# Reference Manual

## Grammar

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation units</td>
<td>1</td>
</tr>
<tr>
<td>Object definition</td>
<td>1</td>
</tr>
<tr>
<td>Category definition</td>
<td>2</td>
</tr>
<tr>
<td>Protocol definition</td>
<td>2</td>
</tr>
<tr>
<td>Entity relations</td>
<td>3</td>
</tr>
<tr>
<td>Implemented protocols</td>
<td>3</td>
</tr>
<tr>
<td>Extended protocols</td>
<td>3</td>
</tr>
<tr>
<td>Imported categories</td>
<td>4</td>
</tr>
<tr>
<td>Extended objects</td>
<td>4</td>
</tr>
<tr>
<td>Extended categories</td>
<td>4</td>
</tr>
<tr>
<td>Instantiated objects</td>
<td>5</td>
</tr>
<tr>
<td>Specialized objects</td>
<td>5</td>
</tr>
<tr>
<td>Complemented objects</td>
<td>5</td>
</tr>
<tr>
<td>Entity scope</td>
<td>6</td>
</tr>
<tr>
<td>Entity identifiers</td>
<td>6</td>
</tr>
<tr>
<td>Object identifiers</td>
<td>6</td>
</tr>
<tr>
<td>Category identifiers</td>
<td>6</td>
</tr>
<tr>
<td>Protocol identifiers</td>
<td>7</td>
</tr>
<tr>
<td>Source file names</td>
<td>7</td>
</tr>
<tr>
<td>Directives</td>
<td>8</td>
</tr>
<tr>
<td>Source file directives</td>
<td>8</td>
</tr>
<tr>
<td>Conditional compilation directives</td>
<td>8</td>
</tr>
<tr>
<td>Object directives</td>
<td>8</td>
</tr>
<tr>
<td>Category directives</td>
<td>8</td>
</tr>
<tr>
<td>Protocol directives</td>
<td>9</td>
</tr>
</tbody>
</table>
Predicate directives .......................................................... 9
Clauses and goals .............................................................. 13
Lambda expressions ........................................................... 14
Entity properties ............................................................ 15
Predicate properties ........................................................ 17

Directives

Source file directives

encoding/1 ................................................................. 20
include/1 ............................................................... 33
initialization/1 ......................................................... 35
op/3 ................................................................. 53
set_logtalk_flag/2 .................................................. 21

Conditional compilation directives

if/1 ............................................................. 22
elif/1 ............................................................ 23
else/0 .......................................................... 24
endif/0 .......................................................... 25

Entity directives

built_in/0 ............................................................. 26
category/1-3 ......................................................... 27
dynamic/0 .......................................................... 29
end_category/0 .................................................... 30
end_object/0 ....................................................... 31
end_protocol/0 .................................................... 32
include/1 .......................................................... 33
info/1 ............................................................. 34
initialization/1 ....................................................... 35
object/1-5 ........................................................... 37
op/3 ............................................................... 53
protocol/1-2 ......................................................... 43
set_logtalk_flag/2 .................................................. 21
threaded/0 .......................................................... 44

Predicate directives

alias/2 ............................................................... 45
coiductive/1 .......................................................... 46
discontiguous/1 ....................................................... 47
dynamic/1 ........................................................... 48
info/2 ............................................................... 49
meta_predicate/1 ...................................................... 50
Built-in predicates

Enumerating objects, categories and protocols

current_category/1................................................................. 62
current_object/1............................................................... 63
current_protocol/1............................................................. 64

Enumerating objects, categories and protocols properties

category_property/2.............................................................. 65
object_property/2............................................................... 66
protocol_property/2............................................................ 67

Creating new objects, categories and protocols

create_category/4................................................................. 68
create_object/4................................................................. 69
create_protocol/3............................................................... 71

Abolishing objects, categories and protocols

abolish_category/1............................................................... 72
abolish_object/1................................................................. 73
abolish_protocol/1.............................................................. 74

Objects, categories and protocols relations

extends_object/2-3............................................................... 75
extends_protocol/2-3............................................................ 76
extends_category/2-3............................................................ 77
implements_protocol/2-3..................................................... 78
conforms_to_protocol/2-3...................................................... 83
imports_category/2-3............................................................ 79
instantiates_class/2-3........................................................... 80
specializes_class/2-3............................................................ 81
complements_object/2........................................................... 82
Event handling

abolish_events/5 ................................................................. 84
current_event/5 ................................................................. 85
define_events/5 ................................................................. 86

Multi-threading meta-predicates

threaded/1 ................................................................. 87
threaded_call/1-2 ................................................................. 88
threaded_once/1-2 ................................................................. 89
threaded_ignore/1 ................................................................. 90
threaded_exit/1-2 ................................................................. 91
threaded.peek/1-2 ................................................................. 92
threaded_wait/1 ................................................................. 93
threaded.notify/1 ................................................................. 94

Compiling and loading objects, categories and protocols

logtalk_compile/1 ................................................................. 95
logtalk_compile/2 ................................................................. 96
logtalk_load/1 ................................................................. 98
logtalk_load/2 ................................................................. 99
logtalk_make/0 ................................................................. 101
logtalk_make/1 ................................................................. 102
logtalk_library_path/2 ............................................................... 103
logtalk_load_context/2 ............................................................... 105

Flags

current_logtalk_flag/2 ............................................................... 106
set_logtalk_flag/2 ............................................................... 107
create_logtalk_flag/3 ............................................................... 108

Built-in methods

Execution context methods

parameter/2 ................................................................. 110
self/1 ................................................................. 111
sender/1 ................................................................. 112
this/1 ................................................................. 113

Reflection methods

current_op/3 ................................................................. 114
current_predicate/1 ................................................................. 115
predicate_property/2 ............................................................... 116

Database methods
abolish/1                      117
asserta/1                      118
assertz/1                      119
clause/2                       120
retract/1                      121
retractall/1                   122

Meta-call methods

call/1-N                       123
ignore/1                       124
once/1                         125
\+/1                           126

Exception-handling methods

catch/3                        127
throw/1                        128

All solutions methods

bagof/3                        129
findall/3                      130
findall/4                      131
forall/2                       132
setof/3                        133

Event handler methods

before/3                       134
after/3                        135

Message forwarding methods

forward/1                      136

DCGs non-terminals and parsing methods

call//1-N                      137
phrase//1                      138
phrase/2                       139
phrase/3                       140

Term and goal expansion methods

expand_term/2                  141
term_expansion/2               142
expand_goal/2                  143
goal_expansion/2               144

Coinduction hook predicates

coinductive_success_hook/1-2  154
Control constructs

Message sending

::/2.......................................................... 156
::/1.......................................................... 158

Message delegation

[]/1.......................................................... 160

Calling imported and inherited predicate definitions

^~/1.......................................................... 159

Calling external code

{}/1.......................................................... 162

Context-switching calls

<</2.......................................................... 163

Methods provided by the logtalk built-in object

Message printing methods

print_message/3....................................................... 145
message_tokens//2................................................... 146
message_hook/4..................................................... 147
message_prefix_stream/4.......................................... 148
print_message_tokens/3........................................... 149
print_message_token/4............................................ 150

Question asking methods

ask_question/5...................................................... 151
question_hook/5.................................................... 152
question_prompt_stream/4......................................... 153
Grammar

The Logtalk grammar is here described using Backus-Naur Form syntax. Non-terminal symbols in *italics* have the definition found in the ISO Prolog Standard. Terminal symbols are represented in a *fixed width font* and between double-quotes.

**Entities**

```plaintext
entity ::= object | category | protocol
```

**Object definition**

```plaintext
object ::= begin_object_directive [ object_directives ] [ clauses ] 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

non_prototype_relations ::= non_prototype_relation | non_prototype_relation " , " non_prototype_relations

non_prototype_relation ::= implements_protocols |
Category definition

```plaintext
category ::= begin_category_directive [ category_directives ] [ clauses ] end_category_directive.
```

```plaintext
begin_category_directive ::= ":- category(" category_identifier [ "," category_relations ] ")."
```

```plaintext
end_category_directive ::= ":- end_category."
```

```plaintext
category_relations ::= category_relation |
                    category_relation "," category_relations
```

```plaintext
category_relation ::= implements_protocols |
                    extends_categories |
                    complements_objects
```

Protocol definition

```plaintext
protocol ::= begin_protocol_directive [ protocol_directives ] end_protocol_directive.
```

```plaintext
begin_protocol_directive ::= ":- protocol(" protocol_identifier [ "," extends_protocols ] ")."
```

