,115,98,71,56,103,100,50,57,121,98,71,81,104]
Bytes = [72,101,108,108,111,32,119,111,114,108,100,33]
yes
```

The Base64 result can also be represented using a list of chars, written to a file or to a stream. See the API documentation for details.

For safe encoding of URLs, use instead the Base64URL format. For example:

```
| ?- base64url::generate(atom(Base64URL), 'https://logtalk.org').
Base64URL == 'aHR0cHM6Ly9sb2d0YWxrLm9yZw'
yes
```

The Base64URL can also be represented using a list of chars or a list of codes. The input URL should be in the same format.

6.4.5 Decoding

Decoding of Base64 data is accomplished using the `base64::parse/2` predicate. For example:

```
| ?- base64::parse(atom('SGVsbG8gd29ybGQh'), Bytes),
     atom_codes(Atom, Bytes).
Atom = 'Hello world!'
Bytes = [72,101,108,108,111,32,119,111,114,108,100,33]
yes

| ?- base64::parse(chars(['S','G','V',s,b,'G','8',g,d,'2','9',y,b,'G','Q',h]), Bytes),
     atom_codes(Atom, Bytes).
Atom = 'Hello world!'
Bytes = [72,101,108,108,111,32,119,111,114,108,100,33]
yes
```

The `base64::parse/2` predicate accepts other input source such as a file or a stream. See the API documentation for details.

For decoding of URLs in the Base64URL format, use the `base64url::parse/2` predicate. For example:

```
| ?- base64url::parse(atom('aHR0cHM6Ly9sb2d0YWxrLm9yZw'), URL).
URL == 'https://logtalk.org'
yes
```

The `base64url::parse/2` predicate also accepts a list of chars or a list of codes as input. See the API documentation for details.

6.5 basic_types

The `basic_types` library is a virtual library that loads only basic types from the `types` library:

- `comparingp`
- `termp`, `term`
- `atomic`, `atom`, `number`, `float`, `integer`
- `compound`, `listp`, `list`
- `type`

6.5.1 API documentation

Open the ../../apis/library_index.html#types link in a web browser.

6.5.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(basic_types(loader)).
```

6.5.3 Testing

To test this library predicates, load the `tester.lgt` file for the `types` library:

```
| ?- logtalk_load(types(tester)).
```

6.6 cbor

The `cbor` library implements predicates for importing and exporting data in the Concise Binary Object Representation (CBOR) format:

- https://www.rfc-editor.org/rfc/rfc8949.html
- http://cbor.io/

This library is a work-in-progress. Currently it requires a backend supporting unbounded integer arithmetic.

6.6.1 Representation

- Maps are represented using curly-bracketed terms, `{Pairs}`, where each pair uses the representation `Key-Value`.
- Arrays are represented using lists.
- Byte strings uses `bytes(List)` compound terms.
- Text strings can be represented as atoms, `chars(List)`, or `codes(List)`. The default when decoding is to use atoms when using the `cbor` object. To decode text strings into lists of chars or code, use the `cbor/1` with the parameter bound to `chars` or `codes`. For example:

```
| ?- cbor::parse([0x65,0x68,0x65,0x6c,0x6c,0x6f], Term).
Term = hello
yes

| ?- cbor(atom)::parse([0x65,0x68,0x65,0x6c,0x6c,0x6f], Term).
Term = hello
yes

| ?- cbor(chars)::parse([0x65,0x68,0x65,0x6c,0x6c,0x6f], Term).
Term = chars([h,e,l,l,o])
yes

| ?- cbor(codes)::parse([0x65,0x68,0x65,0x6c,0x6c,0x6f], Term).
Term = codes([104,101,108,108,111])
yes
```

- Tagged data uses `tag(Tag, Data)` compound terms.
- Simple values can be represented using `simple(Simple)` compound terms.
- The CBOR elements `false`, `true`, `null`, and `undefined` are represented by, respectively, the `@false`, `@true`, `@null`, and `@undefined` compound terms.
- The compound terms `@infinity`, `@negative_infinity`, and `@not_a_number` are used to represent the corresponding CBOR elements.
- Only some backends distinguish between positive zero and negative zero. The compound terms `@zero` and `@negative_zero` can be used as an alternative for encoding. The decoder, however, produces the `0.0` and `-0.0` floats.

6.6.2 Encoding

Encoding is accomplished using the `generate/2` predicate. For example:

```
| ?- cbor::generate([a,{b-c}], Encoding).
Encoding = [0x9f,0x61,0x61,0xbf,0x61,0x62,0x61,0x63,0xff,0xff]
yes
```

The encoding of arrays and maps uses indefinite-length encoding. All floats are currently encoded using decimal fractions. Encoding indicators and big floats are not currently supported.

6.6.3 Decoding

Decoding is accomplished using the `parse/2` predicate. For example:

```
| ?- cbor::parse([0x9f,0x61,0x61,0xbf,0x61,0x62,0x61,0x63,0xff,0xff], Term).
Term = [a,{b-c}]
yes
```

6.6.4 API documentation

Open the ../../apis/library_index.html#cbor link in a web browser.

6.6.5 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(cbor(loader)).
```

6.6.6 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(cbor(tester)).
```

6.7 core

This library consists of built-in entities.

6.7.1 API documentation

See this Handbook "Objects", "Protocols", and "Categories" sections.

6.7.2 Loading

All entities in this library are automatically loaded at Logtalk startup.

6.7.3 Testing

Tests for this library can be found in the `tests/logtalk/entities` directory.

6.8 coroutining

The `coroutining` object provides a portable abstraction over how common coroutining predicates are made available by the supported backend Prolog systems (ECLiPSe, XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, and YAP) that provide them. Partial support for XSB is provided (the predicate `frozen/2` is not available and calls to it fail).

Calls to the library predicates are inlined when compiled with the `optimize` flag turned on. In this case, there is no overhead compared with calling the abstracted predicates directly.

See also the `dif` library.

6.8.1 API documentation

Open the ../../apis/library_index.html#coroutining link in a web browser.

6.8.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(coroutining(loader)).
```

6.8.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(coroutining(tester)).
```

6.8.4 Usage

Load this library from your application loader file. To call the coroutining predicates using implicit message-sending, add the following directive to any object or category calling the predicates (adjust the list to the predicates actually called):

```
:- uses(coroutining, [
    dif/2, dif/1, freeze/2, frozen/2, when/2
]).
```

6.9 csv

The `csv` library provides predicates for reading and writing CSV files and streams:

https://www.rfc-editor.org/rfc/rfc4180.txt

The main object, `csv/3`, is a parametric object allowing passing options for the handling of the header of the file, the fields separator, and the handling of double-quoted fields. The `cvs` object extends the `csv/3` parametric object using default option values.

The library also includes predicates to guess the separator and guess the number of columns in a given CSV file.

Files and streams can be read into a list of rows (with each row being represented by a list of fields) or asserted using a user-defined dynamic predicate. Reading can be done by first loading the whole file (using the `read_file/2-3` predicates) into memory or line by line (using the `read_file_by_line/2-3` predicates). Reading line by line is usually the best option for parsing large CSV files.

Data can be saved to a CSV file or stream by providing the object and predicate for accessing the data plus the name of the destination file or the stream handle or alias.

6.9.1 API documentation

Open the ../../apis/library_index.html#csv link in a web browser.

6.9.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(csv(loader)).
```

6.9.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(csv(tester)).
```

6.9.4 Usage

The `csv(Header, Separator, IgnoreQuotes)` parametric object allows passing the following options:

1. `Header`: possible values are `missing`, `skip`, and `keep`.
2. `Separator`: possible values are `comma`, `tab`, `semicolon`, and `colon`.
3. `IgnoreQuotes`: possible values are `true` to ignore double quotes surrounding field data and `false` to preserve the double quotes.

The `csv` object uses the default values `keep`, `comma`, and `false`.

When writing CSV files or streams, set the quoted fields option to `false` to write all non-numeric fields double-quoted (i.e., escaped).

The library objects can also be used to guess the separator used in a CSV file if necessary. For example:

```
| ?- csv::guess_separator('test_files/crlf_ending.csv', Separator).
Is this the proper reading of a line of this file (y/n)? [aaa,bb,ccc]
|> y.

Separator = comma ?
```

This information can then be used to read the CSV file returning a list of rows:

```
| ?- csv(keep, comma, true)::read_file('test_files/crlf_ending.csv', Rows).

Rows = [[aaa,bbb,ccc],[zzz,yyy,xxx]] ?
```

Alternatively, the CSV data can be saved using a public and dynamic object predicate (that must be previously declared). For example:

```
| ?- assertz(p(_,_,_)), retractall(p(_,_,_)).
yes

| ?- csv(keep, comma, true)::read_file('test_files/crlf_ending.csv', user, p/3).
```

(continues on next page)

(continued from previous page)

```
yes

| ?-  p(A,B,C).

A = aaa
B = bbb
C = ccc ? ;

A = zzz
B = yyy
C = xxx
```

Given a predicate representing a table, the predicate data can be written to a file or stream. For example:

```
| ?- csv(keep, comma, true)::write_file('output.csv', user, p/3).
yes
```

leaving the content just as the original file thanks to the use of `true` for the `IgnoreQuotes` option:

```
aaa,bbb,ccc
zzz,yyy,xxx
```

Otherwise:

```
| ?- csv(keep, comma, false)::write_file('output.csv', user, p/3).
yes
```

results in the following file content:

```
"aaa","bbb","ccc"
"zzz","yyy","xxx"
```

The `guess_arity/2` method, to identify the arity, i. e. the number of fields or columns per record in a given CSV file, for example:

```
| ?- csv(keep, comma, false)::guess_arity('test_files/crlf_ending.csv', Arity).
Is this the proper reading of a line of this file (y/n)? [aaa,bbb,ccc]
|> y.

Arity = 3
```

6.10 dates

The `date` object implements some useful calendar date predicates.

The `time` object implements some useful time predicates.

Please note that the functionality of these objects depends on the chosen Prolog support for accessing the operating system time and date.

6.10.1 API documentation

Open the ../../apis/library_index.html#dates link in a web browser.

6.10.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(dates(loader)).
```

6.11 dependents

The `observer` and `subject` categories implement the Smalltalk dependents handling mechanism. This mechanism can be used as an alternative to Logtalk system-wide support for event-driven programming.

6.11.1 API documentation

Open the ../../apis/library_index.html#dependents link in a web browser.

6.11.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(dependents(loader)).
```

6.12 dictionaries

This library provides a dictionary (also known as associative array, map, or symbol table) protocol and binary tree, AVL tree, and Red–Black tree implementations. The different representations of a dictionary should be regarded as opaque terms and only accessed using the library predicates.

6.12.1 API documentation

Open the ../../apis/library_index.html#dictionaries link in a web browser.

6.12.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(dictionaries(loader)).
```

6.12.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(dictionaries(tester)).
```

6.12.4 Usage

First, select the dictionary implementation that you want to use. For cases where the number of elements is relatively small and performance is not critical, `bintree` can be a good choice. For other cases, `avltree` or `rbtree` are likely better choices. If you want to compare the performance of the implementations, either define an object alias or use a `uses/2` directive so that you can switch between implementations by simply changing the alias definition or the first argument of the directive. Note that you can switch between implementations at runtime without code changes by using a parameter variable in the first argument of a `uses/2` directive.

Dictionary keys should preferably be ground terms. If the keys contain variables, the user must ensure that any instantiation of those variables when calling this library predicates will not affect the key ordering.

To create a new dictionary, you can use the `new/1` predicate. For example:

```
| ?- avltree::new(Dictionary).
Dictionary = ...
yes
```

You can also create a new dictionary from a list of key-value pairs by using the `as_dictionary/2` predicate. For example:

```
| ?- avltree::as_dictionary([a-1,c-3,b-2], Dictionary).
Dictionary = ...
yes
```

Several predicates are provided for inserting key-value pairs, lookup key-value pairs updating the value associated with a key, and deleting key-value pairs. For example:

```
| ?- avltree::(
        new(Dictionary0),
        insert(Dictionary0, a, 1, Dictionary1),
        update(Dictionary1, a, 2, Dictionary2),
        lookup(a, Value, Dictionary2)
    ).
Dictionary0 = ...,
Dictionary1 = ...,
Dictionary2 = ...,
Value = 2
yes
```

For details on these and other provided predicates, consult the library API documentation.

6.12.5 Credits

The AVL tree implementation is an adaptation to Logtalk of the `assoc` SWI-Prolog library authored by R.A.O'Keefe, L.Damas, V.S.Costa, Glenn Burgess, Jiri Spitz, and Jan Wielemaker. Additional predicates authored by Paulo Moura.

The Red–Black tree implementation is an adaptation to Logtalk of the `rbtrees` Prolog library authored by Vitor Santos Costa.

6.13 `dif`

The `dif` object provides a portable abstraction over how the `dif/2` predicate is made available by the supported backend Prolog systems that implement it (B-Prolog, ECLiPSe, XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, XSB, and YAP).

Calls to the library predicates are inlined when compiled with the `optimize` flag turned on. In this case, there is no overhead compared with calling the abstracted predicate directly.

See also the `coroutining` library.

6.13.1 API documentation

Open the ../../apis/library_index.html#dif link in a web browser.

6.13.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(dif(loader)).
```

6.13.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(dif(tester)).
```

6.13.4 Usage

Load this library from your application loader file. To call the `dif/1-2` predicates using implicit message-sending, add the following directive to any object or category calling the predicates:

```
:- uses(dif, [
    dif/2, dif/1
]).
```

6.14 edcg

This library provides a Logtalk port of Peter Van Roy's extended DCG implementation. For full documentation on EDCGs, see:

https://webperso.info.ucl.ac.be/~pvr/edcg.html

This Logtalk version defines a hook object, `edcg`. Source files defining EDCGs must be compiled using the compiler option `hook(edcg)`:

```
| ?- logtalk_load(source, [hook(edcg)]).
```

Alternatively, the following directive can be added at the beginning of the source file containing the EDCGs:

```
:- set_logtalk_flag(hook, edcg).
```

The hook object automatically adds the EDCGs `-->>` infix operator scoped to the source file.

This port has simplified by copying and then modifying Michael Hendricks's `edcg` repo at:

https://github.com/mndrix/edcg

A notable difference is that Michael's version declares Peter's original predicates for declaring accumulators and predicates using the hidden arguments as multifile predicates. But this is risky as two independent EDCGs may use e.g. the same accumulator names and introduce conflicts. The Logtalk version uses instead the `edcg` hook object internal state to temporarily save those predicates in order to parse the corresponding EDCGs.

6.14.1 API documentation

Open the ../../apis/library_index.html#edcg link in a web browser.

6.14.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(edcg(loader)).
```

6.14.3 Testing

To test this library predicates, load the `tester.lgt` file of the `edcgs` example:

```
| ?- logtalk_load(edcgs(tester)).
```

6.14.4 Usage

Follows the usage documentation written by Michael Hendricks (with a contribution from Peter Ludemann), used here with permission, with the necessary changes for the Logtalk port.

```
% declare accumulators
acc_info(adder, X, In, Out, integer::plus(X,In,Out)).

% declare predicates using these hidden arguments
pred_info(len,0,[adder,dcg]).
pred_info(increment,0,[adder]).

increment -->>
    % add one to the accumulator
    [1]:adder.


len(Xs,N) :-
    len(0,N,Xs,[]).

len -->>
    % 'dcg' accumulator has an element
    [_],
    !,
    % increment the 'adder' accumulator
    increment,
    len.
len -->>
    [].
```

6.14.5 Introduction

DCG notation gives us a single, hidden accumulator. Extended DCG notation (implemented by this library) lets predicates have arbitrarily many hidden accumulators. As demonstrated by the synopsis above, those accumulators can be implemented with arbitrary goals (like `integer::plus/3`).

Benefits of this library:

- avoid tedium and errors from manually threading accumulators through your predicates
- add or remove accumulators with a single declaration
- change the accumulator implementation with a single declaration (ex, switching from ordsets to rb-trees)

6.14.6 Syntax

Extended DCG syntax is very similar to DCG notation. An EDCG is created with clauses whose neck is the `-->>` operator. The following syntax is supported inside an EDCG clause:

- `{Goal}` - don't expand any hidden arguments of `Goal`
- `Goal` - expand all hidden arguments of Goal that are also in the head. Those hidden arguments not in the head are given default values.
- `Goal:L` - If `Goal` has no hidden arguments then force the expansion of all arguments in `L` in the order given. If `Goal` has hidden arguments then expand all of them, using the contents of `L` to override the expansion. `L` is either a term of the form `Acc`, `Acc(Left,Right)`, `Pass`, `Pass(Value)`, or a list of such terms. When present, the arguments `Left`, `Right`, and `Value` override the default values of arguments not in the head.
- `List:Acc` - Accumulate a list of terms in the accumulator `Acc`
- `List` - Accumulate a list of terms in the accumulator `dcg`
- `X/Acc` - Unify `X` with the left term for the accumulator `Acc`
- `Acc/X` - Unify `X` with the right term for the accumulator `Acc`
- `X/Acc/Y` - Unify `X` with the left and `Y` with the right term for the accumulator `Acc`
- `insert(X,Y):Acc` - Insert the arguments `X` and `Y` into the chain implementing the accumulator `Acc`. This is useful when the value of the accumulator changes radically because `X` and `Y` may be the arguments of an arbitrary relation
- `insert(X,Y)` - Insert the arguments `X` and `Y` into the chain implementing the accumulator `dcg`. This inserts the difference list X-Y into the accumulated list

6.14.7 Declaration of Predicates

Predicates are declared with facts of the following form:

```
pred_info(Name, Arity, List).
```

The predicate `Name/Arity` has the hidden parameters given in `List`. The parameters are added in the order given by `List` and their names must be atoms.

6.14.8 Declaration of Accumulators

Accumulators are declared with facts in one of two forms. The short form is:

```
acc_info(Acc, Term, Left, Right, Joiner).
```

The long form is:

```
acc_info(Acc, Term, Left, Right, Joiner, LStart, RStart).
```

In most cases the short form gives sufficient information. It declares the accumulator `Acc`, which must be an atom, along with the accumulating function, `Joiner`, and its arguments `Term`, the term to be accumulated, and `Left` & `Right`, the variables used in chaining.

The long form of `acc_info` is useful in more complex programs. It contains two additional arguments, `LStart` and `RStart`, that are used to give default starting values for an accumulator occurring in a body goal that

does not occur in the head. The starting values are given to the unused accumulator to ensure that it will execute correctly even though its value is not used. Care is needed to give correct values for `LStart` and `RStart`. For DCG-like list accumulation both may remain unbound.

Two conventions are used for the two variables used in chaining depending on which direction the accumulation is done. For forward accumulation, `Left` is the input and `Right` is the output. For reverse accumulation, `Right` is the input and `Left` is the output.

6.14.9 Declaration of Passed Arguments

Passed arguments are conceptually the same as accumulators with `=/2` as the joiner function. Passed arguments are declared as facts in one of two forms. The short form is:

```
pass_info(Pass).
```

The long form is:

```
pass_info(Pass, PStart).
```

In most cases the short form is sufficient. It declares a passed argument `Pass`, that must be an atom. The long form also contains the starting value `PStart` that is used to give a default value for a passed argument in a body goal that does not occur in the head. Most of the time this situation does not occur.

6.14.10 Additional documentation

Peter Van Roy's page: Declarative Programming with State

Technical Report UCB/CSD-90-583 Extended DCG Notation: A Tool for Applicative Programming in Prolog by Peter Van Roy

- The Tech Report's PDF is here

A short Wikipedia article on DCGs and extensions.

6.15 events

The objects `event_registry`, `before_event_registry`, and `after_event_registry` implement convenient predicates for registering before and after events.

The code makes use of the `monitoring` built-in protocol, which declares the two basic event handler predicates (`before/3` and `after/3`). You will need to refer to this protocol in your objects if you want to use the super control structure (`^^/1`) with these predicates.

The `monitor` object implements more sophisticated event handling predicates.

6.15.1 API documentation

Open the ../../apis/library_index.html#events link in a web browser.

6.15.2 Loading

To load all entities in this library, load the `loader.lgt` loader file:

```
| ?- logtalk_load(events(loader)).
```

6.16 expand_library_alias_paths

This library provides provides a hook object, `expand_library_alias_paths`, for expanding library alias paths in `logtalk_library_path/2` facts in source files. It is mainly used when embedding Logtalk and Logtalk applications.

6.16.1 API documentation

Open the ../../apis/library_index.html#expand-library-alias-paths link in a web browser.

6.16.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(expand_library_alias_paths(loader)).
```

6.16.3 Usage

Use the `hook/1` option when compiling a source file:

```
| ?- logtalk_load(my_source_file, [hook(expand_library_alias_paths)]).
...
```

Alternatively, assuming it is the only hook object you are using, you can set it as thew default hook object:

```
| ?- set_logtalk_flag(hook, expand_library_alias_paths).
...
```

6.17 expecteds

This library provides an implementation of *expected terms* with an API that is inspired by the `optional` library and C++ standardization proposals for an `Expected<T>` type. An expected term is an opaque compound term that either contains an expected value or an error informing why the expected value is not present. Expected terms provide an alternative to generating an exception (or a failure) when something unexpected happens when asking for a value. This allows, e.g., separating the code that constructs expected terms from the code that processes them, which is then free to deal if necessary and at its convenience with any unexpected events.

6.17.1 API documentation

Open the ../../apis/library_index.html#expecteds link in a web browser.

6.17.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(expecteds(loader)).
```

6.17.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(expecteds(tester)).
```

6.17.4 Usage

The `expected` object provides constructors for expected terms. For example:

```
| ?- expected::of_expected(1, Expected).
...
```

The created expected terms can then be passed as parameters to the `expected/1` parametric object. For example:

```
| ?- expected::of_expected(1, Expected), expected(Expected)::or_else(Value, 0).
Expected = expected(1),
Value = 1
yes

| ?- expected::of_unexpected(-1, Expected), expected(Expected)::or_else(Value, 0).
Expected = unexpected(-1),
Value = 0
yes
```

The `either` object provides types and predicates for extended type-checking and predicates for handling lists of expected terms.

6.17.5 See also

The `optionals` library.

6.18 format

The format object provides a portable abstraction over how the de facto standard format/2-3 predicates are made available by the supported backend Prolog systems. Some systems provide these predicates as built-in predicates, while others make them available using a library that must be explicitly loaded.

Calls to the library predicates are inlined when compiled with the optimize flag turned on for most of the backends. When that's the case, there is no overhead compared with calling the abstracted predicates directly.

This library provides linter checks for calls to the format/2-3 predicates. Given the differences between the implementations of these predicates among Prolog systems, the linter checks focus on detecting common errors such as missing arguments and too many arguments. The linter warnings are printed when the suspicious_calls flag is set to warning (its usual default).

6.18.1 API documentation

Open the ../../apis/library_index.html#format link in a web browser.

6.18.2 Loading

To load all entities in this library, load the loader.lgt file:

```
| ?- logtalk_load(format(loader)).
```

6.18.3 Testing

Minimal tests for this library predicates can be run by loading the tester.lgt file:

```
| ?- logtalk_load(format(tester)).
```

Detailed tests for the format/2-3 predicates are available in the tests/prolog/predicates directory as part of the Prolog standards conformance test suite. Use those tests to confirm the portability of the format specifiers that you want to use.

6.18.4 Usage

Load this library from your application loader file. To call the format/2-3 predicates using implicit message-sending, add the following directive to any object or category calling the predicates:

```
:- uses(format, [
    format/2, format/3
]).
```

6.18.5 Portability

Some Prolog systems provide only a subset of the expected format specifiers. Notably, table-related format specifiers are only fully supported by a few systems. See the section below on testing.

Only some of the supported Prolog backends provide implementations of the `format/2-3` predicates that allow using not only an atom or a list of character codes for the format string (as de facto standard) but also using a list of characters. These currently include ECLiPSe, GNU Prolog, XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, and YAP. Therefore, when wide portability is sought, atoms must be used for the format specifier argument as they bypass any dependency on the `double_quotes` standard Prolog flag. Some systems, like Tau Prolog, only accept a list of characters for the format string. In this case, this library will convert the atom format string before calling these systems native implementations.

6.19 genint

The `genint` library implements predicates for generating positive integers in increasing order. The public predicates are declared synchronized to prevent race conditions when using a backend Prolog compiler with multi-threading support.

6.19.1 API documentation

Open the ../../apis/library_index.html#genint link in a web browser.

6.19.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(genint(loader)).
```

6.19.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(genint(tester)).
```

6.19.4 Usage

The `genint_core` category implements the library predicates. This category is imported by the default `genint` object to provide application global named counters. To make the counters local and thus minimize the potential for counter name clashes, the category can be imported by one or more application objects. Use protected or private import to restrict the scope of the library predicates.

6.20 gensym

The gensym library implements predicates for generating unique atoms. The public predicates are declared synchronized to prevent race conditions when using a backend Prolog compiler with multi-threading support.

6.20.1 API documentation

Open the ../../apis/library_index.html#gensym link in a web browser.

6.20.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(gensym(loader)).
```

6.20.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(gensym(tester)).
```

6.20.4 Usage

The gensym_core category implements the library predicates. This category is imported by the default gensym object to provide application global generators. To make the generators local and thus minimize the potential for generator name clashes, the category can be imported by one or more application objects. Use protected or private import to restrict the scope of the library predicates. For example:

```
:- object(foo,
    imports(private::gensym_core)).

    bar :-
        ^^gensym(p, S),
        ...

:- end_object.
```

6.21 git

This library provides access to a git project current branch and latest commit data (e.g., commit hash). Support for using this library on Windows operating-systems is experimental and may or may not work depending on the used Prolog backend.

6.21.1 API documentation

Open the ../../apis/library_index.html#git link in a web browser.

6.21.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(git(loader)).
```

6.21.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(git(tester)).
```

6.21.4 Usage

All predicates take as first argument a directory, which should be either a git repository directory or a sub-directory of a git repository directory. The main predicate is `commit_log/3`. It provides access to the `git log` command when called with the `--oneline -n 1 --pretty=format:` options. By passing as second argument the desired format, it returns an atom with the formatted output. For example:

```
| ?- git::commit_log('/Users/pmoura/logtalk3', '%h%n%B', Output),
     write(Output), nl.

eccaa1a2a
Update SVG diagrams

Output = 'eccaa1a2a\nUpdate SVG diagrams\n'
yes
```

See e.g. the official documentation on `git log` pretty formats for details:

https://git-scm.com/docs/pretty-formats

Convenient predicates are also provided for commonly used formats such as the commit author and the commit hash. For example:

```
| ?- git::commit_author('/Users/pmoura/Documents/Logtalk/logtalk3', Author).

Author = 'Paulo Moura'
yes

| ?- git::commit_hash('/Users/pmoura/Documents/Logtalk/logtalk3', Hash).

Hash = eccaa1a2a9495fef441915bbace84e0a4b0394a2
yes
```

It's also possible to get the name of the current local branch. For example:

```
| ?- git::branch('/Users/pmoura/Documents/Logtalk/logtalk3', Branch).

Branch = master
yes
```

6.22 grammars

This library provides Definite Clause Grammars (DCGs) for common parsing tasks. The DCGs support parsing both lists of characters (aka chars) and lists of character codes (aka codes).

Currently, four groups of DCGs are available, each defined in its own file:

- blanks (`blank_grammars.lgt`)
- numbers (`number_grammars.lgt`)
- sequences (`sequence_grammars.lgt`)
- IP addresses (`ip_grammars.lgt`; depends on `number_grammars.lgt`)

6.22.1 API documentation

Open the ../../apis/library_index.html#grammars link in a web browser.

6.22.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(grammars(loader)).
```

6.22.3 Testing

Minimal tests for this library predicates can be run by loading the `tester.lgt` file:

```
| ?- logtalk_load(grammars(tester)).
```

6.22.4 Usage

The library uses (when necessary) parametric objects where the single parameter can be either `chars` or `codes`. The parameter must be bound when using the DCGs. For example, when using implicit message-sending:

```
:- uses(blank_grammars(chars), [
    white_spaces//0, new_lines//0
]).
```

6.23 heaps

This library implements minimum and maximum heaps.

6.23.1 API documentation

Open the ../../apis/library_index.html#heaps link in a web browser.

6.23.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(heaps(loader)).
```

6.23.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(heaps(tester)).
```

6.23.4 Credits

Original code by Richard O"Keefe. Adapted to Logtalk by Paulo Moura and Victor Lagerkvist.

6.24 hierarchies

This library provides categories implementing reflection predicates over class and prototype hierarchies. These categories can be imported by any object that requires reasoning about hierarchies.

6.24.1 API documentation

Open the ../../apis/library_index.html#hierarchies link in a web browser.

6.24.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(hierarchies(loader)).
```

6.24.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(hierarchies(tester)).
```

6.25 hook_flows

Hook objects (i.e., objects that define term- and goal-expansion rules) can be combined to define expansion *workflows*. While in some cases the expansions are independent and thus can be applied in any order, in other cases a specific order is required. The `hook_pipeline` and `hook_set` parametric objects in this library implement the two most common scenarios of combining multiple hook objects for the expansion of source files. These parametric hook objects can be combined to define workflows of any complexity (e.g., a pipeline where one of the steps is set with an element that is a pipeline). These two basic hook flows can also be used as examples of how to construct your own custom expansion workflows.

6.25.1 API documentation

Open the ../../apis/library_index.html#hook-flows link in a web browser.

6.25.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(hook_flows(loader)).
```

6.25.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(hook_flows(tester)).
```

6.25.4 Usage

Pre-processing a source file with a hook object can be accomplished by either compiling the source file using the option `hook(HookObject)` or by adding to the top of the file the directive:

```
:- set_logtalk_flag(hook, HookObject).
```

Note that `set_logtalk_flag/2` directives are local to an entity or a source file.

The `hook_pipeline(Pipeline)` is a parametric object where the parameter is a list of hook objects, interpreted as a pre-processing pipeline: the results of a hook object are passed to the next hook object. Terms and goals that are not expanded by a hook object are passed as-is to the next hook object. This parametric object is used when a set of expansions must be applied in a specific order. It also allows overriding the default compiler semantics where term-expansion rules are tried in sequence only until one of them succeeds.

The `hook_set(Set)` is a parametric object where the parameter is a list of hook objects, interpreted as a set of hook objects. The hook objects are tried in sequence until one of them succeeds in expanding the current

term (goal) into a different term (goal). This parametric object is used when applying multiple independent expansions (i.e., expansions that don't apply to the same terms or goals).

Both the `hook_pipeline(Pipeline)` and `hook_set(Set)` objects assume that the individual hook objects publicly implement the `expanding` protocol (this is the default) as they send `term_expansion/2` and `goal_expansion/2` messages to those objects.

When using a backend Prolog compiler that supports modules, it's also possible to use as parameter a list of hook modules as long as their names do not coincide with the names of loaded objects.

6.26 hook_objects

This library provides a set of convenient hook objects for defining custom expansion workflows (using, e.g., the `hook_flows` library) and for debugging. They are usable and useful as-is but should also be regarded as term- and goal-expansion examples that you can learn from, clone, and change to fit your application requirements.

6.26.1 API documentation

Open the ../../apis/library_index.html#hook-objects link in a web browser.

6.26.2 Loading

To load all hook objects in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(hook_objects(loader)).
```

To load a specific hook object, e.g., the `backend_adapter_hook` object:

```
| ?- logtalk_load(hook_objects(backend_adapter_hook)).
```

6.26.3 Testing

To test this library hook objects, load the `tester.lgt` file:

```
| ?- logtalk_load(hook_objects(tester)).
```

6.26.4 Usage

The provided hook objects cover different expansion scenarios as follows.

Using the Prolog backend adapter file expansion rules

Useful when defining a custom expansion workflow. This can be accomplished by loading the `backend_adapter_hook.lgt` file, which defines a `backend_adapter_hook` hook object that can be used as a workflow step.

Restoring the default compiler expansion workflow

In this case, load the `default_workflow_hook.lgt` file, which defines a `default_workflow_hook` hook object, and use the following **goal** to set the default hook object:

```
| ?- set_logtalk_flag(hook, default_workflow_hook).
```

Preventing applying any (other) user-defined expansion rules

When compiling a source file, we sometimes want to prevent applying expansion rules. This can be accomplished by simply loading the `identity_hook.lgt` file, which defines the `identity_hook` hook object, whose expansion rules simply succeed without changing the terms and goals, and setting it as the file-specific hook object writing as the first term in the file the **directive**:

```
:- set_logtalk_flag(hook, identity_hook).
```

Note that the compiler will always convert any grammar rules defined in the file into clauses. Although this conversion can also be performed as an expansion, grammar rules are part of the Logtalk language. If you want to preserve the grammar rules, use the hook objects described below to write them to a stream.

Expanding grammar rules into clauses independently of the compiler

Load the `grammar_rules_hook.lgt` and use the term-expansion rules in the `grammar_rules_hook` object. For example:

```
| ?- grammar_rules_hook::term_expansion((a --> [b],c), Clause).

Clause = (a([b|T], C) :- c(T, C))
yes
```

Using the expansion rules defined in a Prolog module

Load the `prolog_module_hook.lgt`, which defines the parametric hook object `prolog_module_hook(Module)`. To use this hook object, you need to instantiate the parameter to the name of the module. For example:

```
:- set_logtalk_flag(hook, prolog_module_hook(user)).
```

Wrap the contents of a plain Prolog file as an object

Load the `object_wrapper_hook.lgt`, which defines the `object_wrapper_hook/0-2` hook objects. Use them to wrap the contents of a plain Prolog file as an object named after the file (optionally implementing a protocol) or an object with the given name and object relations. Can be used to apply Logtalk developer tools to plain Prolog code or when porting a Prolog application to Logtalk. For example:

```
| ?- logtalk_load('plain.pl', [hook(object_wrapper_hook)]).
...

| ?- current_object(plain).
yes
```

Or:

```
| ?- logtalk_load('world_1.pl', [hook(object_wrapper_hook(some_protocol))]).
...

| ?- current_object(world_1).
yes

| ?- implements_protocol(world_1, Protocol).
Protocol = some_protocol
yes
```

Or:

```
| ?- logtalk_load('foo.pl', [hook(object_wrapper_hook(bar,[imports(some_category)))]).
...

| ?- current_object(bar).
yes

| ?- imports_category(bar, Category).
Category = some_category
yes
```

The `object_wrapper_hook` object sets the `context_switching_calls` flag to `allow` for the generated object. This enables calling the predicates using the `(<<)/2` context-switching control construct. But it's usually better to define a protocol for the predicates being encapsulated and use instead the `object_wrapper_hook/1-2` objects.

Outputting term-expansion results to a stream or a file

Load the `write_to_stream_hook.lgt` file and use the `write_to_stream_hook(Stream)` or `write_to_stream_hook(Stream, Options)` hook objects. Alternatively, you can load the `write_to_file_hook.lgt` file and use the `write_to_file_hook(File)` or `write_to_file_hook(File, Options)` hook objects. The terms are not modified and thus these hook objects may be used at any point in an expansion workflow. The terms are written followed by a period and a new line.

For example, assume that we want to expand all terms in a `input.pl` source file, writing the resulting terms to a `output.pl` file, using the expansion rules defined in a `expansions` hook object. Taking advantage of the `hook_flows` library `hook_pipeline/1` object, we can write:

```
| ?- logtalk_compile(
        'input.pl',
        [hook(hook_pipeline([
            expansions,
            write_to_file_hook('output.pl')
        ]))]
    ).
```

Printing entity predicate goals before or after calling them

This is helpful for quick debugging. Load the `print_goal_hook.lgt` file and use the `print_goal_hook` hook object. For example, we can set this hook object as the default hook:

```
| ?- set_logtalk_flag(hook, print_goal_hook).
```

Then, edit the entity source code to print selected goals:

```
foo :-
    - bar,   % print goal before calling it
    + baz,   % print goal after calling it
    * quux.  % print goal before and after calling it
```

Suppressing goals

The `suppress_goal_hook.lgt` file provides the `suppress_goal_hook` hook object that supports suppressing a goal in a clause body by prefixing it using the `--` operator. We can set this hook object as the default hook using the goal:

```
| ?- set_logtalk_flag(hook, suppress_goal_hook).
```

If the expansion is only to be used in a single file, use instead the source file directive:

```
:- set_logtalk_flag(hook, suppress_goal_hook).
```

Then, edit entity predicates to suppress goals. For example:

```
foo :-
    bar,
    -- baz,
    quux.
```

The suppressed goals are replaced by calls to `true/0`.

6.27 html

This library provides predicates for generating HTML content using either HTML 5 or XHTML 1.1 formats from a term representation. The library performs minimal validation, checking only that all elements are valid. No attempt is made to generate nicely indented output.

Normal elements are represented using a compound term with one argument (the element content) or two arguments (the element attributes represented by a list of `Key=Value` or `Key-Value` pairs and the element content). The element content can be another element or a list of elements. For example:

```
ol([type-a], [li(foo), li(bar), li(baz)])
```

The two exceptions are the `pre` or `code` elements whose content is never interpreted as an element or a list of elements. For example, the fragment:

```
pre([foo,bar,baz])
```

is translated to:

```
<pre>
[foo,bar,baz]
</pre>
```

Void elements are represented using a compound term with one argument, the (possibly empty) list of attributes represented by a list of `Key=Value` or `Key-Value` pairs. For example:

```
hr([class-separator])
```

Atomic arguments of the compound terms are interpreted as element content. Non-atomic element content can be represented as a quoted atom or by using the `pre` or `code` elements as explained above.

This library is a work in progress.

6.27.1 API documentation

Open the ../../apis/library_index.html#html link in a web browser.

6.27.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(html(loader)).
```

6.27.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(html(tester)).
```

6.27.4 Generating a HTML document

HTML documents can be generated from a compound term representation and written to a file or a stream. For example, assuming we want to generate a HTML 5 file:

```
| ?- html5::generate(
        file('hello.html'),
        html([lang=en], [head(title('Hello world!')), body(p('Bye!'))])
    ).
```

When the second argument is a `html/1` or `html/2` compound term, a *doctype* is automatically written. If we prefer instead e.g. a XHTML 1.1 document, we use the `xhtml11` object:

```
| ?- xhtml11::generate(
        file('hello.html'),
        html([lang=en], [head(title('Hello world!')), body(p('Bye!'))])
    ).
```

6.27.5 Generating a HTML fragment

It's also possible to generate just a fragment of a (X)HTML document by using a list of compound terms or a compound term for an element other than `html`. For example:

```
| ?- current_output(Stream),
    html5::generate(stream(Stream), ul([li(foo), li(bar), li(baz)])).

<ul>
<li>
foo</li>
<li>
bar</li>
<li>
baz</li>
</ul>

Stream = ...
```

6.27.6 Working with callbacks to generate content

Often we need to programmatically generate HTML content from queries. In other cases, we may have fixed content that we don't want to keep repeating (e.g., a navigation bar). The library supports a `(::)/2` pseudo-element that sends a message to an object to retrieve content. As an example, assume the following predicate definition in `user`:

```
content(strong('Hello world!')).
```

This predicate can then be called from the HTML term representation. For example:

```
| ?- current_output(Stream),
     html5::generate(stream(Stream), span(user::content)).

<span><strong>Hello world!</strong></span>

Stream = ...
```

Note that the callback always takes the form `Object::Closure` where `Closure` is extended with a single argument (to be bound to the generated content). More complex callbacks are possible by using lambda expressions.