```plaintext
end_protocol_directive ::= ":- end_protocol."
```
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

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 |

---

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
object_identifier "," complemented_object_sequence

complemented_object_list ::= " [" complemented_object_sequence "]"

**Entity and predicate scope**

scope ::= "public" | "protected" | "private"

**Entity identifiers**

entity_identifiers ::= entity_identifier | entity_identifier_sequence | entity_identifier_list

deployer_identifier ::= object_identifier | protocol_identifier | category_identifier

deployer_identifier_sequence ::= entity_identifier | entity_identifier , entity_identifier_sequence

deployer_identifier_list ::= " [" entity_identifier_sequence "]"

**Object identifiers**

object_identifiers ::= object_identifier | object_identifier_sequence | object_identifier_list

deployer_identifier ::= atom | compound

object_identifier_sequence ::= object_identifier | object_identifier , object_identifier_sequence

object_identifier_list ::= " [" object_identifier_sequence "]"

**Category identifiers**

category_identifiers ::= category_identifier |
category_identifier_sequence | category_identifier_list

category_identifier ::= 
    atom | compound

category_identifier_sequence ::= 
    category_identifier |
    category_identifier " , " category_identifier_sequence

category_identifier_list ::= 
    " [ " category_identifier_sequence " ] "

Protocol identifiers

protocol_identifiers ::= 
    protocol_identifier |
    protocol_identifier_sequence |
    protocol_identifier_list

protocol_identifier ::= 
    atom

protocol_identifier_sequence ::= 
    protocol_identifier |
    protocol_identifier " , " protocol_identifier_sequence

protocol_identifier_list ::= 
    " [ " protocol_identifier_sequence " ] "

Module identifiers

module_identifier ::= 
    atom

Source file names

source_file_names ::= 
    source_file_name |
    source_file_name_list

source_file_name ::= 
    atom |
    library_source_file_name

library_source_file_name ::= 
    library_name " ( " atom " ) "

library_name ::= 
    atom

source_file_name_sequence ::= 
    source_file_name |
source_file_name "," source_file_name_sequence

source_file_name_list ::= 
"["source_file_name_sequence "]"

Directives

Source file directives

source_file_directives ::= 

source_file_directive | 
source_file_directive source_file_directives

source_file_directive ::= 
":- encoding(" atom ")." | 
":- set_logtalk_flag(" atom "," nonvar ")." |

Prolog directives

Conditional compilation directives

conditional_compilation_directives ::= 

conditional_compilation_directive | 
conditional_compilation_directive conditional_compilation_directives

conditional_compilation_directive ::= 
":- if(" callable ")." | 
":- elif(" callable ")." | 
":- else." | 
":- endif.

Object directives

object_directives ::= 

object_directive | 
object_directive object_directives

object_directive ::= 
initialization_directive | 
":- built_in." | 
":- threaded." | 
":- dynamic." | 
":- uses(" object_identifier ")." | 
":- use_module(" module_identifier "," module_predicate_indicator_alias_list ")." | 
":- calls(" protocol_identifiers ")." | 
":- info(" entity_info_list ")." | 
":- set_logtalk_flag(" atom "," nonvar ")." |

Predicate directives

Category directives

category_directives ::= 

category_directive |
category_directive category_directives

category_directive ::= 
  initialization_directive | 
  ":- built_in." | 
  ":- dynamic." | 
  ":- uses(" object_identifier ")." | 
  ":- use_module(" module_identifier "," predicate_indicator_alias_list ")." | 
  ":- calls(" protocol_identifiers ")." | 
  ":- info(" entity_info_list ")." | 
  ":- set_logtalk_flag(" atom "," nonvar ")." |

predicate_directives

Protocol directives

protocol_directives ::= 
  protocol_directive | 
  protocol_directive protocol_directives

protocol_directive ::= 
  initialization_directive | 
  ":- built_in." | 
  ":- dynamic." | 
  ":- info(" entity_info_list ")." | 
  ":- set_logtalk_flag(" atom "," nonvar ")." |

Predicate directives

predicate_directives ::= 
  predicate_directive | 
  predicate_directive predicate_directives

predicate_directive ::= 
  alias_directive | 
  synchronized_directive | 
  uses_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 "," predicate_indicator_alias_list ")." |
":- alias("entity_identifier ", "non_terminal_indicator_alias_list ")."

synchronized_directive ::= ":- synchronized("predicate_indicator ")." | ":- synchronized("non_terminal_indicator ")."

uses_directive ::= ":- uses("object_identifier ", "predicate_indicator_alias_list ")."

scope_directive ::= ":- public("predicate_indicator_term | non_terminal_indicator_term ")." | ":- protected("predicate_indicator_term | non_terminal_indicator_term ")." | ":- private("predicate_indicator_term | non_terminal_indicator_term ")."

mode_directive ::= ":- mode("predicate_mode_term | non_terminal_mode_term ", "number_of_proofs ")."
mixed_directive ::= ":- mixed("predicate_indicator | non_terminal_indicator ", "predicate_info_list ")."

dynamic_directive ::= ":- dynamic("predicate_indicator_term | non_terminal_indicator_term ")."

discontiguous_directive ::= ":- discontiguous("predicate_indicator_term | non_terminal_indicator_term ")."
mixed_directive ::= ":- mixed("predicate_indicator_term ")."

coinductive_directive ::= ":- coinductive("predicate_indicator_term | coinductive_predicate_template_term ")."

predicate_indicator_term ::= predicate_indicator | predicate_indicator_sequence | predicate_indicator_list

predicate_indicator_sequence ::= predicate_indicator | predicate_indicator_sequence , predicate_indicator_sequence

predicate_indicator_list ::= [
"\"predicate_indicator_sequence \""]

predicate_indicator_alias ::= predicate_indicator | predicate_indicator "as" predicate_indicator | predicate_indicator ":" predicate_indicator |
predicate_indicator "::" predicate_indicator

predicate_indicator_alias_sequence ::= 
    predicate_indicator_alias | 
    predicate_indicator_alias "," predicate_indicator_alias_sequence

predicate_indicator_alias_list ::= 
    "(" predicate_indicator_alias_sequence ")"

module_predicate_indicator_alias ::= 
    predicate_indicator | 
    predicate_indicator "as" predicate_indicator | 
    predicate_indicator "::" predicate_indicator

module_predicate_indicator_alias_sequence ::= 
    module_predicate_indicator_alias | 
    module_predicate_indicator_alias "," module_predicate_indicator_alias_sequence

module_predicate_indicator_alias_list ::= 
    "(" module_predicate_indicator_alias_sequence ")"

non_terminal_indicator_term ::= 
    non_terminal_indicator | 
    non_terminal_indicator_sequence | 
    non_terminal_indicator_list

non_terminal_indicator_sequence ::= 
    non_terminal_indicator | 
    non_terminal_indicator "," non_terminal_indicator_sequence

non_terminal_indicator_list ::= 
    "(" non_terminal_indicator_sequence ")"

non_terminal_indicator ::= 
    functor "//" arity

non_terminal_indicator_alias ::= 
    non_terminal_indicator | 
    non_terminal_indicator "as" non_terminal_indicator | 
    non_terminal_indicator "::" non_terminal_indicator

non_terminal_indicator_alias_sequence ::= 
    non_terminal_indicator_alias | 
    non_terminal_indicator_alias "," non_terminal_indicator_alias_sequence

non_terminal_indicator_alias_list ::= 
    "(" non_terminal_indicator_alias_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 ::= 
  \" g \" [ type ] | \" + \" [ type ] | \" = \" [ type ] | \" ? \" [ type ] 
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 \" | \" error \" 
meta_predicate_template_term ::= 
  meta_predicate_template | 
  meta_predicate_template_sequence | 
  meta_predicate_template_list 
meta_predicate_template_sequence ::= 
  meta_predicate_template |
Grammar

meta_predicate_template "," meta_predicate_template_sequence

meta_predicate_template_list ::= 
"([" meta_predicate_template_sequence "]"

meta_predicate_template ::= 
object_identifier ":: atom " (" meta_predicate_specifiers ")" |
category_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_item "is" nonvar "] " entity_info_list "]"

entity_info_item ::= 
"comment" | "remarks" |
"author" | "version" | "date" |
"copyright" | "license" |
"parameters" | "parnames" |
atom

predicate_info_list ::= 
"[]" |
"([" predicate_info_item "is" nonvar "] " predicate_info_list "]"

predicate_info_item ::= 
"comment" |
"arguments" | "argnames" |
"redefinition" | "allocation" |
"examples" | "exceptions" |
atom

Clauses and goals

clause ::= 
object_identifier ":: head ":=" body |
head :- body |

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 ::= 
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 ")"

callable

callable

context_switching_call ::= 
o| _
object_identifier "<<" goal

Lambda expressions

lambda_expression ::= 
lambda_free_variables "/" lambda_parameters ">>" callable |
lambda_free_variables "/" callable |
lambda_parameters ">>" callable

lambda_free_variables ::= 
"(" conjunction of variables ")" |
"(" variable ")" |
\[
\lambda_{\text{parameters}} ::= \text{list of terms} | ()
\]

**Entity properties**

category_property ::= "static" | "dynamic" | "built-in" |
"file(" atom ")" | "file(" atom "," atom ")" |
"lines(" integer "," integer ")" |
"events" |
"source_data" |
"public(" predicate_indicator_list ")" |
"protected(" predicate_indicator_list ")" |
"private(" predicate_indicator_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(" predicate_indicator "," predicate_alias_property_list ")" |
"calls(" predicate_called "," predicate_call_property_list ")" |
"number_of_clauses(" integer ")" |
"number_of_user_clauses(" integer ")"

object_property ::= "static" | "dynamic" | "built-in" |
"threaded" |
"file(" atom ")" |
"file(" atom "," atom ")" |
"lines(" integer "," integer ")" |
"context_switching_calls" |
"dynamic_declarations" |
"events" |
"source_data" |
"complements(" allow" |" restrict" ")" |
"complements" |
"public(" predicate_indicator_list ")" |
"protected(" predicate_indicator_list ")" |
"private(" predicate_indicator_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( "predicate_indicator", "predicate_alias_property_list" )" |
"calls( "predicate_called", "predicate_call_property_list" )" |
"number_of_clauses( "integer" )" |
"number_of_user_clauses( "integer" )"

protocol_property ::= "static" |
"dynamic" |
"built_in" |
"source_data" |
"file( atom )" |
"file( atom, atom )" |
"lines( integer, integer )" |
"public( predicate_indicator_list )" |
"protected( predicate_indicator_list )" |
"private( predicate_indicator_list )" |
"declares( predicate_indicator, predicate_declaration_property_list )" |
"alias( predicate_indicator, predicate_alias_property_list )"

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" |
"meta_predicate( meta_predicate_template )" |
"coinductive( coinductive_predicate_template )" |
"non_terminal( non_terminal_indicator )" |
"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 properties**

```
predicate_property ::=
    "static" | "dynamic" |
    "scope" ("scope") |
    "private" | "protected" | "public" |
    "logtalk" | "prolog" | "foreign" |
    "coinductive" ("coinductive_predicate_template") |
    "multifile" |
    "synchronized" |
```
Compiler flags

compiler_flag ::= flag(flag_value)
Directives
encoding/1

Description

encoding(Enciding)

Declares the source file text encoding. This is an experimental source file directive, which is only supported on some back-end Prolog compilers. When used, this directive must be the first term in the source file in the first line. Currently recognized encodings values include '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'. Be sure to use an encoding supported by the chosen back-end Prolog compiler (whose adapter file must define a table that translates between the Logtalk and Prolog-specific atoms that represent each supported encoding). When writing portable code that cannot be expressed using ASCII, 'UTF-8' is usually the best choice.

Template and modes

encoding(+atom)

Examples

:= encoding('UTF-8').
**set_logtalk_flag/2**

**Description**

\[\text{set_logtalk_flag(Flag, Value)}\]

Sets Logtalk flag values. The scope of this directive is the entity containing it or the source file being compiled. For global scope, use the corresponding \text{set_logtalk_flag/2} built-in predicate within an \text{initialization/1} directive.

**Template and modes**

\[\text{set_logtalk_flag(+atom, +nonvar)}\]

**Errors**

- Flag is a variable:
  \text{instantiation_error}
- Value is a variable:
  \text{instantiation_error}
- Flag is not an atom:
  \text{type_error(atom, Flag)}
- Flag is neither a variable nor a valid flag:
  \text{domain_error(flag, Flag)}
- Value is not a valid value for flag Flag:
  \text{domain_error(flag_value, Flag + Value)}
- Flag is a read-only flag:
  \text{permission_error(modify, flag, Flag)}

**Examples**

\[\text{:- set_logtalk_flag(unknown_entities, silent).}\]
### if/1

**Description**

<table>
<thead>
<tr>
<th>if(Goal)</th>
</tr>
</thead>
</table>

Starts conditional compilation. The code following the directive is compiled if `Goal` is true. The goal is subjected to goal expansion before execution.

**Template and modes**

<table>
<thead>
<tr>
<th>if(@callable)</th>
</tr>
</thead>
</table>

**Examples**

```prolog
:- if(current_prolog_flag(double_quotes, atom)).
```
**elsif/1**

**Description**

**elsif(Goal)**

Supports embedded conditionals when performing conditional compilation. The code following the directive is compiled if `Goal` is true. The goal is subjected to goal expansion before execution.

**Template and modes**

**elsif(@callable)**

**Examples**

```prolog
:- elsif(predicate_property(callable(_), built_in)).
```
else/0

Description

else

Starts a else branch when performing conditional compilation.

Template and modes

else

Examples

:= else.
### endif/0

**Description**

```
endif
```

Ends conditional compilation.

**Template and modes**

```
endif
```

**Examples**

```
:- endif.
```
**built_in/0**

**Description**

`built_in`

Declares an entity as built-in. Built-in entities cannot be redefined.

**Template and modes**

`built_in`

**Examples**

`:= built_in.`
category/1-3

Description

category(Category)

category(Category, implements(Protocols))

category(Category, extends(Categories))

category(Category, 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), complements(+complemented_objects))
Examples

```prolog
:- category(monitoring).

:- category(monitoring, implements(monitoringp)).

:- category(attributes, implements(protected::variables)).

:- category(extended, extends(minimal)).

:- category(logging, implements(monitoring), implements(logging), complements(employee)).
```
**dynamic/0**

**Description**

| dynamic |

Declares an entity and its contents as dynamic. Dynamic entities can be abolished at runtime.

**Template and modes**

| dynamic |

**Examples**

```latex
:- dynamic.
```
end_category/0

Description

end_category

Ending category directive.

Template and modes

end_category

Examples

:= end_category.
end_object/0

Description

```
end_object
```

Ending object directive.

Template and modes

```
end_object
```

Examples

```
:- end_object.
```
**end_protocol/0**

**Description**

```
end_protocol
```

Ending protocol directive.

**Template and modes**

```
end_protocol
```

**Examples**

```
:= end_protocol.
```
**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. If the file name have an extension, it must not be omitted.

This directive can be used as either a 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.

**Template and modes**

```
include(@source_file_name)
```

**Examples**

```
:~ include(data('raw_1.txt')).
:~ include('factbase.pl').
:~ include('/home/me/databases/cities.pl').
?- create_object(cities, [], [public(city/4), include('/home/me/databases/cities.pl')], []).
```
info/1

Description

Documentation directive for objects, protocols, and categories. The directive argument is a list of pairs using the format \textit{Key is Value}. See the documenting Logtalk programs section for a description of the valid keys.

Template and modes

\begin{verbatim}
info(+entity_info_list)
\end{verbatim}

Examples

\begin{verbatim}
:- info([  
    version is 1.0,  
    author is 'Paulo Moura',  
    date is 2000/4/20,  
    comment is 'List protocol.' 
  ]).
\end{verbatim}
**initialization/1**

**Description**

`initialization(Goal)`

When used within an object, this directive defines a goal to be called immediately after the object has been loaded into memory. When used at a global level within a source file, this directive defines a goal to be called immediately after the compiled source file is loaded into memory.

**Template and modes**

`initialization(@callable)`

**Examples**

`:- initialization(init).`
multifile/1

Description

```
multifile(Functor/Arity)
multifile((Functor1/Arity1, Functor2/Arity2, ...))
multifile([Functor1/Arity1, Functor2/Arity2, ...])
multifile(Entity::Functor/Arity)
multifile((Entity1::Functor1/Arity1, Entity2::Functor2/Arity2, ...))
multifile([Entity1::Functor1/Arity1, Entity2::Functor2/Arity2, ...])
multifile(Functor//Arity)
multifile((Functor1//Arity1, Functor2//Arity2, ...))
multifile([Functor1//Arity1, Functor2//Arity2, ...])
multifile(Entity::Functor//Arity)
multifile((Entity1::Functor1//Arity1, Entity2::Functor2//Arity2, ...))
multifile([Entity1::Functor1//Arity1, Entity2::Functor2//Arity2, ...])
```

 Declares multifile predicates and multifile grammar rule non-terminals. The predicate (or non-terminal) must also be declared public in the object (or category) holding its primary declaration (i.e. the declaration without the Entity:: prefix). Protocols cannot declare multifile predicates as protocols cannot contain predicate definitions.

Template and modes

```
multifile(+predicate_indicator_term)
multifile(+non_terminal_indicator_term)
multifile(+object_identifier::+predicate_indicator_term)
multifile(+object_identifier::+non_terminal_indicator_term)
multifile(+category_identifier::+predicate_indicator_term)
multifile(+category_identifier::+non_terminal_indicator_term)
```

Examples

```
:- multifile(table/3).
:- multifile(user::hook/2).
```
**object/1-5**

**Description**

*Stand-alone objects (prototypes)*

```plaintext
object (Object)

object (Object, implements (Protocols))

object (Object, imports (Categories))

object (Object, implements (Protocols), imports (Categories))
```

**Prototype extensions**

```plaintext
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,
       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

<table>
<thead>
<tr>
<th>Class with metaclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>implements(+implemented_protocols),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>imports(+imported_categories),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>implements(+implemented_protocols),</td>
</tr>
<tr>
<td>imports(+imported_categories),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class with metaclasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>instantiates(+instantiated_objects),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>implements(+implemented_protocols),</td>
</tr>
<tr>
<td>instantiates(+instantiated_objects),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>imports(+imported_categories),</td>
</tr>
<tr>
<td>instantiates(+instantiated_objects),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
<tr>
<td>object(+object_identifier,</td>
</tr>
<tr>
<td>implements(+implemented_protocols),</td>
</tr>
<tr>
<td>imports(+imported_categories),</td>
</tr>
<tr>
<td>instantiates(+instantiated_objects),</td>
</tr>
<tr>
<td>specializes(+specialized_objects))</td>
</tr>
</tbody>
</table>
Examples

```prolog
:- 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)).
```
**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)).
```
threaded/0

Description

threaded

Declares that an object supports 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.

Template and modes

threaded

Examples

:= threaded.
**alias/2**

**Description**

```
alias(Entity, PredicateAliases)
alias(Entity, NonTerminalAliases)
```

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 `Functor/Arity as Alias/Arity` or, in alternative, the notation `Functor/Arity::Alias/Arity`.

**Template and modes**

```
alias(@entity_identifier, +list(predicate_indicator_alias))
alias(@entity_identifier, +list(non_terminal_indicator_alias))
```

**Examples**

```
:- alias(list, [member/2 as list_member/2]).
:- alias(set, [member/2 as set_member/2]).
:- alias(words, [singular//0 as peculiar//0]).
```
coinductive/1

Description

This is an experimental directive, used for declaring coinductive predicates. Requires a back-end 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 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 "+" or "-" ) 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/2 or coinductive_success_hook/1 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(+coinductive_predicate_template_term)

Examples

:~ coinductive(comember/2).
:~ coinductive(controller(+,+,+,-,-)).
**discontiguous/1**

**Description**

| discontiguous(Functor/Arity) |
| discontiguous((Functor1/Arity1, Functor2/Arity2, ...)) |
| discontiguous([Functor1/Arity1, Functor2/Arity2, ...]) |
| discontiguous(Functor//Arity) |
| discontiguous((Functor1//Arity1, Functor2//Arity2, ...)) |
| discontiguous([Functor1//Arity1, Functor2//Arity2, ...]) |

Declares discontiguous predicates and discontiguous grammar rule non-terminals. The use of this directive should be avoided as not all Prolog compilers support discontiguous predicates.

**Template and modes**

| discontiguous(+predicate_indicator_term) |
| discontiguous(+non_terminal_indicator_term) |

**Examples**

```prolog
:- discontiguous(counter/1).

:- discontiguous((lives/2, works/2)).

:- discontiguous([db/4, key/2, file/3]).
```
**dynamic/1**

**Description**

The predicate indicators (non-terminal indicators) can be explicitly qualified with an object identifier or a category identifier when the predicates (non-terminals) are also declared multifile.

**Template and modes**

```
dynamic(+predicate_indicator_term)
dynamic(+non_terminal_indicator_term)

dynamic(+object_identifier::+predicate_indicator_term)
dynamic(+object_identifier::+non_terminal_indicator_term)

dynamic(+category_identifier::+predicate_indicator_term)
dynamic(+category_identifier::+non_terminal_indicator_term)
```

**Examples**

```
:- dynamic(counter/1).
:- dynamic([lives/2, works/2]).
:- dynamic([db/4, key/2, file/3]).
```
**info/2**

**Description**

```prolog
info Functor/Arity, List
info Functor//Arity, List
```

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 documenting Logtalk programs section for a description of the valid keys.

**Template and modes**

```prolog
info(+predicate_indicator, +predicate_info_list)
info(+non_terminal_indicator, +predicate_info_list)
```

**Examples**

```prolog
:- 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.'
  ]).
```
meta_predicate/1

Description

<table>
<thead>
<tr>
<th>meta_predicate(MetaPredicateTemplate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_predicate(((MetaPredicateTemplate1, MetaPredicateTemplate2, ...)))</td>
</tr>
<tr>
<td>meta_predicate([MetaPredicateTemplate1, MetaPredicateTemplate2, ...])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>meta_predicate(Entity::MetaPredicateTemplate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_predicate(((Entity1::MetaPredicateTemplate1, Entity2::MetaPredicateTemplate2, ...)))</td>
</tr>
<tr>
<td>meta_predicate([Entity1::MetaPredicateTemplate1, Entity2::MetaPredicateTemplate2, ...])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>meta_predicate(Module::MetaPredicateTemplate)</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_predicate(((Module1::MetaPredicateTemplate1, Module2::MetaPredicateTemplate2, ...)))</td>
</tr>
<tr>
<td>meta_predicate([Module1::MetaPredicateTemplate1, Module2::MetaPredicateTemplate2, ...])</td>
</tr>
</tbody>
</table>

Declares meta-predicates, i.e., predicates that have arguments that will be called as goals. An argument may also be a closure instead of a goal if the meta-predicate uses the \texttt{call/N} Logtalk built-in methods to construct and call the actual goal from the closure and the additional arguments.

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 meta-call. Normal arguments are represented by the atom * . Meta-arguments are always called in the meta-predicate calling context, not in the meta-predicate definition context.

Logtalk allows the use of this directive to override the original meta-predicate directive. This is sometimes necessary when calling Prolog module meta-predicates due to the lack of standardization of the syntax of the meta-predicate templates.

Template and modes

<table>
<thead>
<tr>
<th>meta_predicate(+meta_predicate_template_term)</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta_predicate(+object_identifier::+meta_predicate_template_term)</td>
</tr>
<tr>
<td>meta_predicate(+category_identifier::+meta_predicate_template_term)</td>
</tr>
<tr>
<td>meta_predicate(+module_identifier:+meta_predicate_template_term)</td>
</tr>
</tbody>
</table>

Examples

```prolog
:- meta_predicate(findall(*, 0, *)).
:- meta_predicate(forall(0, 0)).
:- meta_predicate(maplist(2, *, *)).
```
**meta_non_terminal/1**

**Description**

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` Logtalk built-in methods to construct and call the actual non-terminal from the closure and the additional arguments.

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.

Logtalk allows the use of this directive to override the original meta-non-terminal directive. This is sometimes necessary when calling Prolog module meta-non-terminals due to the lack of standardization of the syntax of the meta-non-terminal templates.

**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(findall(*, 0, *)).
:- meta_non_terminal(forall(0, 0)).
:- meta_non_terminal(maplist(2, *, *)).
```
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 (not necessarily distinct solutions). You may also use this directive for documenting grammar rule non-terminals. Multiple directives may be used to specify the same predicate or grammar rule non-terminal.

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).
```
**op/3**

**Description**

\[
\text{op(Precedence, Associativity, Operator)}
\]

Declares operators. Operators declared inside objects and categories have local scope. Global operators can be declared inside a source file by writing the respective directives before the entity opening directives.

**Template and modes**

\[
\text{op(+integer, +associativity, +atom_or_atom_list)}
\]

**Examples**

\[
:- \text{op}(950, \text{fx}, +).
:- \text{op}(950, \text{fx}, ?).
:- \text{op}(950, \text{fx}, @).
:- \text{op}(950, \text{fx}, -).
\]
**private/1**

**Description**

```
private(Functor/Arity)
private((Functor1/Arity1, Functor2/Arity2, ...))
private([Functor1/Arity1, Functor2/Arity2, ...])
private(Functor1//Arity1, Functor2//Arity2, ...))
private([Functor1//Arity1, Functor2//Arity2, ...])
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]).
```
### protected/1

**Description**

<table>
<thead>
<tr>
<th>protected(Functor/Arity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected((Functor1/Arity1, Functor2/Arity2, ...))</td>
</tr>
<tr>
<td>protected([Functor1/Arity1, Functor2/Arity2, ...])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>protected(Functor//Arity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected((Functor1//Arity1, Functor2//Arity2, ...))</td>
</tr>
<tr>
<td>protected([Functor1//Arity1, Functor2//Arity2, ...])</td>
</tr>
</tbody>
</table>

| 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` messages sent from the object containing the directive or from an object that inherits the directive. Protected operators are not inherited but declaring them provides useful information for defining descendant objects.

#### Template and modes

- `protected(+predicate_indicator_term)`
- `protected(+non_terminal_indicator_term)`
- `protected(+operator_declaration)`

#### Examples

```prolog
:- protected(init/1).
:- protected((print/2, convert/4)).
:- protected([load/1, save/3]).
```
**public/1**

**Description**

 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` messages sent from any object. Public operators are not exported but declaring them provides useful information for defining client objects.

**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]).
```
**synchronized/1**

**Description**


dsynchronized(Functor/Arity)
synchronized((Functor1/Arity1, Functor2/Arity2, ...))
synchronized([Functor1/Arity1, Functor2/Arity2, ...])

dsynchronized(Functor//Arity)
synchronized((Functor1//Arity1, Functor2//Arity2, ...))
synchronized([Functor1//Arity1, Functor2//Arity2, ...])

Declares synchronized predicates and synchronized grammar rule non-terminals. 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 declared in the same synchronized directive share the same mutex. In order to use a separate mutex for each predicate (so that they are independently synchronized), a per-predicate synchronized directive must be used.