6.27.7 Working with custom elements

The `html5` and `xhtml11` objects recognize the same set of standard HTML 5 normal and void elements and generate an error for non-standard elements. If you need to generate HTML content containing custom elements, define a new object that extends one of the library objects. For example:

```
:- object(html5custom,
    extends(html5)).

    normal_element(foo, inline).
    normal_element(bar, block).
    normal_element(Name, Display) :-
        ^^normal_element(Name, Display).

:- end_object.
```

6.28 `ids`

This library generates random identifiers given the number of bytes of randomness. The identifiers are Base64 encoded. By default, 20 bytes (160 bits) are used.

The generation of random identifiers uses the `/dev/urandom` random number generator when available. This includes macOS, Linux, *BSD, and other POSIX operating-systems. On Windows, a pseudo-random generator is used, but randomized using the current wall time.

Identifiers can be generated as atoms, lists of characters, or lists of character codes.

See also the `uuid` and `ulid` libraries.

6.28.1 API documentation

Open the ../../apis/library_index.html#ids link in a web browser.

6.28.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(ids(loader)).
```

6.28.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(ids(tester)).
```

6.28.4 Usage

To generate an identifier using the default 160 bits of randomness:

```
| ?- ids::generate(Identifier).
Identifier = '2gpMzqAFXBO5mYFIPX1qMkHxgGE='
yes
```

To generate an identifier represented by an atom using 240 bits (30 bytes) of randomness:

```
| ?- ids(atom, 30)::generate(Identifier).
Identifier = 'ie/jYcLsqo8ZguCOF1ZNPFDRvJ03Ww5Qa9e0FxRB'
yes
```

To generate an identifier represented by a list of characters using 64 bits (8 bytes) of randomness:

```
| ?- ids(chars, 8)::generate(Identifier).
Identifier = ['5','0','8','V',d,'S',c,y,n,o,'A',=]
yes
```

To generate an identifier represented by a list of character codes using 64 bits (8 bytes) of randomness:

```
| ?- ids(codes, 8)::generate(Identifier).
Identifier = [111,81,86,55,99,79,70,77,65,74,103,61]
yes
```

6.29 intervals

This library provides an `intervalp` protocol and an `interval` object that implement basic temporal interval relations protocol (based on the James F. Allen Interval Algebra work).

6.29.1 API documentation

Open the ../../apis/library_index.html#intervals link in a web browser.

6.29.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(intervals(loader)).
```

6.29.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(intervals(tester)).
```

6.30 java

The library Java entities define a minimal abstraction for calling Java from Logtalk. This abstraction makes use of Logtalk parametric objects and allows creating Java objects, accessing Java class fields, and calling Java class and object methods using syntax closer to Logtalk. It also gives access to some Java utility predicates.

This abstraction was developed primarily to work with XVM (with its `jni` plug-in installed) or the JPL library bundled with SWI-Prolog and YAP. However, it's expected to be implementable with alternative Java interfaces found in other backend Prolog compilers. Currently, a preliminary implementation is also available for JIProlog.

The main idea in this abstraction layer is to use parametric objects where the first parameter holds the Java reference (usually to a class or object) and an optional second parameter holds the return value. Together with a forward message handler, this allows the use of Java messages with the same functor and number of arguments as found in the relevant JavaDocs.

6.30.1 API documentation

Open the ../../apis/library_index.html#java link in a web browser.

6.30.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(java(loader)).
```

6.30.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(java(tester)).
```

6.30.4 Usage

The two main objects in this library are `java(Reference, ReturnValue)` and `java(Reference)`. Use the latter if you want to ignore the return value or when calling a void Java method.

The `java` object implements utility predicates. For some backend Java interfaces such as JPL (available in XVM, SWI-Prolog, and YAP) there is also a `java_hook` hook object for removing any overhead when using this library abstraction.

For usage examples and additional tests, see the `clustering`, `document_converter`, `jpl`, and `neo4j` examples.

6.30.5 Known issues

When running Java GUI examples on the macOS Terminal application, you may get a Java error saying that the AWT cannot be started. In alternative, try to run the example from within the SWI-Prolog macOS application instead of using the shell integration script. This issue is due to a macOS Java issue that's orthogonal to both Prolog backends and Logtalk.

6.31 `json`

The `json` library provides predicates for parsing and generating data in the JSON format based on the specification and standard found at:

- https://www.rfc-editor.org/rfc/rfc8259
- https://ecma-international.org/publications-and-standards/standards/ecma-404/

It includes parametric objects whose parameters allow selecting the representation for parsed JSON objects (`curly` or `list`), JSON text strings (`atom`, `chars`, or `codes`) and JSON pairs (`dash`, `equal`, or `colon`).

6.31.1 API documentation

Open the ../../apis/library_index.html#json link in a web browser.

6.31.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(json(loader)).
```

6.31.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(json(tester)).
```

Some of the sample JSON test files are based on examples published at:

https://www.json.org/

6.31.4 Representation

The following choices of syntax have been made to represent JSON elements as terms:

- By default, JSON objects are represented using curly-bracketed terms, `{Pairs}`, where each pair uses the representation `Key-Value` (see below for alternative representations).
- Arrays are represented using lists.
- Text strings can be represented as atoms, `chars(List)`, or `codes(List)`. The default when decoding is to use atoms when using the `json` object. To decode text strings into lists of chars or codes, use the `json/1` with the parameter bound to `chars` or `codes`. For example:

```
| ?- json::parse(codes([34,104,101,108,108,111,34]), Term).
Term = hello
yes

| ?- json(atom)::parse(codes([34,104,101,108,108,111,34]), Term).
Term = hello
yes

| ?- json(chars)::parse(codes([34,104,101,108,108,111,34]), Term).
Term = chars([h,e,l,l,o])
yes

| ?- json(codes)::parse(codes([34,104,101,108,108,111,34]), Term).
Term = codes([104,101,108,108,111])
yes
```

- The JSON values `false`, `true` and `null` are represented by, respectively, the `@false`, `@true` and `@null` compound terms.

The following table exemplifies the term equivalents of JSON elements using default representations for objects, pairs, and strings:

JSON	term
[1,2]	[1,2]
true	@true
false	@false
null	@null
-1	-1
[1.2345]	[1.2345]
[]	[]
[2147483647]	[2147483647]
[0]	[0]
[1234567890123456789]	[1234567890123456789]
[false]	[@false]
[-2147483648]	[-2147483648]
{“a”:null,”foo”:”bar”}	{a-@null, foo-bar}
[2.225073858507201e-308]	[2.225073858507201e-308]
[0,1]	[0,1]
[2.2250738585072014e-308]	[2.2250738585072014e-308]
[1.7976931348623157e+308]	[1.7976931348623157e+308]
[0.0]	[0.0]
[4294967295]	[4294967295]
[-1234567890123456789]	[-1234567890123456789]
[“foo”]	[foo]
[1]	[1]
[null]	[@null]
[-1.2345]	[-1.2345]
[5.0e-324]	[5.0e-324]
[-1]	[-1]
[true]	[@true]
[9223372036854775807]	[9223372036854775807]

For JSON objects that are two possible term representations:

JSON object	term (curly)
{“a”:1, “b”:2, “c”:3}	{a-1, b-2, c-3}
{}	{}

and:

JSON object	term (list)
{“a”:1, “b”:2, “c”:3}	json([a-1, b-2, c-3])
{}	json([])

For JSON pairs that are three possible representations:

JSON object	term (dash)
{“a”:1, “b”:2, “c”:3}	{a-1, b-2, c-3}

and:

JSON object	term (equal)
{“a”:1, “b”:2, “c”:3}	{a=1, b=2, c=3}

and:

JSON object	term (colon)
{“a”:1, “b”:2, “c”:3}	{a:1, b:2, c:3}

By default, the curly-term representation and the dash pair representation are used. The `json/3` parametric object allows selecting the desired representation choices. For example:

```
| ?- json(curly,dash,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSON).
JSON = {a-1, b-2, c-3}
yes

| ?- json(list,equal,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSON).
JSON = json([a=1, b=2, c=3])
yes

| ?- json(curly,colon,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSON).
JSON = {a:1, b:2, c:3}
yes
```

6.31.5 Encoding

Encoding is accomplished using the `generate/2` predicate. For example:

```
| ?- json::generate(codes(Encoding), [a,{b-c}]).
Encoding = [91,34,97,34,44,123,34,98,34,58,34,99,34,125,93]
yes
```

Alternatively:

```
| ?- json::generate(chars(Encoding), [a,{b-c}]).
Encoding = ['[','"',a,'"',',','{','"',b,'"',:,'"',c,'"','}',']']
Yes

| ?- json::generate(atom(Encoding), [a,{b-c}]).
Encoding = '["a",{"b":"c"}]'
Yes
```

Notice that `generate/2` takes, as second argument, a Prolog term that corresponds to the JSON syntax in Prolog and produces the corresponding JSON output in the format specified in the first argument: (`codes(Variable)`, `stream(Stream)`, `file(File)`, `chars(Variable)` or `atom(Variable)`).

6.31.6 Decoding

Decoding is accomplished using the `parse/2` predicate. For example, to decode a given json file:

```
| ?- json::parse(file('simple/roundtrip_array_obj_array.json'), Term).
Term = [{a-[b]}]
yes
```

The `parse/2` predicate first argument must indicate the input source (`codes(Codes)`, `stream(Stream)`, `line(Stream)`, `file(Path)`, `chars(Chars)` or `atom(Atom)`) containing a JSON payload to be decoded into the Prolog term in the second argument.

6.31.7 Known issues

Some tests, notably `parse_simple_valid_files` and `roundtrip_hexadecimals`, fail on backends such as ECLiPSe and GNU Prolog that don't support Unicode.

6.32 `json_lines`

The `json_lines` library provides predicates for parsing and generating data in the JSON Lines format based on the proposal found at:

https://jsonlines.org

It includes parametric objects whose parameters allow selecting the representation for parsed JSON objects (`curly` or `list`), JSON text strings (`atom`, `chars`, or `codes`) and JSON pairs (`dash`, `equal`, or `colon`).

6.32.1 API documentation

Open the ../../apis/library_index.html#json_lines link in a web browser.

6.32.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(json_lines(loader)).
```

6.32.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(json_lines(tester)).
```

Some of the sample JSON test files are based on examples published at:

https://jsonlines.org/

6.32.4 Representation

The following choices of syntax have been made to represent JSON elements as terms:

- By default, JSON objects are represented using curly-bracketed terms, `{Pairs}`, where each pair uses the representation `Key-Value` (see below for alternative representations).
- Arrays are represented using lists.
- Text strings can be represented as atoms, `chars(List)`, or `codes(List)`. The default when decoding is to use atoms when using the `json_lines` object. To decode text strings into lists of chars or codes, use the `json_lines/1` object with the parameter bound to `chars` or `codes`. For example:

```
| ?- json_lines::parse(codes([34,104,101,108,108,111,34]), Terms).
Terms = [hello]
yes

| ?- json_lines(atom)::parse(codes([34,104,101,108,108,111,34]), Terms).
Terms = [hello]
yes

| ?- json_lines(chars)::parse(codes([34,104,101,108,108,111,34]), Terms).
Terms = [chars([h,e,l,l,o])]
yes

| ?- json_lines(codes)::parse(codes([34,104,101,108,108,111,34]), Terms).
Terms = [codes([104,101,108,108,111])]
yes
```

- The JSON values `false`, `true` and `null` are represented by, respectively, the `@false`, `@true` and `@null` compound terms.

The following table exemplifies the term equivalents of JSON elements using default representations for objects, pairs, and strings:

JSON	term
[1,2]	[1,2]
true	@true
false	@false
null	@null
-1	-1
[1.2345]	[1.2345]
[]	[]
[2147483647]	[2147483647]
[0]	[0]
[1234567890123456789]	[1234567890123456789]
[false]	[@false]
[-2147483648]	[-2147483648]
{“a”:null,”foo”:”bar”}	{a-@null, foo-bar}
[2.225073858507201e-308]	[2.225073858507201e-308]
[0,1]	[0,1]
[2.2250738585072014e-308]	[2.2250738585072014e-308]
[1.7976931348623157e+308]	[1.7976931348623157e+308]
[0.0]	[0.0]
[4294967295]	[4294967295]
[-1234567890123456789]	[-1234567890123456789]
[“foo”]	[foo]
[1]	[1]
[null]	[@null]
[-1.2345]	[-1.2345]
[5.0e-324]	[5.0e-324]
[-1]	[-1]
[true]	[@true]
[9223372036854775807]	[9223372036854775807]

For JSON objects that are two possible term representations:

JSON object	term (curly)
{“a”:1, “b”:2, “c”:3}	{a-1, b-2, c-3}
{}	{}

and:

JSON object	term (list)
{“a”:1, “b”:2, “c”:3}	json([a-1, b-2, c-3])
{}	json([])

For JSON pairs that are three possible representations:

JSON object	term (dash)
{“a”:1, “b”:2, “c”:3}	{a-1, b-2, c-3}

and:

JSON object	term (equal)
{“a”:1, “b”:2, “c”:3}	{a=1, b=2, c=3}

and:

JSON object	term (colon)
{“a”:1, “b”:2, “c”:3}	{a:1, b:2, c:3}

By default, the curly-term representation and the dash pair representation are used. The `json/3` parametric object allows selecting the desired representation choices. For example:

```
| ?- json_lines(curly,dash,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSONL).
JSONL = [{a-1, b-2, c-3}]
yes

| ?- json_lines(list,equal,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSONL).
JSONL = [json([a=1, b=2, c=3])]
yes

| ?- json_lines(curly,colon,atom)::parse(atom('{"a":1, "b":2, "c":3}'), JSONL).
JSONL = [{a:1, b:2, c:3}]
yes
```

6.32.5 Encoding

Encoding is accomplished using the `generate/2` predicate. For example:

```
| ?- json_lines::generate(codes(Encoding), [a,{b-c}]).
Encoding = [34,97,34,10,123,34,98,34,58,34,99,34,125,10]
yes
```

Alternatively:

```
| ?- json_lines::generate(chars(Encoding), [a,{b-c}]).
Encoding = ['"',a,'"','\n','{','"',b,'"',:,'"',c,'"','}','\n']
Yes

| ?- json_lines::generate(atom(Encoding), [a,{b-c}]).
Encoding = '"a"\n{"b":"c"}\n'
Yes
```

Notice that `generate/2` takes, as second argument, a Prolog term that corresponds to the JSON syntax in Prolog and produces the corresponding JSON output in the format specified in the first argument: `(codes(Variable)`, `stream(Stream)`, `file(File)`, `chars(Variable)` or `atom(Variable))`.

6.32.6 Decoding

Decoding is accomplished using the `parse/2` predicate. For example, to decode a given json file:

```
| ?- json_lines::parse(file('simple/data.jsonl'), Terms).
Term = [{a-[b]}]
yes
```

The `parse/2` predicate first argument must indicate the input source (`codes(Codes)`, `stream(Stream)`, `line(Stream)`, `file(Path)`, `chars(Chars)` or `atom(Atom)`) containing a JSON payload to be decoded into the Prolog term in the second argument.

6.32.7 Known issues

Some tests may fail on backends such as ECLiPSe and GNU Prolog that don't support Unicode.

6.33 `listing`

This library provides support for listing object dynamic predicate clauses. The predicates must (also) be declared using a scope directive to make them visible to the listing predicates.

6.33.1 API documentation

Open the ../../apis/library_index.html#listing link in a web browser.

6.33.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(listing(loader)).
```

6.33.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(listing(tester)).
```

6.33.4 Usage

This library provides a `listing` category that can be imported by any number of objects. The main predicates are declared public. If you want to restrict their scope, use protected or private import. For example:

```
:- object(data_store,
    imports(private::listing)).

    debug :-
        ^^listing(data/4).
```

(continues on next page)

(continued from previous page)

```
    ...

:- end_object.
```

The `listing` category provides a bare-bones `portray_clause/1` predicate implementation. As this predicate is called (by the `listing/0-1` predicates) using the `(::)/1` control construct, the object importing the category can easily override the inherited definition with its own or with a call to the backend system native implementation of the predicate. For example, assuming a backend that provides `portray_clause/1` as a built-in predicate, we can write:

```
:- object(thing,
    imports(listing)).

    :- uses(user, [portray_clause/1]).
    ...

:- end_object.
```

The main predicate, `listing/1`, accepts as argument a predicate indicator, a non-terminal indicator, or a clause head template (to list only clauses with a matching head).

This library is often useful as a debugging helper. For example, assuming that we want to list dynamic predicate clauses for an object `data` compiled (or created) with the `complements` flag set to `allow`, we can hot patch it to add the `listing` category:

```
| ?- create_category(patch, [extends(listing),complements(data)], [], []).
yes

| ?- data::listing.
...
```

Another example:

```
| ?- create_category(
        patch,
        [extends(listing),complements(data)],
        [public(debug/1)],
        [(debug(Key) :- ::listing(p(Key,Datum)))]
    ).
yes

| ?- data::debug(k42).
...
```

Note that the semantics of complementing categories require that we use the `(::)/2` control construct instead of the `(^^)/2` control construct as a *super* call would be interpreted as made from the complemented object.

6.34 logging

This library provides support for logging events to files.

6.34.1 API documentation

Open the ../../apis/library_index.html#logging link in a web browser.

6.34.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(logging(loader)).
```

6.35 loops

This library provides implementations of several kinds of loops typical of imperative languages.

6.35.1 API documentation

Open the ../../apis/library_index.html#loops link in a web browser.

6.35.2 Loading

To load the main entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(loops(loader)).
```

6.35.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(loops(tester)).
```

6.35.4 Usage

See e.g. the `searching` example.

6.36 meta

This library provides implementations of common meta-predicates. The `meta` object implements common meta-predicates like `map/3` and `fold_left/4`.

See also the `meta_compiler` library, which provides optimized compilation of meta-predicate calls.

6.36.1 API documentation

Open the ../../apis/library_index.html#meta link in a web browser.

6.36.2 Loading

To load the main entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(meta(loader)).
```

6.36.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(meta(tester)).
```

6.36.4 Usage

See e.g. the `metapredicates` example and unit tests.

6.37 meta_compiler

This library supports implementations optimized compilation of meta-calls for the predicates defined in the `meta` library.

6.37.1 API documentation

Open the ../../apis/library_index.html#meta-compiler link in a web browser.

6.37.2 Loading

To load the main entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(meta_compiler(loader)).
```

6.37.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(meta_compiler(tester)).
```

6.37.4 Usage

If `meta_compiler` is the only hook object you are using, you can set it as the default hook object (but note that the optimizations are only applied to entities compiled with the `optimize` flag turned on):

```
| ?- set_logtalk_flag(hook, meta_compiler).
...
```

Otherwise, use the `hook(meta_compiler)` and `optimize(on)` compiler options when compiling and loading the code that you want to optimize. For example:

```
| ?- logtalk_load(my_source_file, [hook(meta_compiler), optimize(on)]).
...
```

See also the `metapredicates_compiled` example and unit tests.

6.37.5 Known issues

Goals in clauses containing parameter variables are currently not expanded by this library and thus not optimized.

6.38 `mutations`

Experimental library. Should not be used in production code. Details can be changed without advance notice.

The `mutations` library provides support for generating random mutations of selected types. The library defines default mutation algorithms for the following basic types:

- `atom`
- `integer`
- `float`
- `compound`
- `list`

The user can add additional mutation algorithms for these or other types by defining objects or categories providing clauses for the `mutation/3` predicate and expanding the entity source files using the `mutations_store` object as the hook object.

This library is expected to eventually be used to support mutation-based *fuzz testing*.

By default, loading this library loads a set of default mutation algorithms. These can be overridden by defining alternative mutations and a custom loader file.

6.38.1 API documentation

Open the ../../apis/library_index.html#mutations link in a web browser.

6.38.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(mutations(loader)).
```

6.38.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(mutations(tester)).
```

6.38.4 Usage

The `mutations` category complements the `type` object and thus its predicates are accessed via this object. For example:

```
| ?- type::mutation(integer, 123, M).
M = 1293
yes

| type::mutation(integer, 123, M).
M = 5123
yes

| type::mutation(integer, 123, M).
M = -123
yes

| type::mutation(integer, 123, M).
M = 23
yes
```

When there are multiple mutation algorithms for a given type, the predicate `type::mutation/3` chooses one of them randomly. We can query the number of mutation algorithms available per type using the `mutations_store::counter/2` predicate:

```
| ?- mutations_store::counter(Type, Count).
Type = atom,
Count = 6 ;
Type = integer,
Count = 7 ;
...
```

Loading this library also loads the `arbitrary` library, which provides `get_seed/1` and `set_seed/1` predicates that can be used to control the pseudo-random number generator.

6.39 nested_dictionaries

This library provides nested dictionary implementations based on private extensions to the `dictionaries` library objects. The representations of a nested dictionary should be regarded as opaque terms and only accessed using the library predicates.

This library is experimental, a work in progress, and future versions can introduce incompatible changes.

6.39.1 API documentation

Open the ../../apis/library_index.html#nested-dictionaries link in a web browser.

6.39.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(nested_dictionaries(loader)).
```

6.39.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(nested_dictionaries(tester)).
```

6.39.4 Usage

First, select the nested dictionary implementation that you want to use. For cases where the number of elements is relatively small and performance is not critical, `nbintree` can be a good choice. For other cases, `navltree` or `nrbtree` are likely better choices. If you want to compare the performance of the implementations, either define an object alias or use a `uses/2` directive so that you can switch between implementations by simply changing the alias definition or the first argument of the directive. Note that you can switch between implementations at runtime without code changes by using a parameter variable in the first argument of a `uses/2` directive.

To create an empty nested dictionary, you can use the `new/1` predicate. For example:

```
| ?- navltree::new(Dictionary).
Dictionary = ...
yes
```

You can also create a new nested dictionary from a *curly bracketed term representation* (see below) by using the predicate `as_nested_dictionary/2`. For example:

```
| ?- navltree::as_nested_dictionary(
       {a-1, b-{c-3, d-{e-7,f-8}}},
       Dictionary
    ).

Dictionary = ...
yes
```

Several predicates are provided to insert, lookup, update, and delete key-value pairs given a list of keys interpreted as an *access path* to a nested dictionary. For example:

```
| ?- navltree::as_nested_dictionary(
        {a-1, b-{c-3, d-{e-7,f-8}}},
        Dictionary
    ),
    navltree::lookup_in([b,d,f], Value, Dictionary).

Dictionary = ...
Value = 8
yes
```

For details on these and other provided predicates, consult the library API documentation.

6.39.5 Curly term representation

To simplify importing and exporting data into a nested dictionary, the library provides `as_nested_dictionary/2` and `as_curly_bracketed/2` predicates that work with a *curly term representation*. This format is based on the JSON data interchange format.

A dictionary is represented by the `{Pairs}` term where Pairs is a conjunction of `Key-Value` or `Key:Value` pairs and `Value` can be a nested dictionary or lists of pairs. An empty dictionary is represented using the `{}` term.

6.40 optionals

This library provides an implementation of *optional terms* with an API modeled after the Java 8 `Optional` class (originally due to requests by users working in Logtalk/Java hybrid applications). An optional term is an opaque compound term that may or may not hold a value. Optional terms avoid forcing the user to define a representation for the absence of a value by providing an API with predicates that depend on the presence or absence of a value. Optional terms also allow separating the code that constructs optional terms from the code that processes them, which is then free to deal if necessary and at its convenience with any case where the values held by optional terms are not present.

6.40.1 API documentation

Open the ../../apis/library_index.html#optionals link in a web browser.

6.40.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(optionals(loader)).
```

6.40.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(optionals(tester)).
```

6.40.4 Usage

The `optional` object provides constructors for optional terms. For example:

```
| ?- optional::of(1, Optional).
...
```

The created optional terms can then be passed as parameters to the `optional/1` parametric object. For example:

```
| ?- optional::of(1, Optional), optional(Optional)::or_else(Term, 0).
Optional = optional(1),
Term = 1
yes

| ?- optional::empty(Optional), optional(Optional)::or_else(Term, 0).
Optional = empty,
Term = 0
yes
```

The `maybe` object provides types and predicates for type-checking of the term held by optional terms. It also provides some predicates for handling lists of optional terms.

6.40.5 See also

The `expecteds` library.

6.41 `options`

This library provides useful predicates for managing developer tool and application options.

6.41.1 API documentation

Open the ../../apis/library_index.html#options file in a web browser.

6.41.2 Loading

To load all entities in this library, load the `loader.lgt` utility file:

```
| ?- logtalk_load(options(loader)).
```

6.41.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(options(tester)).
```

6.41.4 Usage

The `options` category is usually imported by the root object of the developer tool or application. The importing object should define the `default_option/1` predicate and, if option type-checking is required, the `valid_option/1` predicate must be defined for each option. This library requires options to be represented by compound terms where the functor is the option name (e.g., `trim(true)` or `box(0,2)`). The `option/2-3` can be used to get or test an option given a list of options. When an option appears multiple times in a list, the `option/2-3` predicates get or test the first (leftmost) occurrence.

The library also supports a user-defined `fix_option/2` predicate. A usage example is when an option value can be a relative file path that should be expanded before use. Another usage example would be converting from a user-friendly option to a form more suitable for internal processing. When a call to the `fix_option/2` predicate fails, the option is used as-is.

A simple example:

```
:- object(foo,
    imports(options)).

    :- uses(type, [
        valid/2
    ]).

    :- public(p/0).
    p :-
        % use default options
        p([]).

    :- public(p/1).
    p(UserOptions) :-
        ^^check_options(UserOptions),
        % construct the full set of options from
        % the user options and the default options
        ^^merge_options(UserOptions, Options),
        ...
        % query an option
        ^^option(baz(Boolean), Options),
        q(Boolean),
        ...
```

(continues on next page)

(continued from previous page)

```
    default_option(baz(true)).
    ...

    valid_option(baz(Boolean)) :-
        valid(boolean, Boolean).
    ...

:- end_object.
```

Note that you can use protected or private import of the `options` category if you don't want to add its public predicates to the object protocol.

6.42 os

This library provides a *portable* operating-system interface for the supported backend Prolog compilers.

The `os_types` category defines some useful operating-system types for type-checking when using with the `type` library object.

6.42.1 API documentation

Open the ../../apis/library_index.html#os link in a web browser.

6.42.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(os(loader)).
```

6.42.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(os(tester)).
```

6.42.4 Known issues

Some predicates may only be supported by a subset of backend Prolog compilers on a subset of operating-systems. They should be used with care and fully tested in your application domain, as some backend Prolog compilers have buggy and inconsistent interfaces, notably across operating-systems. See the remarks section in the `os` object documentation for details.

6.43 queues

This library implements queues. The queue representation should be regarded as an opaque term and only accessed using the library predicates.

6.43.1 API documentation

Open the ../../apis/library_index.html#queues link in a web browser.

6.43.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(queues(loader)).
```

6.43.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(queues(tester)).
```

6.43.4 Usage

To create a new queue, use the `new/1` predicate:

```
| ?- queue::new(Queue).
Queue = ...
yes
```

Elements can be added to either the end of the queue or the front of the queue using, respectively, the `join/3` and `join_all/3` predicates or the `jump/3` and `jump_all/3`. For example:

```
| ?- queue::(new(Queue0), join_all([1,2,3], Queue0, Queue1)).
Queue0 = ...,
Queue1 = ...
yes
```

We can query the head of the queue or remove the head of the queue using, respectively, the `head/2` and `serve/3` predicates. For example:

```
| ?- queue::(new(Queue0), join(1, Queue0, Queue1), head(Queue1, Head)).
Queue0 = ...,
Queue1 = ...,
Head = 1
yes
```

For details on these and other provided predicates, consult the library API documentation.

6.44 random

This library provides portable random number generators and an abstraction over the native backend Prolog compiler random number generator if available. It includes predicates for generating random floats and random integers in a given interval; predicates for generating random sequences and sets; predicates for randomly selecting, enumerating, and swapping elements from a list; predicates that succeed, fail, or call another predicate with a given probability; and predicates for sampling common probability distributions.

6.44.1 API documentation

Open the ../../apis/library_index.html#random link in a web browser.

6.44.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(random(loader)).
```

6.44.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(random(tester)).
```

6.44.4 Usage

The `random` object implements a portable random number generator and supports multiple random number generators, using different seeds, by defining derived objects. For example:

```
:- object(my_random_generator_1,
    extends(random)).

    :- initialization(::reset_seed).

:- end_object.
```

The `fast_random` object also implements a portable random number generator but does not support deriving multiple random number generators, which makes it a bit faster than the `random` object.

The `random` and `fast_random` objects manage the random number generator seed using internal dynamic state. The predicates that update the seed are declared as synchronized (when running on Prolog backends that support threads). Still, care must be taken when using these objects from multi-threaded applications, as there is no portable solution to protect seed updates from signals and prevent inconsistent state when threads are canceled.

The `random` and `fast_random` objects always initialize the random generator seed to the same value, thus providing a pseudo random number generator. The `randomize/1` predicate can be used to initialize the seed with a random value. The argument should be a large positive integer. In alternative, when using a small integer argument, discard the first dozen random values.

The backend_random object abstracts the native backend Prolog compiler random number generator for the basic random/1, get_seed/1, and set_seed/1 predicates providing a portable implementation for the remaining predicates. This makes the object stateless, which allows reliable use from multiple threads. Consult the backend Prolog compiler documentation for details on its random number generator properties. Note that several of the supported backend Prolog systems, notably B-Prolog, CxProlog, ECLiPSe, JIProlog, and Quintus Prolog, do not provide implementations for both the get_seed/1 and set_seed/1 predicates and calling these predicates simply succeed without performing any action.

All random objects (random, fast_random, and backend_random) implement the sampling_protocol protocol. To maximize performance, the shared implementations of the sampling predicates is defined in the sampling.lgt file that's included in the random objects. This allows these predicates to call the basic random/1 and random/3 predicates as locally defined predicates.

6.45 reader

The reader object provides portable predicates for reading text file and text stream contents to lists of terms, characters, or character codes and for reading binary files to lists of bytes. The text file API is loosely based on the SWI-Prolog readutil module.

6.45.1 API documentation

Open the ../../apis/library_index.html#reader link in a web browser.

6.45.2 Loading

To load all entities in this library, load the loader.lgt file:

```
| ?- logtalk_load(reader(loader)).
```

6.45.3 Testing

To test this library predicates, load the tester.lgt file:

```
| ?- logtalk_load(reader(tester)).
```

6.46 recorded_database

The recorded_database library aims to help port Prolog code using the legacy recorded database. Ported applications should still consider migrating to more standard solutions to handle dynamic data that must survive backtracking.

6.46.1 API documentation

Open the ../../apis/library_index.html#recorded_database link in a web browser.

6.46.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(recorded_database(loader)).
```

6.46.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(recorded_database(tester)).
```

6.46.4 Usage

The `recorded_database_core` category implements the library predicates. This category is imported by the default `recorded_database` object to provide application global database. To make the database local and thus minimize potential record clashes, the category can be imported by one or more application objects. Use protected or private import to restrict the scope of the library predicates. For example:

```
:- object(foo,
    imports(private::recorded_database_core)).

    bar :-
        ^^recorda(key, value(1)),
        ...

:- end_object.
```

6.46.5 Known issues

Currently, references are non-negative integers. They still must be regarded as opaque terms and subject to change without notice. But using integers can result in integer overflows when running on backends with bounded integers in applications performing a large number of database updates.

6.47 `redis`

Redis client library. Supports GNU Prolog, XVM, SICStus Prolog, SWI-Prolog, and XSB. Support for Ciao Prolog and ECLiPSe is also included but requires fixes for issues in these systems.

For general information on Redis, including a list of the available commands, visit:

```
https://redis.io
```

6.47.1 API documentation

Open the ../../apis/library_index.html#redis link in a web browser.

6.47.2 Loading

To load this library, load the `loader.lgt` file:

```
| ?- logtalk_load(redis(loader)).
```

6.47.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(redis(tester)).
```

The tests assume a localhost Redis server running on the default port (6379) if the `REDIS_HOST` and `REDIS_PORT` environment variables are not defined. If the server is not detected, the tests are skipped.

The unit tests were originally written by Sean Charles for his GNU Prolog Redis client library:

```
https://github.com/emacstheviking/gnuprolog-redisclient
```

The Logtalk version is a straight-forward port of the original tests using the `test/1` dialect of `lgtunit`.

6.47.4 Credits

This library is inspired by the Sean Charles GNU Prolog Redis client library.

6.47.5 Known issues

Recent versions of macOS seem to disable the mapping of `localhost` to `127.0.0.1`. This issue may prevent running this library unit tests and the `redis::connect/1` predicate from working. This can be fixed either by editing the `/etc/hosts file or by using in alternative the predicateredis::connect/3with'127.0.0.1'`` as the first argument.

6.48 sets

This library provides a set protocol and two implementations of this protocol using *ordered lists*, one of them a parametric object that takes the type of the set elements as a parameter. Although representing sets as ordered lists is a common representation, the best practice is to regard sets as opaque terms and only access them using the library predicates.

6.48.1 API documentation

Open the ../../apis/library_index.html#sets link in a web browser.

6.48.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(sets(loader)).
```

6.48.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(sets(tester)).
```

6.48.4 Usage

First, select a set implementation. Use the `set(Type)` object if you want to type-check the set elements. Otherwise, use the `set` object.

To create a new set, you can use the `new/1` predicate. For example:

```
| ?- set::new(Set).
Set = []
yes
```

You can also create a new set with all unique elements from a list of terms by using the `as_set/2` predicate. For example:

```
| ?- set::as_set([1,3,2,1,2], Set).
Set = [1, 2, 3]
yes
```

Predicates are provided for the most common set operations. For example:

```
| ?- set::(
        as_set([1,3,2,1,2], Set1),
        as_set([7,4,2,5,1], Set2),
        intersection(Set1, Set2, Intersection),
        symdiff(Set1, Set2, Difference)
    ).
Set1 = [1, 2, 3],
Set2 = [1, 2, 4, 5, 7],
Intersection = [1, 2],
Difference = [3, 4, 5, 7]
yes
```

When working with a custom type of set elements, the corresponding object must implement the `comparingp` protocol. For example:

```
:- object(rainbow_colors,
    implements(comparingp)).

    order(red,    1).
    order(orange, 2).
    order(yellow, 3).
    order(green,  4).
    order(blue,   5).
    order(indigo, 6).
    order(violet, 7).

    Color1 < Color2 :-
        order(Color1, N1),
        order(Color2, N2),
        {N1 < N2}.

    Color1 =< Color2 :-
        order(Color1, N1),
        order(Color2, N2),
        {N1 =< N2}.

    ...

:- end_object.
```

We can then use this object with the `set/1` parametric object. For example:

```
| ?- set(rainbow_colors)::as_set([blue, yellow, violet], Set).
Set = [yellow, blue, violet]
yes
```

For details on these and other provided predicates, consult the library API documentation.

6.48.5 Credits

Some predicates adapted from code authored by Richard O'Keefe.

6.49 statistics

The entities in this library define some useful predicates for descriptive statistics. Data is represented as a list of numbers (integers or floats). Use the object `sample` of your data represents a sample. Use the object `population` if your data represents a population.

6.49.1 API documentation

Open the ../../apis/library_index.html#statistics link in a web browser.

6.49.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(statistics(loader)).
```

6.49.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(statistics(tester)).
```

6.50 term_io

This library implements predicates for reading/writing terms from/to atoms, chars (lists of characters), and codes (lists of character codes). These predicates are implemented using a single temporary file created when the library is loaded. This temporary file is unique per Logtalk process. The predicates can be safely used in multi-threaded applications.

6.50.1 API documentation

Open the ../../apis/library_index.html#term-io link in a web browser.

6.50.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(term_io(loader)).
```

6.50.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(term_io(tester)).
```

6.51 `timeout`

The `timeout` object provides a portable abstraction over calling a goal deterministically with a time limit as made available in some form by some of the supported backend Prolog systems (B-Prolog, ECLiPSe, XVM, SICStus Prolog, SWI-Prolog, Trealla Prolog, XSB, and YAP).

For better performance, compile calls to this library meta-predicates with the `optimize` flag turned on so that the meta-arguments, i.e. the goals that you are timing, are also compiled.

6.51.1 API documentation

Open the ../../apis/library_index.html#timeout link in a web browser.

6.51.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(timeout(loader)).
```

6.51.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(timeout(tester)).
```

6.51.4 Known issues

Two tests are currently skipped when using the SWI-Prolog backend as they cannot be interrupted and generate the expected timeout exceptions due to tests being run from an `initialization/1` directive goal that, in the SWI-Prolog implementation of this directive, ignores signals. The test goals do generate the expected timeout exception when not called from an `initialization/1` directive.

6.52 `tsv`

The `tsv` library provides predicates for reading and writing TSV files and streams:

https://www.iana.org/assignments/media-types/text/tab-separated-values

The main object, `tsv/1`, is a parametric object allowing passing a single option for the handling of the header of the file (`keep` or `skip`). The `tvs` object extends the `tsv/1` parametric object using the default `keep` option value.

Files and streams can be read into a list of rows (with each row being represented by a list of fields) or asserted using a user-defined dynamic predicate. Reading can be done by first loading the whole file (using the `read_file/2-3` predicates) into memory or line by line (using the `read_file_by_line/2-3` predicates). Reading line by line is usually the best option for parsing large TSV files.

Data can be saved to a TSV file or stream by providing the object and predicate for accessing the data plus the name of the destination file or the stream handle or alias.

6.52.1 API documentation

Open the ../../apis/library_index.html#tsv link in a web browser.

6.52.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(tsv(loader)).
```

6.52.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(tsv(tester)).
```

6.52.4 Usage

A TSV file can be read as a list of rows:

```
| ?- tsv::read_file('test_files/data.tsv', Rows).

Rows = [['Name','Age','Address'], ['Paul',23,'1115 W Franklin'], ['Bessy the Cow',5,'Big␣
↪Farm Way'], ['Zeke,45,'W Main St']]
yes
```

Alternatively, The TSV data can be saved using a public and dynamic object predicate (that must be previously declared). For example:

```
| ?- assertz(p(_,_,_)), retractall(p(_,_,_)).
yes

| ?- tsv(skip)::read_file('test_files/data.tsv', user, p/3).
yes

| ?-  p(A,B,C).