Declaring a predicate synchronized implicitly makes it deterministic. When using a single-threaded back-end 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]).
```
uses/2

Description

<table>
<thead>
<tr>
<th>uses(Object, Predicates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uses(Object, PredicatesAndAliases)</td>
</tr>
<tr>
<td>uses(Object, NonTerminals)</td>
</tr>
<tr>
<td>uses(Object, NonTerminalsAndAliases)</td>
</tr>
</tbody>
</table>

Declares that all calls (made from predicates defined in the category or object containing the directive) to the specified predicates 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 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, `op(Precedence, Associativity, Operator)`, in the second argument.

This directive is also used when compiling calls to the database and reflection built-in methods by looking into these methods predicate arguments.

It is possible to specify a predicate alias using the notation `Functor/Arity as Alias/Arity` or, in alternative, the notation `Functor/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 using context of the predicates.

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

<table>
<thead>
<tr>
<th>uses(+object_identifier, +predicate_indicator_list)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uses(+object_identifier, +predicate_indicator_alias_list)</td>
</tr>
<tr>
<td>uses(+object_identifier, +non_terminal_indicator_list)</td>
</tr>
<tr>
<td>uses(+object_identifier, +non_terminal_indicator_alias_list)</td>
</tr>
</tbody>
</table>

Examples

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

foo :-
    ...
    findall(X, member(X, L), A),  % the same as findall(X, list::member(X, L), A)
    append(A, B, C),              % the same as list::append(A, B, C)
    assertz(data(X, C)),          % the same as store::assertz(data(X, C))
    ...
```
Another example, using the extended notation that allows us to define predicate aliases:

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

btree_to_queue :-
    ..., 
    new_btree(Tree),     % the same as btrees::new(Tree)
    new_queue(Queue),    % the same as queues::new(Queue)
    ...
```
use_module/2

Description

use_module(Module, Predicates)

This directive is supported only when using a back-end Prolog compiler that supports modules. It declares that all calls (made from predicates defined in the category or object containing the directive) to the specified predicates are to be interpreted as calls to explicitly-qualified module predicates. Thus, this directive may be used to simplify 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, op(Precedence, Associativity, Operator), in the second argument.

This directive is also used when compiling calls to the database and reflection built-in methods by examining these methods predicate arguments.

It is possible to specify a predicate alias using the notation Functor/Arity as Alias/Arity or, in alternative, the notation Functor/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 using context of the predicates.

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 of the used backend Prolog compiler and must be done separately (usually, using a source file use_module/1 or use_module/2 directive in the entity file or in the application loader file). Also note that the name of the module may differ from the name of the module file.

The modules should 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.

Template and modes

use_module(+module_identifier, +predicate_indicator_list)

Examples

:- use_module(lists, [append/3, member/2]).
:- use_module(store, [data/2]).

foo :-

..., % the same as findall(X, lists:member(X, L), A)
findall(X, member(X, L), A),
append(A, B, C), % the same as lists:append(A, B, C)
assertz(data(X, C)), % the same as assertz(store:data(X, C))
assertz(data(X, C)),
...

...
Built-in predicates
**current_category/1**

**Description**

| current_category(Category) |

Enumerates, by backtracking, all currently defined categories. All categories are found, either static, dynamic, or built-in.

**Template and modes**

| current_category(?category_identifier) |

**Errors**

Category is neither a variable nor a valid category identifier:

```
type_error(category_identifier, Category)
```

**Examples**

| |- current_category(monitoring). |
**current_object/1**

**Description**

```
current_object(Object)
```

Enumerates, by backtracking, all currently defined objects. All objects are found, either static, dynamic or built-in.

**Template and modes**

```
current_object(?object_identifier)
```

**Errors**

Object is neither a variable nor a valid object identifier:
```
type_error(object_identifier, Object)
```

**Examples**

```
?- current_object(list).
```
**current_protocol/1**

**Description**

```
current_protocol(Protocol)
```

Enumerates, by backtracking, all currently defined protocols. All protocols are found, either static, dynamic, or built-in.

**Template and modes**

```
current_protocol(?protocol_identifier)
```

**Errors**

Protocol is neither a variable nor a valid protocol identifier:

```
type_error(protocol_identifier, Protocol)
```

**Examples**

```
| ?- current_protocol(listp).
```
**category_property/2**

**Description**

```
category_property(Category, Property)
```

Enumerates, by backtracking, the properties associated with the defined categories. The valid category properties are listed in the language grammar.

**Template and modes**

```
category_property(?category_identifier, ?category_property)
```

**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**

```
?- category_property(Category, dynamic).
```
**object_property/2**

**Description**

```
object_property(Object, Property)
```

Enumerates, by backtracking, the properties associated with the defined objects. The valid object properties are listed in the language grammar.

**Template and modes**

```
object_property(?object_identifier, ?object_property)
```

**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**

```
| ?- object_property(list, Property).
```
**protocol_property/2**

**Description**

\[ \text{protocol_property(Protocol, Property)} \]

Enumerates, by backtracking, the properties associated with the currently defined protocols. The valid protocol properties are listed in the language grammar.

**Template and modes**

\[ \text{protocol_property(?protocol_identifier, ?protocol_property)} \]

**Errors**

- Protocol is neither a variable nor a valid protocol identifier:
  \[ \text{type_error(protocol_identifier, Protocol)} \]
- Property is neither a variable nor a callable term:
  \[ \text{type_error(callable, Property)} \]
- Property is a callable term but not a valid protocol property:
  \[ \text{domain_error(protocol_property, Property)} \]

**Examples**

\[ \text{?- protocol_property(listp, Property).} \]
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.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

Template and modes

create_category(?category_identifier, +list, +list, +list)

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(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
Relations is neither a variable nor a proper list:
type_error(list, Relations)
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)]  
|   ).
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.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

Template and modes

create_object(?object_identifier, +list, +list, +list)

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(replace, category, Identifier)
permission_error(replace, object, Identifier)
permission_error(replace, protocol, Identifier)
Relations is neither a variable nor a proper list:
type_error(list, Relations)
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

Creating a simple, stand-alone object (a prototype):

| ?- create_object(translator, [], [public(int/2)], [int(0, zero)]).

Creating a new prototype derived from a parent prototype:

| ?- create_object(mickey, [extends(mouse)], [public(alias/1)], [alias(mortimer)]).

Creating a new class instance:

| ?- create_object(p1, [instantiates(person)], [], [name('Paulo Moura'), age(42)].
Creating a new class as a specialization of another class:

\[
?- \text{create} \text{\_} \text{object} \text{\(\text{hovercraft}\)}, \langle\text{specializes} \langle\text{vehicle}\rangle\rangle, \langle\text{public}\langle\text{propeller}/2, \text{fan}/2\rangle\rangle, \langle\text{\_\_\_}\rangle).
\]

Creating a new object and defining its initialization goal:

\[
?- \text{create} \text{\_} \text{object} \text{\(\text{runner}\)}, \langle\text{instantiates} \langle\text{runners}\rangle\rangle, \langle\text{initialization} \langle\text{start}\rangle, \text{\_\_\_\_\_}\rangle, \langle\text{length}/22, \text{time}/60\rangle).
\]

Creating a new empty object with dynamic predicate declarations support:

\[
?- \text{create} \text{\_} \text{object} \text{\(\text{database}\)}, \langle\text{\_\_\_}\rangle, \langle\text{set} \text{\_logtalk\_flag}(\text{dynamic\_declarations}, \text{allow})\rangle, \langle\text{\_\_\_}\rangle).
\]
**create_protocol/3**

**Description**

\[
\text{create_protocol(Identifier, Relations, Directives)}
\]

Creates a new, dynamic protocol. This predicate is often used as a primitive to implement high-level protocol creation methods.

When using Logtalk multi-threading features, predicates calling this built-in predicate may need to be declared synchronized in order to avoid race conditions.

**Template and modes**

\[
\text{create_protocol(?protocol_identifier, +list, +list)}
\]

**Errors**

- Either Relations or Directives is a variable:
  - \text{instantiation_error}
- Identifier is neither a variable nor a valid protocol identifier:
  - \text{type_error(protocol_identifier, Identifier)}
- Identifier is already in use:
  - \text{permission_error(replace, category, Identifier)}
  - \text{permission_error(replace, object, Identifier)}
  - \text{permission_error(replace, protocol, Identifier)}
- Relations is neither a variable nor a proper list:
  - \text{type_error(list, Relations)}
- Directives is neither a variable nor a proper list:
  - \text{type_error(list, Directives)}

**Examples**

\[
| ?- create_protocol(  
| \quad \text{logging,}  
| \quad \text{[extends(monitoring)],}  
| \quad \text{[public([log_file/1, log_on/0, log_off/0])]}  
| \quad ).
\]
abolish_category/1

Description

abolish_category(Category)

Removes from the database a dynamic category.

Template and modes

abolish_category(@category_identifier)

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).
abolish_object/1

Description
abolish_object(Object)
Removes from the database a dynamic object.

Template and modes
abolish_object(@object_identifier)

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).
abolish_protocol/1

Description

abolish_protocol(Protocol)

Removes from the database a dynamic protocol.

Template and modes

abolish_protocol(@protocol_identifier)

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).
**extends_object/2-3**

**Description**

```prolog
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`.

**Template and modes**

```prolog
extends_object(?object_identifier, ?object_identifier)
extends_object(?object_identifier, ?object_identifier, ?scope)
```

**Errors**

- Prototype is neither a variable nor a valid object identifier:
  ```prolog
type_error(object_identifier, Prototype)
  ```
- Parent is neither a variable nor a valid object identifier:
  ```prolog
type_error(object_identifier, Parent)
  ```
- Scope is neither a variable nor an atom:
  ```prolog
type_error(atom, Scope)
  ```
- Scope is not a valid entity scope:
  ```prolog
domain_error(scope, Scope)
  ```

**Examples**

```
?- extends_object(Object, state_space).

?- extends_object(Object, list, public).
```
extends_protocol/2-3

Description

extends_protocol(Protocol1, Protocol2)
extends_protocol(Protocol1, Protocol2, 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.

Template and modes

extends_protocol(?protocol_identifier, ?protocol_identifier)
extends_protocol(?protocol_identifier, ?protocol_identifier, ?scope)

Errors

Protocol1 is neither a variable nor a valid protocol identifier:
   type_error(protocol_identifier, Protocol1)
Protocol2 is neither a variable nor a valid protocol identifier:
   type_error(protocol_identifier, Protocol2)
Scope is neither a variable nor an atom:
   type_error(atom, Scope)
Scope is not a valid entity scope:
   domain_error(scope, Scope)

Examples

| ?- extends_protocol(listp, Protocol).
| ?- extends_protocol(Protocol, termp, private).
extends_category/2-3

Description

```
extends_category(Category1, Category2)
extends_category(Category1, Category2, 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`.

Template and modes

```
extends_category(?category_identifier, ?category_identifier)
extends_category(?category_identifier, ?category_identifier, ?scope)
```

Errors

- Category1 is neither a variable nor a valid protocol identifier:
  \[ \text{type_error(category_identifier, Category1)} \]
- Category2 is neither a variable nor a valid protocol identifier:
  \[ \text{type_error(category_identifier, Category2)} \]
- Scope is neither a variable nor an atom:
  \[ \text{type_error(atom, Scope)} \]
- Scope is not a valid entity scope:
  \[ \text{domain_error(scope, Scope)} \]

Examples

```
| ?- extends_category(basic, Category). |
| ?- extends_category(Category, extended, private). |
```
implements_protocol/2-3

Description

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; it does not implement a transitive closure.

Template and modes

| implements_protocol(?object_identifier, ?protocol_identifier) |
| implements_protocol(?category_identifier, ?protocol_identifier) |
| implements_protocol(?object_identifier, ?protocol_identifier, ?scope) |
| implements_protocol(?category_identifier, ?protocol_identifier, ?scope) |

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 not a valid entity scope:  
  domain_error(scope, Scope)

Examples

| ?- implements_protocol(list, listp). |
| ?- implements_protocol(list, listp, public). |
**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`.

**Template and modes**

```
imports_category(?object_identifier, ?category_identifier)
imports_category(?object_identifier, ?category_identifier, ?scope)
```

**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 not a valid entity scope:
```
domain_error(scope, Scope)
```

**Examples**

```
| ?- imports_category(debugger, monitoring).
| ?- imports_category(Object, monitoring, protected).
```
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.

Template and modes

| instantiates_class(?object_identifier, ?object_identifier)  
| instantiates_class(?object_identifier, ?object_identifier, ?scope)  

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 not a valid entity scope:
  domain_error(scope, Scope)  

Examples

| ?- instantiates_class(water_jug, state_space).  
| ?- instantiates_class(Space, state_space, public).  

**specializes_class/2-3**

**Description**

\[
\text{specializes_class(Class, Superclass)} \\
\text{specializes_class(Class, Superclass, Scope)}
\]

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`.

**Template and modes**

\[
\text{specializes_class(?object_identifier, ?object_identifier)} \\
\text{specializes_class(?object_identifier, ?object_identifier, ?scope)}
\]

**Errors**

Class is neither a variable nor a valid object identifier:  
\text{type_error(object_identifier, Class)}  
Superclass is neither a variable nor a valid object identifier:  
\text{type_error(object_identifier, Superclass)}  
Scope is neither a variable nor an atom:  
\text{type_error(atom, Scope)}  
Scope is not a valid entity scope:  
\text{domain_error(scope, Scope)}

**Examples**

| `- specializes_class(Subclass, state_space).  
| `- specializes_class(Subclass, state_space, public).
**complements_object/2**

**Description**

complements_object(Category, Object)

Enumerates, by backtracking, all category–object pairs such that the category explicitly complements the object.

**Template and modes**

complements_object(?category_identifier, ?object_identifier)

**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**

| |- complements_object(logging, employee). |
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.

Template and modes

```
conforms_to_protocol(?object_identifier, ?protocol_identifier)
conforms_to_protocol(?category_identifier, ?protocol_identifier)
conforms_to_protocol(?object_identifier, ?protocol_identifier, ?scope)
conforms_to_protocol(?category_identifier, ?protocol_identifier, ?scope)
```

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 not a valid entity scope:
  ```
  domain_error(scope, Scope)
  ```

Examples

```
| ?- conforms_to_protocol(list, listp).
| ?- conforms_to_protocol(list, listp, public).
```
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`.

Template and modes

abolish_events(@event, @object_identifier, @callable, @object_identifier, @object_identifier)

Errors

- Event is neither a variable nor a valid event identifier:
  
  ```prolog
  type_error(event, Event)
  ```

- Object is neither a variable nor a valid object identifier:
  
  ```prolog
  type_error(object_identifier, Object)
  ```

- Message is neither a variable nor a callable term:
  
  ```prolog
  type_error(callable, Message)
  ```

- Sender is neither a variable nor a valid object identifier:
  
  ```prolog
  type_error(object_identifier, Sender)
  ```

- Monitor is neither a variable nor a valid object identifier:
  
  ```prolog
  type_error(object_identifier, Monitor)
  ```

Examples

```prolog
| ?- abolish_events(_, list, _, _, debugger).
```
**current_event/5**

**Description**

```prolog
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`.

**Template and modes**

```prolog
```

**Errors**

- Event is neither a variable nor a valid event identifier:
  ```prolog
  type_error(event, Event)
  ```
- Object is neither a variable nor a valid object identifier:
  ```prolog
  type_error(object_identifier, Object)
  ```
- Message is neither a variable nor a callable term:
  ```prolog
  type_error(callable, Message)
  ```
- Sender is neither a variable nor a valid object identifier:
  ```prolog
  type_error(object_identifier, Sender)
  ```
- Monitor is neither a variable nor a valid object identifier:
  ```prolog
  type_error(object_identifier, Monitor)
  ```

**Examples**

```prolog
| ?- current_event(Event, Object, Message, Sender, debugger).
```
**define_events/5**

**Description**

```prolog
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`. The object `Monitor` must define the event handler methods required by the `Event` argument.

**Template and modes**

```prolog
define_events(@event, @object_identifier, @callable, @object_identifier, +object_identifier)
```

**Errors**

- Event is neither a variable nor a valid event identifier:
  ```prolog
type_error(event, Event)
```
- Object is neither a variable nor a valid object identifier:
  ```prolog
type_error(object_identifier, Object)
```
- Message is neither a variable nor a callable term:
  ```prolog
type_error(callable, Message)
```
- Sender is neither a variable nor a valid object identifier:
  ```prolog
type_error(object_identifier, Sender)
```
- Monitor is a variable:
  ```prolog
instantiation_error
```
- Monitor is neither a variable nor a valid object identifier:
  ```prolog
existence_error(object_identifier, Monitor)
```
- Monitor does not define the required `before/3` method:
  ```prolog
existence_error(procedure, before/3)
```
- Monitor does not define the required `after/3` method:
  ```prolog
existence_error(procedure, after/3)
```

**Examples**

```prolog
| ?- define_events(_, list, member(_, _), _, debugger).
```
threaded/1

Description

```
threaded(Goals)
threaded(Conjunction)
threaded(Disjunction)
```

Proves each goal in a conjunction (disjunction) of goals in its own thread. This 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.

When the argument is a disjunction of goals, a call to this predicate blocks until either one of the goals succeeds, all the goals fail, or one of the goals generate an exception; the success 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 one of the goals is true.

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.

Template and modes

```
threaded(+callable)
```

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))
threaded_call/1-2

Description

```prolog
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 succeeds and return immediately. The results (success, failure, or exception) are sent back to the message queue of the object containing the call (this); they can be retrieved by calling the threaded_exit/1 predicate.

The variant threaded_call/2 returns a threaded call identifier tag that can be used with the threaded_exit/2 predicate. Tags shall be regarded as an opaque term; users shall not rely on its type.

Template and modes

```prolog
threaded_call(@callable)
threaded_call(@callable, -nonvar)
```

Errors

- Goal is a variable:
  ```prolog
  instantiation_error
  ```
- Goal is neither a variable nor a callable term:
  ```prolog
  type_error(callable, Goal)
  ```
- Tag is not a variable:
  ```prolog
  type_error(variable, Goal)
  ```

Examples

- Prove Goal asynchronously in a new thread:
  ```prolog
  threaded_call(Goal)
  ```
- Prove ::Message asynchronously in a new thread:
  ```prolog
  threaded_call(::Message)
  ```
- Prove Object::Message asynchronously in a new thread:
  ```prolog
  threaded_call(Object::Message)
  ```
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 variant threaded_once/2 returns a threaded call identifier tag that can be used with the threaded_exit/2 predicate. Tags shall be regarded as an opaque term; users shall not rely on its type.

Template and modes

threaded_once(@callable)
threaded_once(@callable, -nonvar)

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:

type_error(variable, Goal)

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)
**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`).

**Template and modes**

```
threaded_ignore(@callable)
```

**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)
  ```
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 call. When the predicate argument is subsumed by the threaded_call/1 call argument, the threaded_exit/1 call will succeed iff its argument is a solution of the (more general) goal.

The variant threaded_exit/2 accepts a threaded call identifier tag generated by the calls to the threaded_call/1 and threaded_once/2 predicates. Tags shall be regarded as an opaque term; users shall not rely on its type.

Template and modes

threaded_exit(+callable)
threaded_exit(+callable, +nonvar)

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:
existence_error(goal_thread, Goal)
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)
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.

The argument of this predicate should be a variant of the argument of the corresponding threaded_call/1 call. When the predicate argument is subsumed by the threaded_call/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 variant threaded_peek/2 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 its type.

Template and modes

threaded_peek(+callable)
threaded_peek(+callable, +nonvar)

Errors

- Goal is a variable:
  instantiation_error
- Goal is neither a variable nor a callable term:
  type_error(callable, Goal)
- 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)
### threaded_wait/1

**Description**

```prolog
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.

**Template and modes**

```prolog
threaded_wait(?term)
threaded_wait(+list(term))
```

**Errors**

*(none)*

**Examples**

Wait until the `data_available` notification is received:

```prolog
threaded_wait(data_available)
```
### 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.

**Template and modes**

```
threaded_notify(@term)
threaded_notify(@list(term))
```

**Errors**

*(none)*

**Examples**

Send the notification `data_available`:

```
threaded_notify(data_available)
```
**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 flags specified in the Logtalk adapter file. 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.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails when errors are found during compilation of a source file.

**Template and modes**

```
logtalk_compile(@source_file_name)
logtalk_compile(@list(source_file_name))
```

**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:
  - 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**

```
| ?- logtalk_compile(set).
| ?- logtalk_load(types(tree)).
| ?- logtalk_compile([listp, list]).
```
logtalk_compile/2

Description

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 consult the User Manual.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails when errors are found during compilation of a source file.

Template and modes

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

?- logtalk_compile_1(list, []).  
?- logtalk_compile_1(types(tree)).  
?- logtalk_compile_1([listp, list], [source_data(off), portability(silent)]).
logtalk_load/1

Description

\[
\begin{align*}
\text{logtalk_load}(\text{File}) \\
\text{logtalk_load}(\text{Files})
\end{align*}
\]

Compiles to disk and then loads to memory a source file or a list of source files using the default compiler flags specified in the Logtalk adapter file. 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.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails when errors are found during compilation of a source file.

Depending on the back-end Prolog compiler, the notation \{File\} may be used in alternative (check the adapter files for its availability).

Template and modes

\[
\begin{align*}
\text{logtalk_load}(\text{@source_file_name}) \\
\text{logtalk_load}(\text{@list(source_file_name)})
\end{align*}
\]

Errors

File is a variable:

\text{instantiation_error}

Files is a variable or a list with an element which is a variable:

\text{instantiation_error}

File, or an element File of the Files list, is neither a variable nor a source file name:

\text{type_error(source_file_name, File)}

File, or an element File of the Files list, uses library notation but the library does not exist:

\text{existence_error(library, Library)}

File or an element File of the Files list, does not exist:

\text{existence_error(file, File)}

Examples

\[
\begin{align*}
| \leftarrow \text{logtalk_load(set)}. \\
| \leftarrow \text{logtalk_load(types(tree))}. \\
| \leftarrow \text{logtalk_load([listp, list])}.
\end{align*}
\]
**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. 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 consult the User Manual. Source file paths can be absolute, relative to the current directory, or use library notation.

Note that only the errors related to problems in the predicate argument are listed below. This predicate fails when errors are found during compilation of a source file.

**Template and modes**

```
logtalk_load(@source_file_name, @list(compiler_flag))
logtalk_load(@list(source_file_name), @list(compiler_flag))
```

**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)`

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
Examples

?- logtalk_load(list, []).
?- logtalk_load(types(tree)).
?- logtalk_load([list, list], [source_data(off), portability(silent)]).
logtalk_make/0

Description

Reloads all Logtalk source files that have been modified since the time they are last loaded. Only source files loaded using the logtalk_load/1-2 predicates are reloaded. Non-modified files will also be reloaded when there is a change to the compilation mode (i.e. when the files were loaded without explicit debug/1 or optimize/1 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).

Template and modes

Errors

(none)

Examples

| ?- logtalk_make.
logtalk_make/1

Description

```
logtalk_make(Target)
```

Allows reloading all Logtalk source files that have been modified since last loaded when called with the target `all` and deleting all intermediate files generated by the compilation of Logtalk source files when called with the target `clean`.

When using the `all` target, only source files loaded using the `logtalk_load/1-2` predicates are reloaded. Non-modified files will also be reloaded when there is a change to the compilation mode (i.e. when the files were loaded without explicit `debug/1` or `optimize/1` 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).

Template and modes

```
logtalk_make(+atom)
```

Errors

*(none)*

Examples

```
| ?- logtalk_make(clean).
```
**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 on the second argument (using the notation `alias(path)`). Paths must always end with the path directory separator character (`/`).

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, 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.