A = 'Paul', B = 23, C = '1115 W Franklin' ? ;
...
```

Given a predicate representing a table, the predicate data can be written to a file or stream. For example:

```
| ?- tsv::write_file('output.tsv', user, p/3).
yes
```

6.53 types

This library implements predicates over standard Prolog term types and also terms representing common data structures such as lists and pairs.

It also includes a user-extensible `type` object defining type checking predicates over common Logtalk and Prolog term types. The types define a hierarchy with the Prolog type `term` at the root (i.e., type-checking a predicate argument of type `term` trivially succeeds). Some types are only meaningful for backend Prolog systems supporting non-universal features (e.g., `cyclic` or `char(CharSet)` with a Unicode character set). See the API documentation for a full list of the types defined by default.

6.53.1 API documentation

Open the ../../apis/library_index.html#types link in a web browser.

6.53.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(types(loader)).
```

If your code only requires the most basic types, you can load in alternative the file:

```
| ?- logtalk_load(basic_types(loader)).
```

See the notes on the `basic_types` virtual library for details.

6.53.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(types(tester)).
```

6.53.4 Type-checking

This library `type` object can be used to type-check common Logtalk and Prolog term types (see the object documentation for a listing of all the pre-defined types). The `valid/2` predicate succeeds or fails if a term is of a given type. For example:

```
| ?- type::valid(positive_integer, 42).
yes

| ?- type::valid(positive_integer, -13).
no
```

The `check/2` and `check/3` predicates throw an exception if a term is not of a given type. For example:

```
| ?- catch(type::check(integer, abc), Error, true).
Error = type_error(integer, abc)
yes
```

If we require a standard `error/2` exception term, the `check/3` predicate takes a *context* argument. For example:

```
| ?- catch(type::check(integer, abc, foo/3), Error, true).
Error = error(type_error(integer, abc), foo/3)
yes
```

Typically, the context is provided by calling the built-in `context/1` method.

6.53.5 Defining new types

To define a custom type, define clauses for the multifile predicates `type::type/1` (to declare the type) and `type::check/2` (to type-check values). For example:

```
:- multifile(type::type/1).
type::type(age).

:- multifile(type::check/2).
type::check(age, Term) :-
    type::check(between(non_negative_integer, 0, 150), Term).
```

Be careful to ensure that the new type definitions don't introduce spurious choice-points for these predicates. The unit tests of the `types` library perform this check for pre-defined and loaded user-defined ground types.

When defining a meta-type (i.e., a type with arguments that are also types), add also a clause for the `type::meta_type/3` multifile predicate. For example:

```
:- multifile(type::meta_type/3).
type::meta_type(tuple(Type1, Type2, Type3), [Type1, Type2, Type3], []).
```

This predicate is called when checking if a type is a defined type. For meta-types, that check must extend to the sub-types.

6.53.6 Examples

See e.g. the `os` library implementation of custom types for files and directories. Or the `expecteds` and `optionals` libraries custom types. See also the `my_types` programming example.

6.54 `ulid`

This library implements a Universally Unique Lexicographically Sortable Identifier (ULID) generator.

https://github.com/ulid/spec

Note that most backends provide time stamps with lower granularity than required (i.e., seconds but not milliseconds). Also note that, per spec, within the same millisecond, monotonic sort order is not guaranteed.

The generation of ULIDs uses the `/dev/urandom` random number generator when available. This includes macOS, Linux, *BSD, and other POSIX operating-systems. On Windows, a pseudo-random generator is used, but randomized using the current wall time.

ULIDs can be generated as atoms, lists of characters, or lists of character codes.

See also the `ids` and `uuid` libraries.

6.54.1 API documentation

Open the ../../apis/library_index.html#ulid link in a web browser.

6.54.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(ulid(loader)).
```

6.54.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(ulid(tester)).
```

6.54.4 Generating ULIDs

By default, ULIDs are generated as atoms. For example:

```
| ?- ulid::generate(ULID).
ULID = '01H0J31SYQXHJZWPRAKHQ6YVYH'
yes
```

To generate a ULID using a list of characters representation, use instead the `ulid/1` parametric object:

```
| ?- ulid(chars)::generate(ULID).
ULID = ['0','1','H','0','J','3','2','Y','V','5','V','S','P','K','5','P','4','5','G','G','0',
↪'9','8','8','M','2']
yes
```

Similarly, to get a ULID using a list of character codes representation:

```
| ?- ulid(codes)::generate(ULID).
ULID = [48,49,72,48,74,51,52,66,54,48,55,57,54,49,67,82,70,65,67,51,67,67,86,82,48,66]
yes
```

It's also possible to generate ULIDs from a given timestamp, i.e. the number of milliseconds since the Unix epoch (00:00:00 UTC on January 1, 1970), using the `generate/2` predicate. The timestamp must be an integer. For example, assuming a backend Prolog system providing a `time_stamp/1` predicate returning the Unix epoch in milliseconds:

```
| ?- time_stamp(Milliseconds), ulid(atom)::generate(Seconds, ULID).
Seconds = 1684245175344, ULID = '01H0JDBQ1GAWJF35C44Y5S97DX'
yes
```

You can also use timestamp discrete components using the `generate/8` predicate. For example:

```
| ?- ulid(atom)::generate(2023, 5, 17, 16, 23, 38, 591, ULID).
ULID = '01H0N8CDAZK75C5H3BJSGS4VCQ'
yes
```

To extract the timestamp from a given ULID, use the `timestamp/2` and `timestamp/8` predicates. For example:

```
| ?- ulid(atom)::timestamp('01H0JDBQ1GAWJF35C44Y5S97DX', Milliseconds).
Milliseconds = 1684245175344
yes

| ?- ulid(atom)::timestamp('01H0N8CDAZK75C5H3BJSGS4VCQ', Year, Month, Day, Hours, Minutes,␣
↪Seconds, Milliseconds).
Year = 2023, Month = 5, Day = 17, Hours = 16, Minutes = 23, Seconds = 38, Milliseconds = 591
yes
```

6.54.5 Type-checking ULIDs

This library also defines a `ulid(Representation)` type for type-checking ULIDs. For example, with an atom containing invalid characters:

```
| ?- type::check(ulid(atom), '01BX5ZIKBKALTAV9OEVGEMMVRY').
uncaught exception: domain_error(ulid,'01BX5ZIKBKALTAV9OEVGEMMVRY')
```

6.55 unicode_data

VivoMind Prolog Unicode Resources

6.55.1 Authors

Arun Majumdar @ VivoMind LLC
Paulo Moura @ VivoMind LLC

6.55.2 License

Creative Commons CC0 1.0 Universal (CC0 1.0) - Public Domain Dedication:

```
https://creativecommons.org/publicdomain/zero/1.0/
```

We do appreciate acknowledgment if you use these resources, however, and we also welcome contributions to improve them.

6.55.3 Website

The latest release of the VivoMind Prolog Unicode Resources is available at the URL:

```
https://github.com/VivoMind
```

At this address you can also find additional information about the VivoMind Prolog Unicode Resources and submit your bug reports and contributions.

6.55.4 Description

The VivoMind Prolog Unicode Resources are a set of files resulting from the conversion of most (but not all) official UCD 6.1 files and updated for the few changes in the 6.2 standard. The original files can be downloaded from:

```
http://www.unicode.org
```

The conversion of the UCD files resulted in a large number of Prolog tables and also a set of auxiliary predicates (described below) for accessing these tables. Other than the obvious conversion in the provided predicate names, no attempt was made to convert the identifiers used for properties and other data.

6.55.5 Requirements

Most of the auxiliary predicates assume that the de facto Prolog standard predicate `between/3` is available. Unicode code point values are represented using the ISO Prolog standard notation for hexadecimal integers. In addition, the ISO Prolog standard directives `include/1` and `ensure_loaded/1` are used in some of the files to load auxiliary files.

6.55.6 Usage

Most applications only require some of the tables present in these resources. Most of these tables define properties for ranges of code points and not for single code points, but the provided auxiliary predicates allow access for a single code point. When increased performance is required, consider using the existing tables and auxiliary predicates to generate derived tables more fit for your specific application.

6.55.7 Known issues

In the file `unicode_unihan_variant.pl`, when there's more than one variant for a code point, only the first one (as listed in the original UCD file) is returned.

The `include/1` and `ensure_loaded/1` directives are specified in the ISO Prolog standard published in 1995. But some Prolog compilers either don't implement one or both directives or have flawed implementations. Thus, you may need to change how some of the files are loaded depending on the chosen Prolog compiler. Using conditional compilation directives would help in some cases, but it would also raise portability issues on its own.

6.55.8 Acknowledgements

We thank Richard O'Keefe for helpful suggestions to improve the usability of these resources.

6.55.9 Files and API Summary

The Prolog file names are derived from the original file names by prefixing them with the `unicode_` string, converting to lower case, and replacing the camel case spelling with underscores. There are, however, two exceptions: the files and directories holding the code point categories and names.

There's also a utility file, `unicode_data.pl`, that can be used to load all the files in these resources. It is mostly used to test the portability of the code across Prolog compilers. Also included is a Logtalk version of this file, `unicode_data.lgt`, which uses Logtalk's own implementation of the `include/1` directive and the `logtak_load/1` predicate to load all files. This file can be used to workaround Prolog systems with buggy or missing implementations of the `ensure_loaded/1` and `include/1` directives.

An overview of the original file names and the code point properties can be found at:

```
http://www.unicode.org/reports/tr44/#Directory_Structure

http://www.unicode.org/reports/tr44/#Property_Definitions
```

unicode_arabic_shaping.pl

- Provides: `unicode_arabic_shaping/4`
- Dependencies: (none)

unicode_bidi_mirroring.pl

- Provides: `unicode_bidi_mirroring/2`
- Dependencies: (none)

unicode_blocks.pl

- Provides: `unicode_block/2-3`
- Dependencies: (none)

unicode_case_folding.pl

- Provides: `unicode_case_folding/3`
- Dependencies: (none)

unicode_categories.pl

- Provides: `unicode_category/2`
- Dependencies: files in the `unicode_categories` directory

unicode_cjk_radicals.pl

- Provides: `unicode_cjk_radical/3`
- Dependencies: (none)

unicode_composition_exclusions.pl

- Provides: `unicode_composition_exclusion/1`
- Dependencies: (none)

unicode_core_properties.pl

- Provides: `unicode_math/1-2` `unicode_alphabetic/1-2` `unicode_range_alphabetic/2` `unicode_lowercase/1-2` `unicode_uppercase/1-2` `unicode_cased/1-2` `unicode_case_ignorable/1-2` `unicode_changes_when_lowercased/1-2` `unicode_changes_when_uppercased/1-2` `unicode_changes_when_titlecased/1-2` `unicode_changes_when_casefolded/1-2` `unicode_changes_when_casemapped/1-2` `unicode_id_start/1-2` `unicode_id_continue/1-2` `unicode_xid_start/1-2` `unicode_xid_continue/1-2` `unicode_default_ignorable/1-2` `unicode_grapheme_extend/1-2` `unicode_grapheme_base/1-2` `unicode_grapheme_link/1-2`
- Dependencies: files in the `unicode_core_properties` directory

unicode_decomposition_type.pl

- Provides: `unicode_canonical/1-2` `unicode_compat/1-2` `unicode_font/1-2` `unicode_nobreak/1-2` `unicode_initial/1-2` `unicode_medial/1-2` `unicode_final/1-2` `unicode_isolated/1-2` `unicode_circle/1-2` `unicode_super/1-2` `unicode_sub/1-2` `unicode_vertical/1-2` `unicode_wide/1-2` `unicode_narrow/1-2` `unicode_small/1-2` `unicode_square/1-2` `unicode_fraction/1-2`
- Dependencies: files in the `unicode_decomposition_type` directory

unicode_derived_age.pl

- Provides: `unicode_age/2-3`
- Dependencies: (none)

unicode_derived_bidi_class.pl

- Provides: `unicode_bidi_class/2-3`
- Dependencies: (none)

unicode_derived_combining_class.pl

- Provides: `unicode_combining_class/2-3`
- Dependencies: (none)

unicode_derived_core_properties.pl

- Provides: `unicode_core_property/2-3`
- Dependencies: (none)

unicode_derived_decomposition_type.pl

- Provides: `unicode_decomposition_type/2-3`
- Dependencies: (none)

unicode_derived_east_asian_width.pl

- Provides: `unicode_east_asian_width/2-3`
- Dependencies: (none)

unicode_derived_joining_group.pl

- Provides: `unicode_joining_group/2-3`
- Dependencies: (none)

unicode_derived_joining_type.pl

- Provides: `unicode_joining_type/2-3`
- Dependencies: (none)

unicode_derived_line_break.pl

- Provides: `unicode_line_break/2-3`
- Dependencies: (none)

unicode_derived_normalization_props.pl

- Provides: `unicode_fc_nfkc/2` `unicode_nfkc_cf/2` `unicode_full_composition_exclusion/1-2` `unicode_nfd_qc_no/1-2` `unicode_nfc_qc_no/1-2` `unicode_nfc_qc_maybe/1-2` `unicode_nfkd_qc_no/1-2` `unicode_nfkc_qc_no/1-2` `unicode_nfkc_qc_maybe/1-2` `unicode_expands_on_nfd/1-2` `unicode_expands_on_nfc/1-2` `unicode_expands_on_nfkd/1-2` `unicode_expands_on_nfkc/1-2` `unicode_changes_when_nfkc_casefolded/1-2`
- Dependencies: files in the `unicode_derived_normalization_props` directory

unicode_derived_numeric_type.pl

- Provides: `unicode_numeric_type/2-3`
- Dependencies: (none)

unicode_derived_numeric_values.pl

- Provides: `unicode_numerical_value/3`
- Dependencies: (none)

unicode_hangul_syllable_type.pl

- Provides: `unicode_hangul_syllable_type/2-3`
- Dependencies: (none)

unicode_indic_matra_category.pl

- Provides: `unicode_indic_matra_category/2-3`
- Dependencies: (none)

unicode_indic_syllabic_category.pl

- Provides: `unicode_indic_syllabic_category/2-3`
- Dependencies: (none)

unicode_jamo.pl

- Provides: `unicode_jamo/2`
- Dependencies: (none)

unicode_name_aliases.pl

- Provides: `unicode_name_alias/3`
- Dependencies: (none)

unicode_names.pl

- Provides: `unicode_name/2`
- Dependencies: files in the `unicode_names` directory

unicode_prop_list.pl

- Provides: `unicode_white_space/1-2` `unicode_bidi_control/1-2` `unicode_join_control/1-2` `unicode_dash/1-2` `unicode_hyphen/1-2` `unicode_quotation_mark/1-2` `unicode_terminal_punctuation/1-2` `unicode_other_math/1-2` `unicode_hex_digit/1-2` `unicode_ascii_hex_digit/1-2` `unicode_other_alphabetic/1-2` `unicode_ideographic/1-2` `unicode_diacritic/1-2` `unicode_extender/1-2` `unicode_other_lowercase/1-2` `unicode_other_uppercase/1-2` `unicode_noncharacter_code_point/1-2` `unicode_other_grapheme_extend/1-2` `unicode_ids_binary_operator/1-2` `unicode_ids_trinary_operator/1-2` `unicode_radical/1-2` `unicode_unified_ideograph/1-2` `unicode_other_default_ignorable/1-2` `unicode_deprecated/1-2` `unicode_soft_dotted/1-2` `unicode_logical_order_exception/1-2` `unicode_other_id_start/1-2` `unicode_other_id_continue/1-2` `unicode_sterm/1-2` `unicode_variation_selector/1-2` `unicode_pattern_white_space/1-2` `unicode_pattern_syntax/1-2`
- Dependencies: files in the `unicode_prop_list` directory

unicode_range_scripts.pl

- Provides: `unicode_range_script/3` `unicode_script/2`
- Dependencies: (none)

unicode_script_extensions.pl

- Provides: `unicode_script_extension/2-3`
- Dependencies: `unicode_scripts.pl`

unicode_scripts.pl

- Provides: `unicode_script/6` `unicode_script_category/3`
- Dependencies: (none)

unicode_special_casing.pl

- Provides: `unicode_special_casing/5`
- Dependencies: (none)

unicode_unihan_variants.pl

- Provides: `unicode_unihan_variant/2-3`
- Dependencies: (none)

unicode_version.pl

- Provides: `unicode_version/3`
- Dependencies: (none)

6.56 union_find

This library implements a union-find data structure. This structure tracks a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides fast operations to add new sets, to merge existing sets, and to determine whether elements are in the same set. This implementation of the union-find algorithm provides the following features:

- Path compression: Path compression flattens the structure of the tree by making every node point to the root whenever a find predicate is used on it.
- Union by rank: Union predicates always attach the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the original unless they were of equal height, in which case the resulting tree is taller by one node.

For a general and extended discussion on this data structure, see e.g.

https://en.wikipedia.org/wiki/Disjoint-set_data_structure

6.56.1 API documentation

Open the ../../apis/library_index.html#union-find link in a web browser.

6.56.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(union_find(loader)).
```

6.56.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(union_find(tester)).
```

6.56.4 Usage

An usage example is Kruskal's algorithm, a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph by adding increasing cost arcs at each step.

```
:- object(kruskal).

    :- public(kruskal/2).

    :- uses(union_find, [
        new/2, find/4, union/4
    ]).

    kruskal(g(Vertices-Edges), g(Vertices-Tree)) :-
        new(Vertices, UnionFind),
        keysort(Edges, Sorted),
        kruskal(UnionFind, Sorted, Tree).

    kruskal(_, [], []).
    kruskal(UnionFind0, [Edge| Edges], [Edge| Tree]) :-
        Edge = _-(Vertex1, Vertex2),
        find(UnionFind0, Vertex1, Root1, UnionFind1),
        find(UnionFind1, Vertex2, Root2, UnionFind2),
        Root1 \== Root2,
        !,
        union(UnionFind2, Vertex1, Vertex2, UnionFind3),
        kruskal(UnionFind3, Edges, Tree).
    kruskal(UnionFind, [_| Edges], Tree) :-
        kruskal(UnionFind, Edges, Tree).

:- end_object.
```

Sample query:

```
| ?- kruskal::kruskal(g([a,b,c,d,e,f,g]-[7-(a,b), 5-(a,d), 8-(b,c), 7-(b,e), 9-(b,d), 5-(c,
↪e), 15-(d,e), 6-(d,f), 8-(e,f), 9-(e,g), 11-(f,g)]), Tree).