**Template and modes**

```
logtalk_library_path(?atom, -atom)
logtalk_library_path(?atom, -compound)
```
Errors

(no errors)

Examples

?- 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_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`, `entity_prefix`, `entity_type` (returns the value `module` when compiling a module as an object), `source`, `file` (same as `source`), `basename`, `directory`, `stream`, `target` (the full path of the intermediate Prolog file), `term_position` (Start-End), and `variable_names` ([Name1=Variable1, ...]). The `term_position` key is only supported in back-end Prolog compilers that provide access to the start and end lines of a read term.

The `source`, `file`, `basename`, `directory`, and `target` keys can also be used in calls to the `logtalk_load_context/2` predicate wrapped in `initialization/1` directives.

Using the `variable_names` key requires calling the standard built-in predicate `term_variables/2` on the term read and unifying the term variables with the variables in the names list. This, however, rises portability issues with those Prolog compilers that don't return the variables in the same order for the `term_variables/2` predicate and the option `variable_names/1` of the `read_term/3` built-in predicate, which is used by the Logtalk compiler to read source files.

**Template and modes**

`logtalk_load_context(?atom, -nonvar)`

**Errors**

( none )

**Examples**

```
| ?- logtalk_load_context(entity_identifier, Name).
Name = list
yes

| ?- logtalk_load_context(source, Source).
Source = '/Users/me/project/library/list.lgt'

| ?- logtalk_load_context(basename, Basename).
Basename = 'list.lgt'

| ?- logtalk_load_context(directory, Directory).
Directory = '/Users/me/project/library/'
yes
```
**current_logtalk_flag/2**

**Description**

```prolog
current_logtalk_flag(Flag, Value)
```

Enumerates, by backtracking, the current Logtalk flag values.

**Template and modes**

```prolog
current_logtalk_flag(?atom, ?atom)
```

**Errors**

- Flag is neither a variable nor an atom:
  ```prolog
type_error(atom, Flag)
  ```
- Flag is not a valid flag:
  ```prolog
domain_error(flag, Value)
  ```

**Examples**

```prolog
| :- current_logtalk_flag(xml, Value).
```
set_logtalk_flag/2

Description

\hspace{1cm} \texttt{set_logtalk_flag(Flag, Value)}

Sets Logtalk default, global, flag values. For local flag scope, use the corresponding \texttt{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 \texttt{initialization/1} directive.

Template and modes

\hspace{1cm} \texttt{set_logtalk_flag(+atom, +nonvar)}

Errors

- Flag is a variable:
  \hspace{1cm} \texttt{instantiation_error}
- Value is a variable:
  \hspace{1cm} \texttt{instantiation_error}
- Flag is not an atom:
  \hspace{1cm} \texttt{type_error(atom, Flag)}
- Flag is neither a variable nor a valid flag:
  \hspace{1cm} \texttt{domain_error(flag, Flag)}
- Value is not a valid value for flag Flag:
  \hspace{1cm} \texttt{domain_error(flag_value, Flag + Value)}
- Flag is a read-only flag:
  \hspace{1cm} \texttt{permission_error(modify, flag, Flag)}

Examples

\hspace{1cm} \texttt{| ?- set_logtalk_flag(unknown_entities, silent).}

Built-in predicates

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at http://www.renderx.com/
**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 pre-defined flags by using, respectively, the `current_logtalk_flag/2` and `set_logtalk_flag/2` built-in predicates.

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.

**Template and modes**

```
create_logtalk_flag(+atom, +nonvar, +list)
```

**Errors**

- Flag is not a ground term: `instantiation_error`
- Value is not a ground term: `instantiation_error`
- Options is not a ground term: `instantiation_error`
- Flag is not 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 a Type term `type_error(Type, Value)`

**Examples**

```
| ?- create_logtalk_flag(pretty_print_blobs, false, []).
```

108
Built-in methods
**parameter/2**

**Description**

```prolog
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 containing 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. See also `this/1`.

**Template and modes**

```prolog
parameter(+integer, ?term)
```

**Errors**

- Number is a variable:
  ```prolog
  instantiation_error
  ```
- Number is neither a variable nor an integer value:
  ```prolog
  type_error(integer, Number)
  ```
- Number is smaller than one or greater than the parametric entity identifier arity:
  ```prolog
  domain_error(out_of_range, Number)
  ```
- Entity identifier is not a compound term:
  ```prolog
  type_error(compound, Entity)
  ```

**Examples**

```prolog
:- object(box(_Color, _Weight)).
...

color(Color) :-
  parameter(1, Color). % this clause is translated into a fact

heavy :-
  parameter(2, Weight), % after compilation, the >/2 call will be
  Weight > 10. % the first condition on the clause body

...
```
**self/1**

**Description**

```
self(Self)
```

Returns the object that has received the message under processing. This private method is translated to a unification between its argument and the corresponding implicit context argument in the predicate containing the call. This unification occurs at the clause head, not at the clause body.

**Template and modes**

```
self(?object_identifier)
```

**Errors**

(none)

**Examples**

```
test :-
    self(Self), % after compilation, the write/1
    write('executing a method in behalf of '), % call will be the first goal on
    writeq(Self), nl. % the clause body
```
sender/1

Description

sender(Sender)

Returns the object that has 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 containing the call. This unification occurs at the clause head, not at the clause body.

Template and modes

sender(?object_identifier)

Errors

(none)

Examples

% after compilation, the write/1 call will be the first goal on the clause body:

test :-
    sender(Sender),
    write('executing a method to answer a message sent by '),
    writeq(Sender), nl.
**this/1**

**Description**

```
this(This)
```

Unifies its argument with the identifier of the object for which the predicate clause whose body is being executed is defined (or the object importing the category that contains the predicate clause). This private method is implemented as a unification between its argument and the corresponding implicit execution-context argument in the predicate containing the call. This unification occurs at the clause head, not at the clause body. 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. See also `parameter/2`.

**Template and modes**

```
this(?object_identifier)
```

**Errors**

`(none)`

**Examples**

```
% after compilation, the write/1 call will be the first goal on the clause body:

test :-
    this(This),
    write('executing a definition contained in '),
    writeq(This), nl.
```
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 found.

Template and modes

current_op(?operator_priority, ?operator_specifier, ?atom)

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 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)
**current_predicate/1**

**Description**

\[
\text{current_predicate(Predicate)}
\]

Enumerates, by backtracking, the visible user predicates for an object.

**Template and modes**

\[
\text{current_predicate(?predicate_indicator)}
\]

**Errors**

Predicate is neither a variable nor a valid predicate indicator:

\[
\text{type_error(predicate_indicator, Predicate)}
\]

Predicate is a Functor/Arity term but Functor is neither a variable nor an atom:

\[
\text{type_error(atom, Name)}
\]

Predicate is a Functor/Arity term but Arity is neither a variable nor an integer:

\[
\text{type_error(integer, Arity)}
\]

Predicate is a Functor/Arity term but Arity is a negative integer:

\[
\text{domain_error(not_less_than_zero, Arity)}
\]

**Examples**

To enumerate, by backtracking, the user predicates visible in *this*:

\[
\text{current_predicate(Predicate)}
\]

To enumerate, by backtracking, the public and protected user predicates visible in *self*:

\[
::\text{current_predicate(Predicate)}
\]

To enumerate, by backtracking, the public user predicates visible for an explicit object:

\[
\text{Object::current_predicate(Predicate)}
\]
**predicate_property/2**

**Description**

```
predicate_property(Predicate, Property)
```

Enumerates, by backtracking, the properties of a visible predicate. The valid predicate properties are listed in the language grammar.

**Template and modes**

```
predicate_property(+callable, ?predicate_property)
```

**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 predicate visible in `this`:
```
predicate_property(foo(_), Property)
```

To enumerate, by backtracking, the properties of a public or protected predicate visible in `self`:
```
::predicate_property(foo(_), Property)
```

To enumerate, by backtracking, the properties of a public predicate visible in an explicit object:
```
Object::predicate_property(foo(_), Property)
```
**abolish/1**

**Description**

abolish(Predicate)
abolish(Functor/Arity)

Removes a runtime declared dynamic predicate or a local dynamic predicate from an object database.

**Template and modes**

abolish(+predicate_indicator)

**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, Functor/Arity)
Predicate is a private predicate:
permission_error(modify, private_predicate, Functor/Arity)
Predicate is a protected predicate:
permission_error(modify, protected_predicate, Functor/Arity)
Predicate is a static predicate:
permission_error(modify, static_predicate, Functor/Arity)
Predicate is not declared for the object receiving the message:
existence_error(predicate_declaration, Functor/Arity)

**Examples**

To abolish any 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)
asserta/1

Description

\[
\begin{align*}
\text{asserta(Head)} \\
\text{asserta((Head:-Body))}
\end{align*}
\]

Asserts a clause as the first one for an object's dynamic predicate. If the predicate is not already declared, then a dynamic predicate declaration is added to the object (assuming that we are asserting locally or that the compiler flag `dynamic_declarations` was switched on when the object was created or compiled).

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.

Template and modes

\[
\text{asserta(+clause)}
\]

Errors

- Head is a variable:
  - `instantiation_error`
- Head is a 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, Functor/Arity, is that of a private predicate:
  - `permission_error(modify, private_predicate, Functor/Arity)`
- The predicate indicator of Head, Functor/Arity, is that of a protected predicate:
  - `permission_error(modify, protected_predicate, Functor/Arity)`
- The predicate indicator of Head, Functor/Arity, is that of a static predicate:
  - `permission_error(modify, static_predicate, Functor/Arity)`
- Target object was created/compiled with support for dynamic declaration of predicates turned off:
  - `permission_error(create, predicate_declaration, Functor/Arity)`

Examples

To assert a clause as the first one for any dynamic predicate in `this`:

\[
\text{asserta(Clause)}
\]

To assert a clause as the first one for any public or protected dynamic predicate in `self`:

\[
::\text{asserta(Clause)}
\]

To assert a clause as the first one for any public dynamic predicate in an explicit object:

\[
\text{Object::asserta(Clause)}
\]
assertz/1

Description

assertz(Head)
assertz((Head:-Body))

Asserts a clause as the last one for an object's dynamic predicate. If the predicate is not already declared, then a dynamic predicate declaration is added to the object (assuming that we are asserting locally or that the compiler flag dynamic_declarations was switched on when the object was created or compiled).

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.

Template and modes

assertz(+clause)

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, Functor/Arity, is that of a private predicate:
permission_error(modify, private_predicate, Functor/Arity)
The predicate indicator of Head, Functor/Arity, is that of a protected predicate:
permission_error(modify, protected_predicate, Functor/Arity)
The predicate indicator of Head, Functor/Arity, is that of a static predicate:
permission_error(modify, static_predicate, Functor/Arity)
Target object was created/compiled with support for dynamic declaration of predicates turned off:
permission_error(create, predicate_declaration, Functor/Arity)

Examples

To assert a clause as the last one for any 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)
### clause/2

**Description**

```
clause(Head, Body)
```

Enumerates, by backtracking, the clauses of an object's dynamic predicates.

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*.

**Template and modes**

```
clause(+callable, ?body)
```

**Errors**

- Head is a variable:
  - `instantiation_error`
- Head is a neither a variable nor a callable term:
  - `type_error(callable, Head)`
- Body is a neither a variable nor a callable term:
  - `type_error(callable, Body)`
- The predicate indicator of Head, Functor/Arity, is that of a private predicate:
  - `permission_error(access, private_predicate, Functor/Arity)`
- The predicate indicator of Head, Functor/Arity, is that of a protected predicate:
  - `permission_error(access, protected_predicate, Functor/Arity)`
- The predicate indicator of Head, Functor/Arity, is that of a static predicate:
  - `permission_error(access, static_predicate, Functor/Arity)`
- Head is not a declared predicate:
  - `existence_error(predicate_declaration, Functor/Arity)`

**Examples**

- To retrieve a matching clause of any 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)
  ```
**retract/1**

**Description**

| retract(Head)  
| retract((Head:-Body)) |

Retracts a dynamic clause from an object.

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*.