Tree = g([a,b,c,d,e,f,g]-[5-(a,d),5-(c,e),6-(d,f),7-(a,b),7-(b,e),9-(e,g)])
yes
```

6.57 uuid

This library implements a Universally Unique Identifier (UUID) generator. Currently only version 1 and version 4 UUIDs are supported. For reference material, see e.g.

https://en.wikipedia.org/wiki/Universally_unique_identifier

Some backends provide time stamps with low granularity (e.g., seconds but not milliseconds or nanoseconds). To compensate, the generation of version 1 UUIDs uses 14 random bits for the clock sequence.

The generation of version 4 UUIDs uses the `/dev/urandom` random number generator when available. This includes macOS, Linux, *BSD, and other POSIX operating-systems. On Windows, a pseudo-random generator is used, but randomized using the current wall time.

UUIDs can be generated as atoms, lists of characters, or lists of character codes.

See also the `ids` and `ulid` libraries.

6.57.1 API documentation

Open the ../../apis/library_index.html#uuid link in a web browser.

6.57.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(uuid(loader)).
```

6.57.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(uuid(tester)).
```

6.57.4 Generating version 1 UUIDs

By default, version 1 UUIDs are generated as atoms. For example:

```
| ?- uuid::uuid_v1([0xf2,0xd1,0x90,0x94,0xdc,0x4b], UUID).
UUID = '00a66fc0-82cf-11eb-bc83-f2d19094dc4b'
yes
```

To generate a UUID using a list of characters representation, use instead the `uuid/1` parametric object:

```
| ?- uuid(chars)::uuid_v1([0xf2,0xd1,0x90,0x94,0xdc,0x4b], UUID).
UUID = ['0','0',d,e,'9','0',c,'0',-,'8','2',c,f,-,'1','1',e,b,-,
        a,'9','8','5',-,f,'2',d,'1','9','0','9','4',d,c,'4',b]
yes
```

Similarly, to get a UUID using a list of character codes representation:

```
| ?- uuid(codes)::uuid_v1([0xf2,0xd1,0x90,0x94,0xdc,0x4b], UUID).
UUID = [48,48,52,99,99,54,99,48,45,56,50,99,102,45,49,49,101,98,45,
        98,57,102,52,45,102,50,100,49,57,48,57,52,100,99,52,98]
yes
```

6.57.5 Generating version 4 UUIDs

By default, version 4 UUIDs are generated as atoms. For example:

```
| ?- uuid::uuid_v4(UUID).
UUID = '1c652782-69c5-4252-88c8-09e576a44db5'
yes
```

To generate a UUID using a list of characters representation, use instead the `uuid/1` parametric object:

```
| ?- uuid(chars)::uuid_v4(UUID).
UUID = [d,'3',d,'3','3','5','1','3',-,'8','1',e,c,-,'4',d,'2','6',-,
        '9',f,'2','2',-,e,d,'9','5',e,'0','0',e,'1','5','7','0']
yes
```

Similar to get a UUID using a list of character codes representation:

```
| ?- uuid(codes)::uuid_v4(UUID).
UUID = [102,97,52,54,57,98,100,50,45,51,57,54,51,45,52,97,100,55,45,
        98,50,50,55,45,101,100,52,99,56,55,99,54,53,55,102,98]
yes
```

6.57.6 Generating the null UUID

A predicate is also provided that returns the null UUID:

```
| ?- uuid::uuid_null(UUID).
UUID = '00000000-0000-0000-0000-000000000000'
yes
```

6.58 `zippers`

This library implements zippers over lists.

6.58.1 API documentation

Open the ../../apis/library_index.html#zippers link in a web browser.

6.58.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(zippers(loader)).
```

6.58.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(zippers(tester)).
```

CHAPTER

SEVEN

PORTS

The documentation of each port can also be found in the port directory in the `NOTES.md` file.

7.1 fcube

This folder contains a Logtalk port of FCube: An Efficient Prover for Intuitionistic Propositional Logic available from:

```
https://www.vidal-rosset.net/fCube/
```

The port includes portability changes (notably, operator names) plus changes to use an ordered representation for sets. Also some code formatting changes for Logtalk coding guidelines.

The port tests are adapted from the examples available from the web page above.

To load this port and for sample queries, please see the `SCRIPT.txt` file.

For more information about FCube, see the following paper:

```
@InProceedings{10.1007/978-3-642-16242-8_21,
    author="Ferrari, Mauro and Fiorentini, Camillo and Fiorino, Guido",
    editor="Ferm{\"u}ller, Christian G. and Voronkov, Andrei",
    title="fCube: An Efficient Prover for Intuitionistic Propositional Logic",
    booktitle="Logic for Programming, Artificial Intelligence, and Reasoning",
    year="2010",
    publisher="Springer Berlin Heidelberg",
    address="Berlin, Heidelberg",
    pages="294--301",
    isbn="978-3-642-16242-8"
}
```

For sample queries, please see the `SCRIPT.txt` file.

7.1.1 API documentation

Open the ../../apis/library_index.html#fcube link in a web browser.

7.1.2 Loading

To load all entities in this port, load the `loader.lgt` file:

```
| ?- logtalk_load(fcube(loader)).
```

7.1.3 Testing

To test this port predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(fcube(tester)).
```

7.2 metagol

This folder contains a Logtalk port of `metagol`, an inductive logic programming (ILP) system based on meta-interpretive learning available from:

```
https://github.com/metagol/metagol
```

See the original code git repo for details and bibliography on Metagol and ILP.

The port allows any number of datasets to be loaded simultaneously with per-dataset learning options. A dataset is simply wrapped in an object that extends and is expanded by the `metagol` object as illustrated by the ported examples.

Both the original code and the port requires the coroutining `when/2` predicate, which is only available in some backend Prolog systems. The port currently supports ECLiPSe, XVM, SICStus Prolog, SWI-Prolog, and YAP. It can be used on both POSIX and Windows operating-systems.

The examples are ported from the original Metagol distribution. Some of the examples are taken from the following paper (with the original Prolog examples source code files made available by MystikNinja):

```
@article{DBLP:journals/jair/EvansG18,
    author    = {Richard Evans and Edward Grefenstette},
    title     = {Learning Explanatory Rules from Noisy Data},
    journal   = {J. Artif. Intell. Res.},
    volume    = {61},
    pages     = {1--64},
    year      = {2018},
    url       = {https://doi.org/10.1613/jair.5714},
    doi       = {10.1613/jair.5714},
    timestamp = {Mon, 21 Jan 2019 15:01:17 +0100},
    biburl    = {https://dblp.org/rec/bib/journals/jair/EvansG18},
    bibsource = {dblp computer science bibliography, https://dblp.org}
}
```

The paper can be downloaded at

```
https://arxiv.org/pdf/1711.04574.pdf
```

For sample queries, please see the `SCRIPT.txt` file.

7.2.1 API documentation

Open the ../../apis/library_index.html#metagol link in a web browser.

7.2.2 Loading

To load all entities in this port, load the `loader.lgt` file:

```
| ?- logtalk_load(metagol(loader)).
```

7.2.3 Testing

To test this port predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(metagol(tester)).
```

There are three lengthy tests that only run when the tests are being run manually instead of automatically.

7.3 toychr

This folder contains a Logtalk port of ToyCHR, a reference implementation of Constraint Handling Rules (CHR) available from:

```
https://www.comp.nus.edu.sg/~gregory/toychr/
```

The port is work in progress and includes significant modifications to the original code:

- Instead of compiling `.chr` files, it uses the term-expansion mechanism, by defining `toychrdb` as a hook object, to support writing rules inside objects and categories. As a consequence, the original `chr_compile/1` is not available.
- The port is portable and should run on all supported backends.

The port also includes examples ported from the SWI-Prolog CHR package examples and documentation. These examples are ported using the same license of the original code (BSD-2-Clause).

For sample queries, please see the `SCRIPT.txt` file.

7.3.1 API documentation

Open the ../../apis/library_index.html#toychr link in a web browser.

7.3.2 Loading

To load all entities in this port, load the `loader.lgt` file:

```
| ?- logtalk_load(toychr(loader)).
```

7.3.3 Testing

To test this port predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(toychr(tester)).
```

CHAPTER

EIGHT

CONTRIBUTIONS

The documentation of contribution can also be found in the contribution directory in the `NOTES.md` file.

8.1 flags

Flags was developed at Katholieke Universiteit Leuven

SPDX-FileCopyrightText: 2010 Katholieke Universiteit Leuven SPDX-License-Identifier: Artistic-2.0

Contributions to this file:
Author: Theofrastos Mantadelis
Suggestions: Paulo Moura
Version: 1
Date: 27/11/2010

8.1.1 API documentation

Open the ../../apis/library_index.html#flags link in a web browser.

8.1.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
?- logtalk_load(flags(loader)).
```

8.1.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(flags(tester)).
```

8.2 iso8601

ISO 8601 (and European civil calendar) compliant library of date and time (clock) related predicates. That is, an ISO 8601 handler.

Author: Daniel L. Dudley Created: 2004-02-18

Modified: 2014-09-26 (to use the library "os" object to get the current date)

8.2.1 API documentation

Open the ../../apis/library_index.html#iso8601 link in a web browser.

8.2.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(iso8601(loader)).
```

8.2.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(iso8601(tester)).
```

8.3 pddl_parser

PDDL 3.0 parser
Release 1.0

Copyright (c) 2011 Robert Sasak. All Rights Reserved. This is free software. You can redistribute it and/or modify it under the terms of the "Artistic License 2.0" as published by The Perl Foundation. Consult the "LICENSE.txt" file for details.

This PDDL 3.0 file parser converts PDDL files to Logtalk/Prolog friendly syntax. For example:

```
PDDL          Prolog
(on ?x ?y)    on(?x, ?y)
```

Syntax sugar: op(200, fy, ?).

For whole example check the "pddl.lgt" file for usage and example output.

8.3.1 API documentation

Open the ../../apis/library_index.html#pddl_parser link in a web browser.

8.3.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
?- logtalk_load(pddl_parser(loader)).
```

8.3.3 Testing

The provided unit tests are based on a collection of problem set files from International Planning Competition 2008. In order to run all unit tests from the parser's directory type:

```
?- logtalk_load(tester).
```

Or from any directory by typing:

```
?- logtalk_load(pddl_parser(tester)).
```

Some of the unit tests fail in some Prolog compilers due to limitations to the maximum arity of a term.

8.4 verdi_neruda

Verdi Neruda - Meta-interpreter collection for Prolog.
Release 1.0

Copyright (c) 2010 Victor Lagerkvist. All Rights Reserved. Verdi Neruda is free software. You can redistribute it and/or modify it under the terms of the simplified BSD license.

CONTENTS

1. License
2. About
3. Verdi Neruda web site
4. Installation and running
5. Examples
6. Authors
7. LICENSE

Copyright 2010 Victor Lagerkvist. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the copyright holders.

2. ABOUT

Verdi Neruda is a meta-interpreter collection for Prolog. Or, to be more precise, for a Prolog like language. Or, to be pedantically precise to the point that you are annoying people, it's not really a meta-interpreter collection at all since the interpreters themselves aren't interpreting the language that they are written in. Let's just say that it is a collection of interpreters for a logic programming language very much like pure Prolog with negation as finite failure.

Verdi Neruda is written entirely in Logtalk and compatible with most major Prolog systems. The name is sadly not a subtle wordplay or an acronym, but was generated by a computer with the help of a soundex algorithm. The purpose of the interpreter suite was to compare top-down methods to bottom-up methods and how resolution tree search rules affected performance and completeness. In the top-down family we find such interpreters as the long-time standing champion depth-first, its slow but orderly brother breadth-first and the youngster iterative deepening. A best-first framework can be found a stone throw away. With it it's possible to define interpreters that use greedy best-first search as well as A* search.

In the bottom-up camp we find an interpreter that uses a semi naive fixpoint construction. Since bottom-up interpreters by their very nature are not goal oriented a transformation technique called magic transformation is used on logic programs before any inferences are made. This technique allows the interpreter to only generate the facts that a top-down interpreter would have used on the same logic program.

A shell akin to a Prolog top loop is also included. It has commands both for proving goals with an interpreter of choice and for benchmarking logical inferences. If Verdi Neruda is run with a Prolog system that supports statistics/2 it's possible to obtain statistics such as CPU-time as well.

3. VERDI NERUDA WEB SITE

Visit the Verdi Neruda GitHub www-page at:

https://github.com/Joelbyte/verdi-neruda

4. INSTALLATION AND RUNNING

Verdi Neruda requires Logtalk 2.40.0 or a later version.

To use the snapshot of Verdi Neruda bundled with Logtalk:

- Start Logtalk.
- Type `{verdi_neruda(loader)}.` (Including `.`).

To use the latest version of Verdi Neruda, fetch the latest source code, either as an archive or from the git repository, extract it to a directory of your choice, and:

- Start Logtalk from that directory.

- Type `{loader}`. (Including `.`). If everything went according to the plan you should be greeted by the welcoming message. If you replace the bundled version with the new one, you can use in alternative the steps above.

5. EXAMPLES

Follow the previous instructions to get everything up and running. First we're going to run some predefined programs in the included databases. Begin by typing `databases.` from the shell - this should print a list of the currently loaded databases. The demo database `demodb` should be included in the list. Next type `listing(demodb).` to print the contents of the database. The output should look something like:

```
append([],A,A) if
    true.
append([A|B],C,[A|D]) if
    append(B,C,D).
.
.
.
```

Which means that the `append/3` program is loaded and ready for action. Next we need to decide which interpreter to use. Fortunately the shell does not leave much to the imagination - as might be expected, the `interpreters.` command prints the currently loaded interpreters. The list should look like:

```
dfs_interpreter
bfs_interpreter
iddfs_interpreter(A)
bup_interpreter
a_star_interpreter(A)
```

The variables means that the interpreters are parametric objects and that additional information is needed in order to run them. The iddfs-interpreter needs to know the increment and the A*-interpreter needs to know what weight should be used when calculating the cost of nodes. To start with let's use the dfs-interpreter and do something exciting, namely appending two lists!

```
prove(dfs_interpreter, append([a,b], [c,d], Xs), demodb).
```

The prove command takes three arguments. The first is a interpreter, the second the goal that shall be proved and the last the database that the clauses are derived from.

To accomplish the same thing with the iddfs-interpreter with an increment of 1 we need only type

```
prove(iddfs_interpreter(1), append([a,b], [c,d], Xs), demodb).
```

The shell also has support for counting logical inferences. To compare the dfs- and iddfs-interpreter with the append program we could write:

```
benchmark(dfs_interpreter, append([a,b,c,d],[e,f], Xs), demodb).  ->
dfs_interpreter inferences: 5

benchmark(iddfs_interpreter(1), append([a,b,c,d],[e,f], Xs), demodb).
-> iddfs_interpreter(1) inferences: 15
```

For more information regarding the built in shell commands consult the 'help.' command.

6. AUTHORS

The bulk of Verdi Neruda was written by Victor Lagerkvist during his bachelor thesis at Linköping university in the spring of 2010. Paulo Moura also helped a great deal during the later stages of development, especially with regards to compatibility between various Prolog systems.

8.5 xml_parser

This folder contains a Logtalk version of John Fletcher's Prolog XML parser:

https://binding-time.co.uk/index.php/Parsing_XML_with_Prolog

For a detailed description of this XML parser, please see the comments in the `xml.lgt` source file or convert the automatically generated documentation to HTML or PDF. For sample queries, please see the `SCRIPT.txt` file.

See the copyright and license information on the contributed files for usage and distributions conditions.

8.5.1 API documentation

Open the ../../apis/library_index.html#xml_parser link in a web browser.

8.5.2 Loading

To load all entities in this library, load the `loader.lgt` file:

```
| ?- logtalk_load(xml_parser(loader)).
```

8.5.3 Testing

To test this library predicates, load the `tester.lgt` file:

```
| ?- logtalk_load(xml_parser(tester)).
```

8.5.4 Known issues

When using GNU Prolog as the backend compiler, you may need to to use a larger default global stack size (see the GNU Prolog documentation on the environment variable `GLOBALSZ`).

CHAPTER

NINE

GLOSSARY

abstract class
A *class* that cannot be instantiated by sending it a message. Usually used to contain common predicates that are inherited by other classes.

abstract method
A *method* implementing an algorithm whose step corresponds to calls to methods defined in the descendants of the object (or *category*) containing it.

adapter file
A Prolog source file defining a minimal abstraction layer between the Logtalk compiler/runtime and a specific *backend Prolog compiler*.

after method
The *predicate definition* used to answer an `after` event.

ancestor
A *class* or a parent *prototype* that contributes (via inheritance) to the definition of an object. For class-based hierarchies, the ancestors of an instance are its class(es) and all the superclasses of its class(es). For prototype-based hierarchies, the ancestors of a prototype are its parent(s) and the ancestors of its parent(s).

around method
An overriding definition for a predicate that calls additional goals before and after a call to the overriden predicate definition.

backend Prolog compiler
The Prolog compiler that is used to host and run Logtalk and that is called for compiling the intermediate Prolog code generated by the Logtalk compiler when compiling source files.

before method
The *predicate definition* used to answer a `before` event.

built-in method
A predefined *method* that can be called from within any object or *category*. I.e. built-in methods are built-in object and category predicates. Built-in methods cannot be redefined.

built-in predicate
A predefined predicate that can be called from anywhere. Built-in predicates can be redefined within *objects* and *categories*.

category
A set of *predicates directives* and clauses that can be (virtually) imported by any object. Categories support composing objects using fine-grained units of code reuse and also *hot patching* of existing objects. A category should be functionally-cohesive, defining a single functionality.

class
An *object* that specializes another object, interpreted as its *superclass*. A class defines the common predicates of a set of objects that instantiate it. An object can also be interpreted as a class when it instantiates itself.

clause reference
An opaque term that uniquely identifies a clause. Provided by some backends via alternative database built-in predicates.

closed-world assumption
The assumption that what cannot be proved true is false. Therefore, sending a *message* corresponding to a *declared* but not *defined* predicate, or calling a declared predicate with no clauses, fails. But messages or calls to undeclared predicates generate an error.

closure
A callable term (i.e., an atom or a compound term) passed to a *meta-predicate* call where it is extended with additional arguments to form a goal called by the meta-predicate.

coinductive predicate
A predicate whose calls are proved using greatest fixed point semantics. Coinductive predicates allows reasoning about infinite rational entities such as cyclic terms and ω-automata.

complementing category
A category used for *hot patching* an existing object (or a set of objects).

component
A unique atom or compound term template identifying a library, tool, application, or application subsystem. Component names are notably used by the message printing and question asking mechanisms. Compound terms are used instead of atoms when parameterization is required.

directive
A source file term that affects the interpretation of source code. Directives use the `(:-)/1` prefix operator as functor.

discontiguous predicate
A predicate whose clauses are not contiguous in a *source file*. I.e. a predicate whose clauses are mixed with clauses for other predicates.

doclet file
A *source file* whose main purpose is to generate documentation for a *library* or an application.

doclet object
An object specifying the steps necessary to (re)generate the API documentation for a project. See the *doclet* and *lgtdoc* tools for details.

dynamic binding
Runtime lookup of a *predicate declaration* and *predicate definition* to verify the validity of a *message* (or a *super call*) and find the predicate definition that will be used to answer the message (or the super call). Also known as *late binding*. See also *static binding*.

dynamic entity
See *entity*.

dynamic predicate
A predicate whose clauses can be dynamically added or retracted at runtime.

early binding
See *static binding*.

encapsulation
The hiding of an object implementation. This promotes software reuse by isolating the object clients

from its implementation details. Encapsulation is enforced in Logtalk by using *predicate scope directives*.

entity
Generic name for Logtalk compilation units: *objects*, *categories*, and *protocols*. Entities share a single namespace (i.e., entity *identifiers* must be unique) and can be static (the default) or dynamic. Static entities are defined in source files. Dynamic entities can also be defined in source files but are usually created and abolished at runtime using the language built-in predicates.

entity alias
An alternative name for an entity. Entity aliases can be defined using the *uses/1* and *use_module/1* directives. Entity aliases can be used to improve code clarity by using alternative names that are more meaningful in the calling context, to bind parametric entity parameters, and to simplify experimenting with alternative entities implementing the same protocol.

entity directive
A directive that affects how Logtalk *entities* are used or compiled.

event
The sending of a *message* to an object. An event can be expressed as an ordered tuple: `(Event, Object, Message, Sender)`. Logtalk distinguishes between the sending of a message — `before` event — and the return of control to the sender — `after` event.

expansion workflow
A sequence of *term-expansion* or *goal-expansion* steps where each step is usually defined using a *hook object* or a combination of hook objects.

flaky test
A test that can unpredictably succeed, fail, or throw an error and thus can yield different results when repeated without any code changes. Can result from, e.g., race conditions or dependencies on external resources outside user control.

goal-expansion
The transformation of a goal into another goal, defined by clauses for the *goal_expansion/2* predicate that is declared by the expanding built-in protocol. Goal-expansion is usually recursively applied at compile time until a fixed-point is reached (i.e., until the result of an expansion cannot be further expanded). See also *term-expansion* and *expansion workflow*.

grammar rule
An alternative notation for predicates used to parse or generate sentences in some language. This notation hides the arguments used to pass the sequences of tokens being processed, thus simplifying the representation of grammars. Grammar rules are represented using as functor the infix operator `(-->)/2` instead of the `(:-)/2` operator used with predicate clauses.

grammar rule non-terminal
A syntactic category of words or phrases. A non-terminal is identified by its *non-terminal indicator*, i.e. by its name and number of arguments using the notation `Name//Arity`.

grammar rule terminal
A word or basic symbol of a language.

homoiconic
A property of programming languages where *code* and *data* use the same representation. Logtalk (and Prolog) are examples of homoiconic programming languages. A Logtalk program is a set of terms (clauses and directives) that can be handled as *data* by e.g. the *term-expansion* mechanism.

hook object
An object, implementing the expanding built-in protocol, defining *term-expansion* and *goal-expansion* predicates, used in the compilation of Logtalk or Prolog source files. A hook object can be specified using the *hook* flag. It can also be specified using a *set_logtalk_flag/2* directive in the source files to be expanded.

hook predicate
A predicate, usually declared *multifile*, that allows the user to customize another predicate or provide alternative definitions for a default predicate definition.

hot patching
The act of fixing *entity directives* and predicates or adding new entity directives and predicates to loaded *entities* in a running application without requiring access to the entities source code or restarting the application. Achieved using *complementing categories*.

identity
Property of an *entity* that distinguishes it from every other entity. The identifier of an entity is its functor (i.e., its name and arity), which must be unique. Object and *category* identifiers can be atoms or compound terms. Protocol identities must be atoms. All Logtalk entities (objects, protocols, and categories) share the same namespace.

inheritance
An *entity* inherits *predicates directives* and clauses from related entities. In the particular case of objects, when an object extends other object, we have prototype-based inheritance. When an object specializes or instantiates another object, we have class-based inheritance. See also *public inheritance*, *protected inheritance*, and *private inheritance*.

instance
An *object* that instantiates another object, interpreted as its *class*. An object may instantiate multiple objects (also known as multiple instantiation).

instantiation
The process of creating a new *class instance*. In Logtalk, this does not necessarily imply dynamic creation of an object at runtime; an instance may also be defined as a static object in a source file.

interface
See *protocol*.

lambda expression
A compound term that can be used in place of a goal or *closure* meta-argument and that abstracts a *predicate definition* by listing its variables and a callable term that implements the definition. Lambda expressions help avoid the need of naming and defining auxiliary predicates.

lambda free variable
A variable that is global to a *lambda expression*. All used global variables must be explicitly listed in a lambda expression for well-defined semantics.

lambda parameter
A term (usually a variable or a non-ground compound term) that is local to a *lambda expression*. All lambda parameters must be explicitly enumerated in a lambda expression for well-defined semantics.

late binding
See *dynamic binding*.

library
A directory containing source files. See also *library alias* and *library notation*.

library alias
An atom that can be used as an alias for a *library* full path. Library aliases and their corresponding paths can be defined using the *logtalk_library_path/2* predicate. See also *library notation*.

library notation
A compound term where the name is a *library alias* and the single argument is a *source file* relative path. Use of library notation simplifies compiling and loading source files and can make an application easily relocatable by defining an alias for the root directory of the application files.

loader file
A *source file* whose main purpose is to load a set of source files (possibly with specific compiler flags) and any library dependencies.

local predicate
A predicate that is defined in an object (or in a *category*) but that is not listed in a *scope directive*. These predicates behave like private predicates but are invisible to the reflection *built-in methods*. Local predicates are usually auxiliary predicates and only relevant to the entity where they are defined.

message
A query sent to an object. In logical terms, a message can be interpreted as a request for proof construction using an object database and the databases of related entities.

message lookup
Sending a message to an object requires a lookup for the *predicate declaration*, to check if the message is within the scope of the sender, and a lookup for the *predicate definition* that is going to be called to answer the message. Message lookup can occur at *compile* time or at *runtime*.

message to self
A message sent to the object that received the original message under processing. Messages to self require *dynamic binding* as the value of self is only know at runtime.

meta-argument
A predicate argument that is called as a goal, used as a *closure* to construct a goal that will be called, or that is handled in a way that requires awareness of the predicate calling context.

meta-interpreter
A program capable of running other programs written in the same language.

meta-predicate
A predicate with one or more *meta-arguments*. For example, *call/1-N* and *findall/3* are built-in meta-predicates.

meta-variable
A variable in a *meta-argument* position that is expected to be unified with a goal or a closure at runtime.

metaclass
The *class* of a class, when interpreted as an *instance*. Metaclass instances are themselves classes. Metaclasses are optional, except for the root class, and can be shared by several classes.

method
The *predicate definition* used to answer a *message* sent to an object. Logtalk supports both *static binding* and *dynamic binding* to find which method to run to answer a message. See also *built-in method*.

mocking
Techniques used to replace dependencies in the code being tested with controlled substitutes, simplifying testing. Can be accomplished using, e.g., *term-expansion*, hot patching, or message interception.

module
A Prolog entity characterized by an identity and a set of *predicates directives* and clauses. Prolog modules are usually static although some Prolog systems allow the creation of dynamic modules at runtime. Prolog modules can be interpreted as *prototypes*.

monitor
Any object, implementing the monitoring built-in protocol, that is notified by the runtime when a spied event occurs. The spied *events* can be set by the monitor itself or by any other object.

multifile predicate
A predicate whose clauses can be defined in multiple *entities* and *source files*. The object or category holding the directive without an entity prefix qualifying the predicate holds the multifile predicate *primary declaration*, which consists of both a *scope directive* and a *multifile/1* directive for the predicate.

naked meta-variable
A *meta-variable* used as the body of a predicate clause or grammar rule or used in a cut-transparent argument of a control construct. The "naked" designation highlights that the meta-variable is not wrapped by `call/1` or `phrase//1` goals.

object
An entity characterized by an *identity* and a set of *predicates directives* and clauses. Logtalk objects can be either static or dynamic. Logtalk objects can play the *role* of classes, instances, or prototypes. The role or roles an object plays are a function of its relations with other objects.

object database
The set of predicates locally defined inside an object.

parameter
An argument of a parametric object or a parametric category identifier. Parameters are *logical variables* implicitly shared by all the entity directives and predicate clauses.

parameter variable
A variable used as parameter in a parametric object or a parametric category using the syntax `_ParameterName_` (i.e., a variable whose name stars and ends with an underscore). Parameter variables are *logical variables* shared by all entity terms. Occurrences of parameter variables in *entity directives* and clauses are implicitly unified with the corresponding entity parameters.

parametric category
See *parametric entity*.

parametric entity
An *object* or *category* whose *identifier* is a compound term possibly containing free variables that can be used to parameterize the entity predicates. Parameters are *logical variables* implicitly shared by all the entity clauses. Note that the identifier of a parametric entity is its functor, irrespective of the possible values of its arguments (e.g., `foo(bar)` and `foo(baz)` are different parameterizations of the same parametric entity, `foo/1`).

parametric object
See *parametric entity*.

parametric object proxy
A compound term (usually represented as a plain Prolog fact) with the same name and number of arguments as the identifier of a parametric object.

parent
A *prototype* that is extended by another prototype.

polymorphism
Different objects (and categories) can provide different implementations of the same predicate. The predicate declaration can be inherited from a common ancestor, also known as *subtype polymorphism*. Logtalk implements *single dispatch* on the receiver of a message, which can be described as *single-argument polymorphism*. As *message lookup* only uses the predicate functor, multiple predicate implementations for different types of arguments are possible, also known as *ad hoc polymorphism*. *Parametric objects and categories* enable implementation of *parametric polymorphism* by using one or more parameters to pass object identifiers that can be used to parameterize generic predicate definitions.

predicate
Predicates describe what is true about the application domain. A predicate is identified by its *predicate indicator*, i.e. by its name and number of arguments using the notation `Name/Arity`. See also *built-in predicate* and *method*.

predicate alias
An alternative functor (`Name/Arity`) for a predicate. Predicate aliases can be defined for any inherited predicate using the *alias/2* directive and for predicates listed in *uses/2* and *use_module/2* directives.

Predicate aliases can be used to solve inheritance conflicts, to improve code clarity by using alternative names that are more meaningful in the calling context, and to use a different order of the predicate arguments.

predicate calling context
The object or category from within a predicate is called (either directly or using a control construct such as a message-sending control construct).

predicate declaration
A predicate declaration is composed by a set of *predicates directives*, which must include at least a *scope directive*.

predicate definition
The set of clauses for a predicate, contained in an object or category. Predicate definitions can be overridden or specialized in descendant entities.

predicate definition context
The object or category that contains the definition (i.e., clauses) for a predicate.

predicate directive
A directive that specifies a predicate property that affects how predicates are called or compiled.

predicate execution context
The implicit arguments (including *sender*, *self*, and *this*) required for the correct execution of a predicate call.

predicate scope container
The object that inherits a *predicate declaration* from an imported *category* or an implemented *protocol*.

predicate scope directive
A directive that declares a predicate by specifying its visibility as *public*, *protected*, or *private*.

predicate shorthand
A *predicate alias* that defines a call template, possibly using a different name, with a reduced number of arguments by hard-coding the value of the omitted arguments in the original call template. Predicate shorthands can be defined using *uses/2* and *use_module/2* directives. They can be used to simplify predicate calls and to ensure consistent call patterns when some of the arguments always use the same fixed values in the calling context.

primary predicate declaration
See *multifile predicate*.

private inheritance
All public and protected predicates are inherited as private predicates. See also *public inheritance* and *protected inheritance*.

private predicate
A predicate that can only be called from the object that contains its *scope directive*.

profiler
A program that collects data about other program performance.

protected inheritance
All public predicates are inherited as protected. No scope change for protected or private predicates. See also *public inheritance* and *private inheritance*.

protected predicate
A predicate that can only be called from the object containing its *scope directive* or from an object that inherits the predicate.

protocol
An entity that contains *predicate declarations*. A predicate is declared using a *scope directive*. It may be

further specified by additional predicate directives. Protocols support the separation between interface and implementation, can be implemented by both objects and categories, and can be extended by other protocols. A protocol should be functionally-cohesive, specifying a single functionality. Also known as *interface*.

prototype
A self-describing *object* that may extend or be extended by other objects. A prototype typically describes a concrete object instead of an abstraction of a set of objects. An object with no instantiation or specialization relations with other objects is always interpreted as a prototype.

public inheritance
All inherited predicates maintain their declared scope. See also *protected inheritance* and *private inheritance*.

public predicate
A predicate that can be called from any object.

scratch directory
The directory used to save the intermediate Prolog files generated by the compiler when compiling *source files*.

self
The object that received the *message* under processing.

sender
An object that sends a *message* to other object. When a message is sent from within a *category*, the *sender* is the object importing the category on which behalf the message was sent.

settings file
A *source file*, compiled and loaded automatically by default at Logtalk startup, mainly defining default values for compiler flags that override the defaults found on the backend Prolog compiler *adapter files*.

singleton method
A *method* defined in an *instance* itself. Singleton methods are supported in Logtalk and can also be found in other object-oriented programming languages.

source file
A text file defining Logtalk and/or Prolog code. Multiple Logtalk entities may be defined in a single source file. Plain Prolog code may be intermixed with Logtalk entity definitions. Depending on the used *backend Prolog compiler*, the text encoding may be specified using an *encoding/1* directive as the first term in the first line in the file.

source file directive
A *directive* that affects how a *source file* is compiled.

specialization
A *class* is specialized by defining a new class that inherits its predicates and possibly adds new ones.

static binding
Compile time lookup of a *predicate declaration* and *predicate definition* when compiling a *message* sending call (or a *super call*). Dynamic binding is used whenever static binding is not possible (e.g., due to the predicate being dynamic or due to lack of enough information at compilation time). Also known as *early binding*. See also *dynamic binding*.

static entity
See *entity*.

steadfastness
A predicate definition is *steadfast* when it still generates only correct answers when called with unexpectedly bound arguments (notably, bound output arguments). Typically, a predicate may not be steadfast when output argument unifications can occur before a cut in a predicate clause body.

subclass
A *class* that is a specialization, direct or indirectly, of another class. A class may have multiple subclasses.

super call
Call of an inherited (or imported) *predicate definition*. Mainly used when redefining an inherited (or imported) predicate to call the overridden definition while making additional calls. Super calls preserve *self* and may require *dynamic binding* if the predicate is dynamic.

superclass
A *class* from which another class is a specialization (directly or indirectly via another class). A class may have multiple superclasses.

synchronized predicate
A synchronized predicate is protected by a mutex ensuring that, in a multi-threaded application, it can only be called by a single thread at a time.