**Template and modes**

| retract(+clause) |

**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, Functor/Arity, is that of a private predicate: 
*permission_error(modify, private_predicate, Functor/Arity)*

The predicate indicator of Head, Functor/Arity, is that of a protected predicate: 
*permission_error(modify, protected_predicate, Functor/Arity)*

The predicate indicator of Head, Functor/Arity, is that of a static predicate: 
*permission_error(modify, static_predicate, Functor/Arity)*

The predicate indicator of Head, Functor/Arity, is not declared: 
*existence_error(predicate_declaration, Functor/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)*
retractall/1

Description

retractall(Head)

Retracts all matching predicates from an object.

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.

Template and modes

retractall(+callable)

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, Functor/Arity, is that of a private predicate:

permission_error(modify, private_predicate, Functor/Arity)

The predicate indicator of Head, Functor/Arity, is that of a protected predicate:

permission_error(modify, protected_predicate, Functor/Arity)

The predicate indicator of Head, Functor/Arity, is that of a static predicate:

permission_error(modify, static_predicate, Functor/Arity)

The predicate indicator of Head, Functor/Arity, is not declared:

existence_error(predicate_declaration, Functor/Arity)

Examples

To retract all matching predicate definitions in this:

retractall(Head)

To retract all matching public or protected predicate definitions in self:

::retractall(Head)

To retract all matching public predicate definitions in an explicit object:

Object::retractall(Head)
**call/1–N**

**Description**

```
call(Goal)
call(Closure, Arg1, ...)
call(Object::Closure, Arg1, ...) 
call(::Closure, Arg1, ...)
```

Calls a goal, which might be 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 back-end Prolog compiler. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object. When using a back-end Prolog compiler supporting a module system, calls in the format `call(Module::Closure, Arg1, ...)` may also be used.

Note that `::Closure` closures are only supported for local meta-calls. Passing this kind of closure as an argument of a meta-predicate called using message sending is not supported and always fails as the value of `self` is lost in the round-trip to the object defining the meta-predicate. The workaround is to simply call the `self(Self)` built-in method and use the returned value with a `Self::Closure` closure instead.

**Template and modes**

```
call(+callable)
call(+callable, ?term)
call(+callable, ?term, ?term)
... 
```

**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, ...) 
```
**ignore/1**

**Description**

ignore(Goal)

This predicate succeeds weather 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.

**Template and modes**

ignore(+callable)

**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 goal as a non-backtracable message to *self*:
  
  ```
  ignore(::Goal)
  ```

- To send a goal as a non-backtraceable message to an explicit object:
  
  ```
  ignore(Object::Goal)
  ```
Once/1

Description

\texttt{once(Goal)}

This predicate behaves as \texttt{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.

Template and modes

\texttt{once(+callable)}

Errors

Goal is a variable:

\texttt{instantiation\_error}

Goal is neither a variable nor a callable term:

\texttt{type\_error(callable, Goal)}

Examples

Call a goal deterministically in the context of the object or category containing the call:

\texttt{once(Goal)}

To send a goal as a non-backtraceable message to \texttt{self}:

\texttt{once(::Goal)}

To send a goal as a non-backtraceable message to an explicit object:

\texttt{once(Object::Goal)}
\+/1

Description

\+ Goal

Not-provable meta-predicate. True iff \texttt{call(Goal)} is false. This built-in meta-predicate cannot be used as a message to an object.

Template and modes

\+ +callable

Errors

Goal is a variable:
\texttt{instantiation\_error}

Goal is neither a variable nor a callable term:
\texttt{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 \textit{self}:
\+ ::Goal

Not-provable goal sent as a message to an explicit object:
\+ Object::Goal
catch/3

Description

\[\text{catch(Goal, Catcher, Recovery)}\]

Catches exceptions thrown by a goal. See the Prolog ISO standard definition. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Template and modes

\[\text{catch(?callable, ?term, ?term)}\]

Errors

- Goal is a variable:
  \[\text{instantiation_error}\]
- Goal is neither a variable nor a callable term:
  \[\text{type_error}(\text{callable, Goal})\]

Examples

(none)
throw/1

Description

\[\text{throw(Exception)}\]

Throws an exception. This built-in predicate is declared as a private method and thus cannot be used as a message to an object.

Template and modes

\[\text{throw(+nonvar)}\]

Errors

- Exception is a variable: \text{instantiation_error}
- Exception does not unify with the second argument of any call of \text{catch/3}: \text{system_error}

Examples

(none)
**bagof/3**

**Description**

\[
\text{bagof}(\text{Term}, \text{Goal}, \text{List})
\]

See the Prolog ISO standard definition. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

**Template and modes**

\[
\text{bagof}(\_\text{term}, +\text{callable}, -\text{list})
\]

**Errors**

(see the Prolog ISO standard)

**Examples**

To find all solutions in the context of the object or category containing the call:

\[
\text{bagof}(\text{Term}, \text{Goal}, \text{List})
\]

To find all solutions by sending the goal as a message to *self*:

\[
\text{bagof}(\text{Term}, ::\text{Goal}, \text{List})
\]

To find all solutions by sending the goal as a message to an explicit object:

\[
\text{bagof}(\text{Term}, \text{Object}:::\text{Goal}, \text{List})
\]
**findall/3**

**Description**

```prolog
findall(Term, Goal, List)
```

See the Prolog ISO standard definition. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

**Template and modes**

```prolog
findall(?term, +callable, ?list)
```

**Errors**

(see the Prolog ISO standard)

**Examples**

To find all solutions in the context of the object or category containing the call:

```prolog
findall(Term, Goal, List)
```

To find all solutions by sending the goal as a message to `self`:

```prolog
findall(Term, ::Goal, List)
```

To find all solutions by sending the goal as a message to an explicit object:

```prolog
findall(Term, Object::Goal, List)
```
**findall/4**

**Description**

```
findall(Term, Goal, List, Tail)
```

Variant of the standard `findall/3` that allows passing the tail of the results list. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

**Template and modes**

```
findall(?term, +callable, ?list, +list)
```

**Errors**

(same as the standard `findall/3` predicate)

**Examples**

To find all solutions in the context of the object or category containing the call:

```
findall(Term, Goal, List, Tail)
```

To find all solutions by sending the goal as a message to `self`:

```
findall(Term, ::Goal, List, Tail)
```

To find all solutions by sending the goal as a message to an explicit object:

```
findall(Term, Object::Goal, List, Tail)
```
forall/2

Description

forall(Generator, Test)

For all solutions of `Generator`, `Test` is true. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Template and modes

forall(+callable, +callable)

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)
**setof/3**

**Description**

```
setof(Term, Goal, List)
```

See the Prolog ISO standard definition. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

**Template and modes**

```
setof(@term, +callable, -list)
```

**Errors**

(see the Prolog ISO standard)

**Examples**

To find all solutions in the context of the object or category containing the call:

```
setof(Term, Goal, List)
```

To find all solutions by sending the goal as a message to `self`:

```
setof(Term, ::Goal, List)
```

To find all solutions by sending the goal as a message to an explicit object:

```
setof(Term, Object::Goal, List)
```
**before/3**

**Description**

```latex
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. Note that you can make its scope protected or private by using, respectively, protected or private implementation of the monitoring protocol.

**Template and modes**

```latex
before(?object_identifier, ?callable, ?object_identifier)
```

**Errors**

*(none)*

**Examples**

```prolog
:- object(...,
    implements(monitoring),
    ...).

    before(Object, Message, Sender) :-
        writeq(Object), write('::'), writeq(Message),
        write(' from '), writeq(Sender), nl.
```
**after/3**

**Description**

User-defined method for handling *after* events. This method is declared in the *monitoring* 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 *monitoring* protocol.

**Template and modes**

```
after(?object_identifier, ?callable, ?object_identifier)
```

**Errors**

*(none)*

**Examples**

```prolog
:- object(...,
    implements(monitoring),
    ...
).

after(Object, Message, Sender) :-
    writeq(Object), write('::'), writeq(Message),
    write(' from '), writeq(Sender), nl.
```
**forward/1**

**Description**

```prolog
forward(Message)
```

User-defined method for forwarding unknown messages sent to an object (using the `::/2` control construct), automatically called 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.

**Template and modes**

```prolog
forward(+callable)
```

**Errors**

```prolog
(none)
```

**Examples**

```prolog
:- object(proxy,
    implements(forwarding),
    ...).

    forward(Message) :-
    % delegate the unknown message to other object
    [real::Message].
```
**call//1-N**

**Description**

This non-terminal takes a closure and is processed by appending the input list of tokens and the list of remaining tokens 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. When using a back-end Prolog compiler supporting a module system, calls in the format `call(Module::Closure)` may also be used. By using as argument a lambda expression, this built-in non-terminal provides controlled access to the input list of tokens and to the list of the remaining tokens processed by the grammar rule containing the call.

**Template and modes**

- `call(+callable)`
- `call(+callable, ?term)`
- `call(+callable, ?term, ?term)`
- ...

**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 input list of tokens and the list of remaining tokens to the arguments of the closure, in the context of the object or category containing the call:

- `call(Closure)`

To send a goal, constructed by appending the input list of tokens and the list of remaining tokens to the arguments of the closure, as a message to `self`:

- `call(::Closure)`

To send a goal, constructed by appending the input list of tokens and the list of remaining tokens to the arguments of the closure, as a message to an explicit object:

- `call(Object::Closure)`
**phrase/**1

**Description**

| phrase(NonTerminal) |

This non-terminal takes a non-terminal or a grammar rule body and parses it using the current implicit list of tokens. A common use is to wrap what otherwise would be a naked variable in a grammar rule body.

**Template and modes**

| phrase(+callable) |

**Errors**

- NonTerminal is a variable:
  
  instantiation_error

- NonTerminal is neither a variable nor a callable term:

  type_error(callable, NonTerminal)

**Examples**
phrase/2

Description

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 back-end 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.

Template and modes

phrase(+callable, ?list)

Errors

NonTerminal is a variable:

instantiation_error

NonTerminal is neither a variable nor a callable term:

type_error(callable, NonTerminal)

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)
phrase/3

Description

\[
\text{phrase}(\text{GrammarRuleBody}, \text{Input}, \text{Rest}) \\
\text{phrase}(::\text{GrammarRuleBody}, \text{Input}, \text{Rest}) \\
\text{phrase}(\text{Object}::\text{GrammarRuleBody}, \text{Input}, \text{Rest})
\]

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 back-end Prolog compiler supporting a module system, calls in the format \text{phrase}(Module:GrammarRuleBody, \text{Input}, \text{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.

Template and modes

\[
\text{phrase}(+\text{callable}, ?\text{list}, ?\text{list})
\]

Errors

NonTerminal is a variable:

instantiation_error

NonTerminal is neither a variable nor a callable term:

type_error(callable, NonTerminal)

Examples

To parse a list of tokens using a local non-terminal:

\[\text{phrase}(\text{NonTerminal}, \text{Input}, \text{Rest})\]

To parse a list of tokens using a non-terminal within the scope of self:

\[\text{phrase}(::\text{NonTerminal}, \text{Input}, \text{Rest})\]

To parse a list of tokens using a public non-terminal of an explicit object:

\[\text{phrase}(\text{Object}::\text{NonTerminal}, \text{Input}, \text{Rest})\]
**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 predicate `term_expansion/2`.

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 clauses for the `term_expansion/2` predicate are 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.

Term expansion is only performed by Logtalk at compile time (to expand terms read from a source file). This predicate can be used by the user to perform term expansion at runtime (for example, to convert a grammar rule into a clause).

**Template and modes**

```
expand_term(?term, ?term)
```

**Errors**

*(none)*

**Examples**

*(none)*
term_expansion/2

Description

term_expansion(Term, Expansion)

Defines an expansion for a term. This predicate, when defined, is automatically called by the expand_term/2 method. Use of this predicate by the expand_term/2 method may be restricted by changing its default public scope. The term_expansion/2 clauses are only used by the expand_term/2 method if they are within the scope of the sender. When that is not the case, the expand_term/2 method only uses the default expansions. The term_expansion/2 predicate may return a list of terms.

Term expansion may be 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, 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. This protocol implementation relation can be declared as either protected or private to restrict the scope of this predicate.

Template and modes

term_expansion(+nonvar, -nonvar)
term_expansion(+nonvar, -list(nonvar))

Errors

(none)

Examples

term_expansion((:- license(default)), (:- license(gplv3))).
term_expansion(data(Millimeters), data(Meters)) :- Meters is Millimeters / 1000.
**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 clauses for the `goal_expansion/2` predicate are 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 call to the `goal_expansion/2` predicate fails, the two arguments are unified.

Goal expansion is only performed by Logtalk at compile time (to expand the body of clauses and meta-directives read from a source file). This predicate can be used by the user to perform goal expansion at runtime (for example, before asserting a clause).

**Template and modes**

```
expand_goal(?term, ?term)
```

**Errors**

(none)

**Examples**

(none)
**goal_expansion/2**

**Description**

```prolog
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 there are no changes. Thus, care must be taken to avoid compilation loops. This predicate, when defined, 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, 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 protocol. This protocol implementation relation can be declared as either protected or private to restrict the scope of this predicate.

**Template and modes**

```prolog
goal_expansion(+callable, -callable)
```

**Errors**

*(none)*

**Examples**

```prolog
goal_expansion(write(Term), (write_term(Term, []), nl)).
goal_expansion(read(Term), (write('Input: '), {read(Term)})).
```
print_message/3

Description

\texttt{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 \texttt{logtalk::message_tokens(Term, Component)} hook predicate. This method is declared in the \texttt{logtalk} built-in object as a public predicate. The line prefix and the output stream used for each \texttt{Kind-Component} pair can be found using the \texttt{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 \texttt{logtalk::message_hook(Message, Kind, Component, Tokens)} hook predicate. If this predicate succeeds, the \texttt{print_message/3} predicate assumes that the message have been successfully printed.

Template and modes

\texttt{print_message(+nonvar, +atom, +nonvar)}

Errors

\texttt{(none)}

Examples

\texttt{..., logtalk::print_message(information, core, redefining_entity(object, foo)), ...}
**message_tokens//2**

**Description**

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.

**Template and modes**

```
message_tokens(+nonvar, +atom)
```

**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].
```
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 have been successfully printed.

Template and modes

`message_hook(@callable, @callable, @callable, @list(nonvar))`

Errors

(none)

Examples

```prolog
:- 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).
```

Built-in methods

XML to PDF by RenderX XEP XSL-FO Formatter, visit us at [http://www.renderx.com/](http://www.renderx.com/)
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.

Template and modes

message_prefix_stream(?nonvar, ?atom, ?atom, ?stream_or_alias)

Errors

(none)

Examples

:- multifile(logtalk::message_prefix_stream/4).
:- dynamic(logtalk::message_prefix_stream/4).

logtalk::message_prefix_stream(information, core, '% ', user_output).
print_message_tokens/3

Description

\texttt{print_message_tokens(Stream, Prefix, Tokens)}

Built-in method for printing a list of message tokens, declared in the \texttt{logtalk} built-in object as a public predicate. This method is automatically called by the \texttt{print_message/3} method (assuming that the message was not intercepted by a \texttt{message_hook/4} definition) and calls the user-defined hook predicate \texttt{print_message_token/4} for each token. When a call to this hook predicate succeeds, the \texttt{print_message_tokens/3} predicate assumes that the token have been printed. When the call fails, the \texttt{print_message_tokens/3} predicate uses a default printing procedure for the token.

Template and modes

\texttt{print_message_tokens(@stream_or_alias, +atom, @list(nonvar))}

Errors

\texttt{(none)}

Examples

\ldots, \texttt{logtalk::print_message_tokens(user_output, '% ', ['Redefining \texttt{-w -q'}-[object, foo], nl]),}
**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.

**Template and modes**

```
print_message_token(@stream_or_alias, @atom, @nonvar, @list(nonvar))
```

**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).
```
**ask_question/5**

**Description**

```prolog
ask_question(Question, Kind, Component, 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 checking predicate succeeds.

This predicate starts by calling the `logtalk::question_hook(Question, Kind, Component, Check, Answer)` hook predicate. If this predicate succeeds, the `ask_question/5` predicate assumes that the question have been successfully asked and replied.

**Template and modes**

```prolog
ask_question(+nonvar, +atom, +nonvar, +callable, -term)
```

**Meta-predicate template**

```prolog
ask_question(*, *, *, 1, *)
```

**Errors**

*(none)*

**Examples**

```prolog
..., logtalk::ask_question(enter_context_spy_point(Template), question, debugger, callable, SpyPoint), ...
```
question_hook/5

Description

question_hook(Question, Kind, Component, 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.

Template and modes

question_hook(+nonvar, +nonvar, +atom, +callable, -term)

Meta-predicate template

question_hook(*, *, *, 1, *)

Errors

(none)

Examples

:- multifile(logtalk::question_hook/5).
:- dynamic(logtalk::question_hook/5).

% use a pre-defined answer instead of asking the user
logtalk::question_hook(upper_limit, question, my_app, float, 3.7).
**question_prompt_stream/4**

**Description**

```prolog
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.

**Template and modes**

```prolog
question_prompt_stream(?nonvar, ?atom, ?atom, ?stream_or_alias)
```

**Errors**

*(none)*

**Examples**

```prolog
:- multifile(logtalk::question_prompt_stream/4).
:- dynamic(logtalk::question_prompt_stream/4).
logtalk::question_prompt_stream(question, debugger, '    > ', user_input).
```
**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 predicate. The hook predicates are called with the head of the coinductive predicate on coinductive success and, optionally, with the hypothesis used that 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.

**Template and modes**

```
coinductive_success_hook(+callable, +callable)
coinductive_success_hook(+callable)
```

**Errors**

*(none)*

**Examples**

*(none)*
Control constructs
**Description**

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).

The `{Proxy}::Message` syntax allows simplified access to parametric object *proxies*. Its operational semantics is equivalent to the goal conjunction `(call(Proxy), Proxy::Message)`. 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.

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 in subsequent messages.

**Template and modes**

```
+object_identifier::+callable
{+object_identifier}::+callable
```

**Errors**

Either `Object` or `Message` is a variable:

- `instantiation_error`

Object is not 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 `Functor/Arity`, is declared private:

- `permission_error(access, private_predicate, Functor/Arity)`

Message, with predicate indicator `Functor/Arity`, is declared protected:

- `permission_error(access, protected_predicate, Functor/Arity)`

Message, with predicate indicator `Functor/Arity`, is not declared:

- `existence_error(predicate_declaration, Functor/Arity)`

Object does not exist:

- `existence_error(object, Object)`

Proxy is a variable:

- `instantiation_error`

Proxy is not a valid object identifier:

- `type_error(object_identifier, Proxy)`
The predicate Proxy does not exist in the user pseudo-object:

\texttt{existence\_error(procedure, ProxyFunctor/ProxyArity)}

**Examples**

\begin{verbatim}
| ?- list::member(X, [1, 2, 3]).
  X = 1 ;
  X = 2 ;
  X = 3
  yes
\end{verbatim}


::/1

**Description**

::Message

Send a message to *self*. Only 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).

The lookups for the message declaration and the corresponding method are performed using a depth-first strategy. A message to *self* necessarily requires the use of dynamic binding but a caching mechanism is used to improve performance in subsequent messages.

**Template and modes**

::+callable

**Errors**

- Message is a variable:  
  instantiation_error
- Message is neither a variable nor a callable term:  
  type_error(callable, Message)
- Message, with predicate indicator Functor/Arity, is declared private:  
  permission_error(access, private_predicate, Functor/Arity)
- Message, with predicate indicator Functor/Arity, is not declared:  
  existence_error(predicate_declaration, Functor/Arity)

**Examples**

area(Area) :-  
  ::width(Width),  
  ::height(Height),  
  Area is Width*Height.
^/^1

Description

Calls an imported or inherited predicate definition. The call fails if the predicate is declared but there is no imported and
no 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. When used within a category, 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. 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.

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 whenever sufficient information is
available. When the lookups are performed at runtime, a caching mechanism is used to improve performance in subsequent
calls.

Template and modes

^++callable

Errors

Predicate is a variable:
  instantiation_error
Predicate is neither a variable nor a callable term:
  type_error(callable, Predicate)
Predicate, with predicate indicator Functor/Arity, is declared private:
  permission_error(access, private_predicate, Functor/Arity)
Predicate, with predicate indicator Functor/Arity, is not declared:
  existence_error(predicate_declaration, Functor/Arity)

Examples

init :-
  assertz(counter(0)),
  ^^init.
[]/1

Description

This control construct allows the programmer to send a message to an object while preserving the original sender. 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 using of this control construct to break object encapsulation, an attempt to delegate a message to same object as 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.

Template and modes

Errors

Object and the original sender are the same object:

`permission_error(access, object, Sender)`

Either Object or Message is a variable:

`instantiation_error`

Object is not 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 Functor/Arity, is declared private:

`permission_error(access, private_predicate, Functor/Arity)`

Message, with predicate indicator Functor/Arity, is declared protected:

`permission_error(access, protected_predicate, Functor/Arity)`

Message, with predicate indicator Functor/Arity, is not declared:

`existence_error(predicate_declaration, Functor/Arity)`

Object does not exist:

`existence_error(object, Object)`

Proxy is a variable:

`instantiation_error`

Proxy is not a valid object identifier:

`type_error(object_identifier, Proxy)`

The predicate Proxy does not exist in the user pseudo-object:

`existence_error(procedure, ProxyFunctor/ProxyArity)`
Examples

```
forward(Message) :-
    [backup::Message].
```
{}/1

Description

{Term}
{Goal}

This control construct allows the programmer to bypass the Logtalk compiler. It can be used to wrap a source file term (either a clause or a directive) to bypass the term-expansion mechanism. Similarly, it can be used to wrap a goal to bypass the goal-expansion mechanism. When used to wrap a goal, it is opaque to cuts and the argument is called within the context of the pseudo-object user. It is also possible to use (Closure) as the first argument of call/2-N calls. In this case, Closure will be extended with the remaining arguments of the call/2-N call in order to construct a goal that will be called within the context of user. It can also be used as a message to any object. This is useful when the message is e.g. a conjunction of messages, some of which being calls to Prolog built-in predicates.

This control construct may also be used in place of an object identifier when sending a message. In this case, the result of proving its argument as a goal (within the context of the pseudo-object user) is used as an object identifier in the message sending call. This feature is mainly used with parametric objects when the their identifiers correspond to predicates defined in user.

Template and modes

{+callable}

Errors

Term or Goal is a variable:
  instantiation_error
Term or Goal is neither a variable nor a callable term:
  type_error(callable, Goal)

Examples

{:- load_foreign_resource(file)}.

N1/D1 < N2/D2 :-
  {N1*D2 < N2*D1}.

call_in_user(F, X, Y, Z) :-
 call({F}, X, Y, Z).

| ?- {circle(Id, Radius, Color)}::area(Area).
  ...|
| ?- logtalk::{write('Hello world!'), nl}. |
Hello world!
yes
Calls a goal within the context of the specified object. Goal is called with the execution context (sender, this, and self) set to Object. Goal may need to be written within brackets to avoid parsing errors due to operator clashes. 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 compiler flag `context_switching_calls` set to `allow`. Set this compiler flag to `deny` to disable this control construct and thus preventing using it to break encapsulation.

The `{Proxy}<<Message` syntax allows simplified access to parametric object proxies. Its operational semantics is equivalent to the goal conjunction `(call(Proxy), Proxy<<Message)`. 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 of the necessary information for the second compiler pass may not be available at runtime.

**Template and modes**

```
+object_identifier<<+callable
(+object_identifier)<<+callable
```

**Errors**

- Either Object or Goal is a variable:
  `instantiation_error`
- Object is neither a variable nor a valid object identifier:
  `type_error(object_identifier, Object)`
- Goal is neither a variable nor a callable term:
  `type_error(callable, Goal)`
- 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)`

**Examples**

```
test(member) :-
  list << member(1, [1]).
```