template method
See *abstract method*.

term-expansion
The transformation of a term (usually a directive or a clause) into another term or a list of terms, defined by clauses for the *term_expansion/2* predicate that is declared by the expanding built-in protocol. Term-expansion is usually applied at compile time once (i.e., the result of an expansion is not further expanded). See also *goal-expansion* and *expansion workflow*.

test dialect
Predicate syntax for defining a test. I.e., the predicate name; the number, syntax, and semantics of the predicate arguments; and the syntax and semantics of any options.

test outcome
The expected result of a test goal. E.g., success, failure, an error, deterministic success, success with bindings that subsume or are a variant of specific terms.

tester file
A *source file* whose main purpose is to load and a run a set of unit tests.

this
The object that contains the predicate clause under execution. When the predicate clause is contained in a *category*, *this* is a reference to the object importing the category on which behalf the predicate clause is being used to prove the current goal.

threaded engine
A computing thread running a goal whose solutions can be lazily and concurrently computed and retrieved. A threaded engine also supports a term queue that allows passing arbitrary terms to the engine. This queue can be used to pass e.g. data and new goals to the engine.

top-level interpreter shorthand
Aliases for frequently used built-in predicates such as *logtalk_load/1* and *logtalk_make/1*. These shorthands are **not** part of the Logtalk language and must only be used at the top-level interpreter.

visible predicate
A predicate that is within scope, a locally defined predicate, a *built-in method*, a Logtalk built-in predicate, or a Prolog built-in predicate.

BIBLIOGRAPHY

[Alexiev93] Mutable Object State for Object-Oriented Logic Programming: A Survey Alexiev, V. Technical Report TR 93-15, Department of Computing Science, University of Alberta, Canada

[Belli_et_al_92] Object-oriented programming in Prolog: rationale and a case study Belli, F., Jack, O., Naish, L. Technical Report 92/2, Department of Electrical and Electronics Engineering, University of Paderborn, Germany URL: https://lee-naish.github.io/papers/oolp/index.html

[Block89] An Extended Frame Language Block, F. P., Chan, N. C. Proceedings OOPLSLA 89(10):151-157, ACM

[Bobrow_et_al_88] Common Lisp Object System Specification Bobrow, D. G., Michiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., Moon, D. A. ACM SIGPLAN Notices(23)

[Bratko90] Prolog Programming for Artificial Intelligence Bratko, I. Addison Wesley, 2º edition, 1990

[Champaux92] A comparative Study of Object-Oriented Analysis Methods Champaux, D., Faure, P. Journal of Object-Oriented Programming, Vol. 5, N.1, 1992

[Clocksin87] Programming in Prolog Clocksin, W.F., Mellish, C.S. Springer-Verlag, New York, 1987

[Cointe87] Metaclasses are First Class: the ObjVlisp Model Cointe, P. Proceedings OOPLSLA 87(10):156-167, ACM

[Cordes91] The Literate Programming Paradigm Cordes, D., Brown, M. IEEE Computer, June 1991:52-61

[Covington94] ISO Prolog: A Summary of the Draft Proposed Standard Covington, M. A. URL: ftp://ai.uga.edu/pub/prolog.standard/

[Cox86] Object-Oriented Programming: An Evolutionary Approach Cox, Brad J. Addison-Wesley Publishing Company, Don Mills, Ontario

[Davison89] Polka: A Parlog Object oriented language Davison, A. Ph.D. Thesis, Imperial College, London, 1989

[Davison92] A survey of logic programming-based object oriented languages Davison, A. Tech Report 92/3, Dept. of Computer Science, University of Melbourne, Australia URL: https://catalogue.nla.gov.au/catalog/772526

[Davison93] The deductive and object oriented features of BeBOP Davison, A. Tech Report 93/6, Dept. of Computer Science, University of Melbourne, Australia URL: https://catalogue.nla.gov.au/catalog/1273317

[Delzanno97] Logic and Object-Oriented Programming in Linear Logic Delzanno, G. Ph.D. Thesis, University of Pisa, Italy URL: https://opac.bncf.firenze.sbn.it/Record/BVE0136144?uri=BVE0136144

[Dony90] Exception Handling and Object-Oriented Programming: Towards a Synthesis Dony, C. Proceedings OOPLSLA 90:322-330, ACM

[Fornarino_et_al_89] An Original Object-Oriented Approach for Relation Management Fornarino, M., Pinna, A.-M.,Trousse, B. Proceedings of the 4th Portuguese Conference on Artificial Intelligence Lecture Notes in Artificial Intelligence, Springer-Verlag (390):13-26

[Fromherz93] OL(P): Object Layer for Prolog Fromherz, M. URL: ftp://parcftp.xerox.com/ftp/pub/ol/

[Fukunaga86] An Experience with a Prolog-based Object-Oriented Language Fukunaga, K., Hirose, S. Proceedings OOPLSLA 86, 21(11):224-231, ACM

[Goldberg83] Smalltalk-80 The language and its implementation Goldberg, A., Robson, D. Addison-Wesley Series in Computer Science

[Joy_et_al_00] The Java Language Specification, Second Edition Joy, B., Steele, G., Gosling, J., Bracha, G. Addison-Wesley, 2000

[ISO95] ISO/IEC DIS 13211-1 - Programming Language Prolog Part 1: General Core Joint Technical Committee ISO/IEC JTC 1 URL: https://www.iso.org/standard/21413.html

[Knuth84] Literate Programming Knuth, D. E. Computer Journal, May 84, 27(2):97-111

[Lieberman86] Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems Lieberman, H. Proceedings OOPLSLA 86:189-214, ACM

[Maes87] Concepts and Experiments in Computational Reflection Maes, P. Proceedings OOPLSLA 87, ACM

[McCabe92] Logic and Objects McCabe, F. G. Prentice Hall Series in Computer Science

[Moon86] Object-Oriented Programming in Flavors Moon, D. Proceedings OOPLSLA 86:1-8, ACM

[Moss94] Prolog++ The Power of Object-Oriented and Logic Programming Moss, C. Addison-Wesley International Series in Logic Programming, 1994

[Moura94] Logtalk: Programação Orientada para Objectos em Prolog Moura, P., Costa, E. 2ª Conferência e Exposição Portuguesa de Tecnologia Orientada por Objectos 3i Consultores, Lisboa

[Moura99] Porting Prolog: Notes on porting a Prolog program to 22 Prolog compilers or the relevance of the ISO Prolog standard Moura, P. ALP Newsletter, Vol. 12/2, May 1999

[Moura00] Logtalk 2.6 Documentation Moura, P. Technical Report DMI 2000/1 University of Beira Interior, Portugal

[Razek92] Combining Objects and Relations Razek, G. Communications of the ACM, 27(12):66-70

[Rumbaugh87] Relations as Semantic Constructs in an Object-Oriented Language Rumbaugh, J. Proceedings OOPLSLA 87:466-481, ACM

[Rumbaugh88] Controlling Propagation of Operations using Attributes on Relations Rumbaugh, J. Proceedings OOPLSLA 88:285-296, ACM

[Schachte95] Efficient Object-Oriented Programming in Prolog Schachte, P., Saab, G. Logic Programming: Formal Methods and Pratical Applications Studies in Computer Science and Artificial Intelligence, 11 Elsevier Science B.V. North-Holland, Amsterdam, 1995

[SICStus95] SICStus Prolog Manual SICStus URL: https://sicstus.sics.se

[Shan_et_al_93] Is Multiple Inheritance Essential to OOP? (Panel) Shan, Y., Cargill, T., Cox, B., Cook, W., Loomis, M., Snyder, A. Proceedings OOPLSLA 93:360-363

[Stefik_et_al_86] Integrating Acess-Oriented Programming into a Multiparadigm Environment Stefik, M. J., Bobrow, D. G. , Kahn, K. M. IEEE Software, January 1986:10-18

[Stroustrup86] The C++ Programming Language Stroustrup, B. Addison-Wesley Series in Computer Science

[Taenzer89] Problems in Object-Oriented Software Reuse Taenzer, D., Ganti, M., Podar, S. Proceedings of ECOOP 89 British Computer Society Workshop Series, Cambridge University Press

[Tanzer95] Remarks on Object-Oriented Modeling of Associations Tanzer, C. Journal of Object-Oriented Programming, February 1995, SIGS Publications

[Tanenbaum87] Operating Systems - Design and Implementation Tanenbaum, A. Prentice-Hall Software Series, 1987

[Welsch89] Reasoning Objects with Dynamic Knowledge Bases Welsch, C., Barth, G. Proceedings of the 4th Portuguese Conference on Artificial Intelligence(390):257-268 Lecture Notes in Artificial Intelligence, Springer-Verlag, 1989

INDEX

Symbols

!/0
    Built-in method, 310
(::)/1
    Control construct, 196
(::)/2
    Control construct, 194
(@)/1
    Control construct, 199
(^^)/1
    Control construct, 198
(\+)/1
    Built-in method, 334
(<<)/2
    Control construct, 202
{}/1
    Control construct, 200
[]/1
    Control construct, 197

A

abolish/1
    Built-in method, 322
abolish_category/1
    Built-in predicate, 255
abolish_events/5
    Built-in predicate, 268
abolish_object/1
    Built-in predicate, 256
abolish_protocol/1
    Built-in predicate, 257
abstract class, **593**
abstract method, **593**
adapter file, **593**
after method, **593**
after/3
    Built-in method, 359
alias/2
    Directive, 227
always_true_or_false_goals
    Flag, 114
ancestor, **593**
arithmetic_expressions
    Flag, 114
around method, **593**
ask_question/5
    Built-in method, 379
asserta/1
    Built-in method, 323
assertz/1
    Built-in method, 326

B

backend Prolog compiler, **593**
bagof/3
    Built-in method, 353
before method, **593**
before/3
    Built-in method, 359
begin_of_file, 131
behavioral reflection, 103
black-box view, 103
built_in/0
    Directive, 214
built-in method, **593**
built-in predicate, **593**
Built-in method
    !/0, 310
    (\+)/1, 334
    abolish/1, 322
    after/3, 359
    ask_question/5, 379
    asserta/1, 323
    assertz/1, 326
    bagof/3, 353
    before/3, 359
    call//1-N, 361
    call/1-N, 331
    catch/3, 336
    clause/2, 328
    coinductive_success_hook/1-2, 371
    consistency_error/3, 343
    context/1, 313
    current_op/3, 318

current_predicate/1, 319
domain_error/2, 341
eos//0, 363
evaluation_error/1, 349
existence_error/2, 344
expand_goal/2, 369
expand_term/2, 367
fail/0, 311
false/0, 311
findall/3, 354
findall/4, 355
forall/2, 356
forward/1, 360
goal_expansion/2, 370
ignore/1, 332
instantiation_error/0, 338
message_hook/4, 374
message_prefix_file/6, 376
message_prefix_stream/4, 375
message_tokens//2, 373
once/1, 333
parameter/2, 314
permission_error/3, 346
phrase//1, 364
phrase/2, 365
phrase/3, 366
predicate_property/2, 321
print_message/3, 373
print_message_token/4, 378
print_message_tokens/3, 377
question_hook/6, 379
question_prompt_stream/4, 380
repeat/0, 312
representation_error/1, 348
resource_error/1, 350
retract/1, 329
retractall/1, 330
self/1, 316
sender/1, 316
setof/3, 357
syntax_error/1, 351
system_error/0, 352
term_expansion/2, 368
this/1, 317
throw/1, 337
true/0, 310
type_error/2, 340
ununinstantiation_error/1, 339
Built-in predicate
abolish_category/1, 255
abolish_events/5, 268
abolish_object/1, 256
abolish_protocol/1, 257
category_property/2, 248
complements_object/2, 264
conforms_to_protocol/2-3, 262
create_category/4, 251
create_logtalk_flag/3, 307
create_object/4, 252
create_protocol/3, 254
current_category/1, 245
current_event/5, 268
current_logtalk_flag/2, 306
current_object/1, 246
current_protocol/1, 247
define_events/5, 269
extends_category/2-3, 260
extends_object/2-3, 258
extends_protocol/2-3, 259
implements_protocol/2-3, 261
imports_category/2-3, 264
instantiates_class/2-3, 265
logtalk_compile/1, 291
logtalk_compile/2, 292
logtalk_library_path/2, 301
logtalk_linter_hook/7, 309
logtalk_load/1, 294
logtalk_load/2, 296
logtalk_load_context/2, 303
logtalk_make/0, 297
logtalk_make/1, 298
logtalk_make_target_action/1, 300
object_property/2, 248
protocol_property/2, 249
set_logtalk_flag/2, 306
specializes_class/2-3, 266
threaded/1, 271
threaded_call/1-2, 272
threaded_cancel/1, 279
threaded_engine/1, 284
threaded_engine_create/3, 282
threaded_engine_destroy/1, 283
threaded_engine_fetch/1, 290
threaded_engine_next/2, 286
threaded_engine_next_reified/2, 287
threaded_engine_post/2, 289
threaded_engine_self/1, 285
threaded_engine_yield/1, 288
threaded_exit/1-2, 276
threaded_ignore/1, 275
threaded_notify/1, 281
threaded_once/1-2, 273
threaded_peek/1-2, 277
threaded_wait/1, 280

C

call//1-N
Built-in method, 361

call/1-N
    Built-in method, 331
catch/3
    Built-in method, 336
catchall_catch
    Flag, 115
category, **593**
category/1-4
    Directive, 214
category_property/2
    Built-in predicate, 248
class, **594**
clause reference, **594**
clause/2
    Built-in method, 328
clean
    Flag, 117
closed-world assumption, **594**
closure, **594**
code_prefix
    Flag, 116
coinduction
    Flag, 112
coinductive predicate, **594**
coinductive/1
    Directive, 228
coinductive_success_hook/1-2
    Built-in method, 371
complementing category, **594**
complements
    Flag, 115
complements_object/2
    Built-in predicate, 264
component, **594**
conditionals
    Flag, 114
conforms_to_protocol/2-3
    Built-in predicate, 262
consistency_error/3
    Built-in method, 343
context/1
    Built-in method, 313
context_switching_calls
    Flag, 115
Control construct
    (::)/1, 196
    (::)/2, 194
    (@)/1, 199
    (^^)/1, 198
    (<<)/2, 202
    {}/1, 200
    []/1, 197
create_category/4
    Built-in predicate, 251
create_logtalk_flag/3
    Built-in predicate, 307
create_object/4
    Built-in predicate, 252
create_protocol/3
    Built-in predicate, 254
current_category/1
    Built-in predicate, 245
current_event/5
    Built-in predicate, 268
current_logtalk_flag/2
    Built-in predicate, 306
current_object/1
    Built-in predicate, 246
current_op/3
    Built-in method, 318
current_predicate/1
    Built-in method, 319
current_protocol/1
    Built-in predicate, 247

D

debug
    Flag, 117
define_events/5
    Built-in predicate, 269
deprecated
    Flag, 113
Directive
    alias/2, 227
    built_in/0, 214
    category/1-4, 214
    coinductive/1, 228
    discontiguous/1, 229
    dynamic/0, 216
    dynamic/1, 230
    elif/1, 211
    else/0, 212
    encoding/1, 204
    end_category/0, 217
    end_object/0, 217
    end_protocol/0, 218
    endif/0, 213
    if/1, 210
    include/1, 205
    info/1, 218
    info/2, 231
    initialization/1, 206
    meta_non_terminal/1, 233
    meta_predicate/1, 232
    mode/2, 234
    multifile/1, 235
    object/1-5, 219
    op/3, 207

    private/1, 236
    protected/1, 237
    protocol/1-2, 223
    public/1, 238
    set_logtalk_flag/2, 209
    synchronized/1, 239
    threaded/0, 224
    use_module/1, 226
    use_module/2, 243
    uses/1, 225
    uses/2, 240
directive, **594**
discontiguous predicate, **594**
discontiguous/1
    Directive, 229
disjunctions
    Flag, 114
doclet file, **594**
doclet object, **594**
domain_error/2
    Built-in method, 341
duplicated_clauses
    Flag, 114
duplicated_directives
    Flag, 113
dynamic binding, **594**
dynamic entity, **594**
dynamic predicate, **594**
dynamic/0
    Directive, 216
dynamic/1
    Directive, 230
dynamic_declarations
    Flag, 115

E

early binding, **594**
elif/1
    Directive, 211
else/0
    Directive, 212
encapsulation, **594**
encoding/1
    Directive, 204
encoding_directive
    Flag, 112
encodings
    Flag, 115
end_category/0
    Directive, 217
end_object/0
    Directive, 217
end_of_file, 131
end_protocol/0
    Directive, 218
endif/0
    Directive, 213
engines
    Flag, 112
entity, **595**
entity alias, **595**
entity directive, **595**
eos//0
    Built-in method, 363
evaluation_error/1
    Built-in method, 349
event, **595**
events
    Flag, 115
existence_error/2
    Built-in method, 344
expand_goal/2
    Built-in method, 369
expand_term/2
    Built-in method, 367
expansion workflow, **595**
extends_category/2-3
    Built-in predicate, 260
extends_object/2-3
    Built-in predicate, 258
extends_protocol/2-3
    Built-in predicate, 259

F

fail/0
    Built-in method, 311
false/0
    Built-in method, 311
findall/3
    Built-in method, 354
findall/4
    Built-in method, 355
Flag
    always_true_or_false_goals, 114
    arithmetic_expressions, 114
    catchall_catch, 115
    clean, 117
    code_prefix, 116
    coinduction, 112
    complements, 115
    conditionals, 114
    context_switching_calls, 115
    debug, 117
    deprecated, 113
    disjunctions, 114
    duplicated_clauses, 114
    duplicated_directives, 113
    dynamic_declarations, 115

encoding_directive, 112
encodings, 115
engines, 112
events, 115
general, 115
grammar_rules, 114
hook, 117
lambda_variables, 114
left_recursion, 115
linter, 113
missing_directives, 113
modules, 112
naming, 114
optimize, 116
portability, 113
prolog_compatible_version, 112
prolog_compiler, 116
prolog_dialect, 112
prolog_loader, 116
prolog_version, 112
redefined_built_ins, 114
redefined_operators, 114
relative_to, 117
reload, 117
report, 116
scratch_directory, 116
settings_file, 112
singleton_variables, 114
source_data, 117
steadfastness, 113
suspicious_calls, 114
tabling, 112
tail_recursive, 115
threads, 112
trivial_goal_fails, 114
undefined_predicates, 113
underscore_variables, 116
unicode, 112
unknown_entities, 113
unknown_predicates, 113
version_data, 113
flaky test, **595**
forall/2
Built-in method, 356
forward/1
Built-in method, 360

G

general
Flag, 115
goal_expansion/2
Built-in method, 370
goal-expansion, **595**
grammar rule, **595**
grammar rule non-terminal, **595**
grammar rule terminal, **595**
grammar_rules
Flag, 114

H

homoiconic, **595**
hook
Flag, 117
hook object, **595**
hook predicate, **596**
hot patching, **596**

I

identity, **596**
if/1
Directive, 210
ignore/1
Built-in method, 332
implements_protocol/2-3
Built-in predicate, 261
imports_category/2-3
Built-in predicate, 264
include/1
Directive, 205
info/1
Directive, 218
info/2
Directive, 231
inheritance, **596**
initialization/1
Directive, 206
instance, **596**
instantiates_class/2-3
Built-in predicate, 265
instantiation, **596**
instantiation_error/0
Built-in method, 338
interface, **596**

L

lambda expression, **596**
lambda free variable, **596**
lambda parameter, **596**
lambda_variables
Flag, 114
late binding, **596**
left_recursion
Flag, 115
library, **596**
library alias, **596**
library notation, **596**
linter
Flag, 113

loader file, 597
local predicate, 597
logtalk_compile/1
    Built-in predicate, 291
logtalk_compile/2
    Built-in predicate, 292
logtalk_library_path/2
    Built-in predicate, 301
logtalk_linter_hook/7
    Built-in predicate, 309
logtalk_load/1
    Built-in predicate, 294
logtalk_load/2
    Built-in predicate, 296
logtalk_load_context/2
    Built-in predicate, 303
logtalk_make/0
    Built-in predicate, 297
logtalk_make/1
    Built-in predicate, 298
logtalk_make_target_action/1
    Built-in predicate, 300

M

message, 597
message lookup, 597
message to self, 597
message_hook/4
    Built-in method, 374
message_prefix_file/6
    Built-in method, 376
message_prefix_stream/4
    Built-in method, 375
message_tokens//2
    Built-in method, 373
meta_non_terminal/1
    Directive, 233
meta_predicate/1
    Directive, 232
meta-argument, 597
meta-interpreter, 597
meta-predicate, 597
meta-variable, 597
metaclass, 597
method, 597
missing_directives
    Flag, 113
mocking, 597
mode/2
    Directive, 234
module, 597
modules
    Flag, 112
monitor, 597
multifile predicate, 597
multifile/1
    Directive, 235

N

naked meta-variable, 598
naming
    Flag, 114

O

object, 598
object database, 598
object/1-5
    Directive, 219
object_property/2
    Built-in predicate, 248
once/1
    Built-in method, 333
op/3
    Directive, 207
optimize
    Flag, 116

P

parameter, 598
parameter variable, 598
parameter/2
    Built-in method, 314
parametric category, 598
parametric entity, 598
parametric object, 598
parametric object proxy, 598
parent, 598
permission_error/3
    Built-in method, 346
phrase//1
    Built-in method, 364
phrase/2
    Built-in method, 365
phrase/3
    Built-in method, 366
polymorphism, 598
portability
    Flag, 113
predicate, 598
predicate alias, 598
predicate calling context, 599
predicate declaration, 599
predicate definition, 599
predicate definition context, 599
predicate directive, 599
predicate execution context, 599
predicate scope container, 599
predicate scope directive, 599

predicate shorthand, **599**
predicate_property/2
    Built-in method, 321
primary predicate declaration, **599**
print_message/3
    Built-in method, 373
print_message_token/4
    Built-in method, 378
print_message_tokens/3
    Built-in method, 377
private inheritance, **599**
private predicate, **599**
private/1
    Directive, 236
profiler, **599**
prolog_compatible_version
    Flag, 112
prolog_compiler
    Flag, 116
prolog_dialect
    Flag, 112
prolog_loader
    Flag, 116
prolog_version
    Flag, 112
protected inheritance, **599**
protected predicate, **599**
protected/1
    Directive, 237
protocol, **599**
protocol/1-2
    Directive, 223
protocol_property/2
    Built-in predicate, 249
prototype, **600**
public inheritance, **600**
public predicate, **600**
public/1
    Directive, 238

Q

question_hook/6
    Built-in method, 379
question_prompt_stream/4
    Built-in method, 380

R

redefined_built_ins
    Flag, 114
redefined_operators
    Flag, 114
reflection, 102
relative_to
    Flag, 117
reload
    Flag, 117
repeat/0
    Built-in method, 312
report
    Flag, 116
representation_error/1
    Built-in method, 348
resource_error/1
    Built-in method, 350
retract/1
    Built-in method, 329
retractall/1
    Built-in method, 330

S

scratch directory, **600**
scratch_directory
    Flag, 116
self, **600**
self/1
    Built-in method, 316
sender, **600**
sender/1
    Built-in method, 316
set_logtalk_flag/2
    Built-in predicate, 306
    Directive, 209
setof/3
    Built-in method, 357
settings file, **600**
settings_file
    Flag, 112
singleton method, **600**
singleton_variables
    Flag, 114
source file, **600**
source file directive, **600**
source_data
    Flag, 117
specialization, **600**
specializes_class/2-3
    Built-in predicate, 266
static binding, **600**
static entity, **600**
steadfastness, **600**
    Flag, 113
structural reflection, 102
subclass, **601**
super call, **601**
superclass, **601**
suspicious_calls
    Flag, 114
synchronized predicate, **601**

synchronized/1
    Directive, 239
syntax_error/1
    Built-in method, 351
system_error/0
    Built-in method, 352

T

tabling
    Flag, 112
tail_recursive
    Flag, 115
template method, **601**
term_expansion/2
    Built-in method, 368
term-expansion, **601**
test dialect, **601**
test outcome, **601**
tester file, **601**
this, **601**
this/1
    Built-in method, 317
threaded engine, **601**
threaded/0
    Directive, 224
threaded/1
    Built-in predicate, 271
threaded_call/1-2
    Built-in predicate, 272
threaded_cancel/1
    Built-in predicate, 279
threaded_engine/1
    Built-in predicate, 284
threaded_engine_create/3
    Built-in predicate, 282
threaded_engine_destroy/1
    Built-in predicate, 283
threaded_engine_fetch/1
    Built-in predicate, 290
threaded_engine_next/2
    Built-in predicate, 286
threaded_engine_next_reified/2
    Built-in predicate, 287
threaded_engine_post/2
    Built-in predicate, 289
threaded_engine_self/1
    Built-in predicate, 285
threaded_engine_yield/1
    Built-in predicate, 288
threaded_exit/1-2
    Built-in predicate, 276
threaded_ignore/1
    Built-in predicate, 275
threaded_notify/1
    Built-in predicate, 281
threaded_once/1-2
    Built-in predicate, 273
threaded_peek/1-2
    Built-in predicate, 277
threaded_wait/1
    Built-in predicate, 280
threads
    Flag, 112
throw/1
    Built-in method, 337
top-level interpreter shorthand, **601**
transparent-box view, 102
trivial_goal_fails
    Flag, 114
true/0
    Built-in method, 310
type_error/2
    Built-in method, 340

U

undefined_predicates
    Flag, 113
underscore_variables
    Flag, 116
unicode
    Flag, 112
unknown_entities
    Flag, 113
unknown_predicates
    Flag, 113
ununinstantiation_error/1
    Built-in method, 339
use_module/1
    Directive, 226
use_module/2
    Directive, 243
uses/1
    Directive, 225
uses/2
    Directive, 240

V

version_data
    Flag, 113
visible predicate, **601**