

Logtalk 2.40.1

Reference Manual

Copyright © Paulo Moura

pmoura@logtalk.org

<http://logtalk.org/>

Last updated on May 23, 2010

Reference Manual

Grammar

- Compilation units. 1
 - Object definition. 1
 - Category definition. 2
 - Protocol definition. 2
- Entity relations. 3
 - Implemented protocols. 3
 - Extended protocols. 3
 - Imported categories. 4
 - Extended objects. 4
 - Extended categories. 4
 - Instantiated objects. 5
 - Specialized objects. 5
 - Complemented objects. 5
 - Entity scope. 6
- Entity identifiers. 6
 - Object identifiers. 6
 - Category identifiers. 6
 - Protocol identifiers. 7
- Source file names. 7
- Directives. 8
 - Source file directives. 8
 - Conditional compilation directives. 8
 - Object directives. 8
 - Category directives. 8
 - Protocol directives. 9

Predicate directives.....	9
Clauses and goals.....	12
Lambda expressions.....	13
Entity properties.....	14
Predicate properties.....	14

Directives

Source file directives

encoding/1.....	16
initialization/1.....	30
op/3.....	48
set_logtalk_flag/2.....	17

Conditional compilation directives

if/1.....	18
elif/1.....	19
else/0.....	20
endif/0.....	21

Entity directives

calls/1.....	22
category/1-3.....	23
dynamic/0.....	25
end_category/0.....	26
end_object/0.....	27
end_protocol/0.....	28
info/1.....	29
initialization/1.....	30
object/1-5.....	32
protocol/1-2.....	38
synchronized/0.....	39
threaded/0.....	40
uses/1.....	41

Predicate directives

alias/3.....	42
discontiguous/1.....	43
dynamic/1.....	44
info/2.....	45
meta_predicate/1.....	46
mode/2.....	47
multifile/1.....	31
op/3.....	48



private/1..... 49
 protected/1..... 50
 public/1..... 51
 synchronized/1..... 52
 uses/2..... 53

Built-in predicates

Enumerating objects, categories and protocols

current_category/1..... 56
 current_object/1..... 57
 current_protocol/1..... 58

Enumerating objects, categories and protocols properties

category_property/2..... 59
 object_property/2..... 60
 protocol_property/2..... 61

Creating new objects, categories and protocols

create_category/4..... 62
 create_object/4..... 63
 create_protocol/3..... 65

Abolishing objects, categories and protocols

abolish_category/1..... 66
 abolish_object/1..... 67
 abolish_protocol/1..... 68

Objects, categories and protocols relations

extends_object/2-3..... 69
 extends_protocol/2-3..... 70
 extends_category/2-3..... 71
 implements_protocol/2-3..... 72
 imports_category/2-3..... 73
 instantiates_class/2-3..... 74
 specializes_class/2-3..... 75
 complements_object/2..... 76

Event handling

abolish_events/5..... 77
 current_event/5..... 78
 define_events/5..... 79

Multi-threading meta-predicates

threaded/1.....	80
threaded_call/1-2.....	81
threaded_once/1-2.....	82
threaded_ignore/1.....	83
threaded_exit/1-2.....	84
threaded_peek/1-2.....	85
threaded_wait/1.....	86
threaded_notify/1.....	87

Compiling and loading objects, categories and protocols

logtalk_compile/1.....	88
logtalk_compile/2.....	89
logtalk_load/1.....	91
logtalk_load/2.....	92
logtalk_library_path/2.....	94

Flags

current_logtalk_flag/2.....	95
set_logtalk_flag/2.....	96

Others

forall/2.....	97
retractall/1.....	98

Built-in methods

Execution context methods

parameter/2.....	100
self/1.....	101
sender/1.....	102
this/1.....	103

Reflection methods

current_predicate/1.....	104
predicate_property/2.....	105

Database methods

abolish/1.....	106
asserta/1.....	107
assertz/1.....	108
clause/2.....	109
retract/1.....	110



retractall/1.....	111
Meta-call methods	
call/1-N.....	112
once/1.....	113
\+/1.....	114
Exception-handling methods	
catch/3.....	115
throw/1.....	116
All solutions methods	
bagof/3.....	117
findall/3.....	118
forall/2.....	119
setof/3.....	120
Event handler methods	
before/3.....	121
after/3.....	122
DCGs rules parsing methods and non-terminals	
call//1.....	123
phrase/2.....	124
phrase/3.....	125
Term and goal expansion methods	
expand_term/2.....	126
term_expansion/2.....	127
expand_goal/2.....	128
goal_expansion/2.....	129
Control constructs	
Message sending	
::/2.....	132
::/1.....	134
^^/1.....	135
Calling external code	
{}/1.....	136
Context-switching calls	
<</2.....	137

Direct calls of imported predicates

:/1.....	138
----------	-----

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 "".

Compilation units

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

Object definition

```
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 |
    imports_categories |
```

instantiates_classes |
specializes_classes

Category definition

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

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

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

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

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

Protocol definition

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

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

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 |
```

object_identifier ", " complemented_object_sequence

complemented_object_list ::=
"[" complemented_object_sequence "]"

Entity scope

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

Entity identifiers

entity_identifiers ::=
entity_identifier |
entity_identifier_sequence |
entity_identifier_list

entity_identifier ::=
object_identifier |
protocol_identifier |
category_identifier

entity_identifier_sequence ::=
entity_identifier |
entity_identifier ", " entity_identifier_sequence

entity_identifier_list ::=
"[" entity_identifier_sequence "]"

Object identifiers

object_identifiers ::=
object_identifier |
object_identifier_sequence |
object_identifier_list

object_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 "]"
```

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 "). |
  initialization_directive |
  operator_directive
```

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 |
  threaded. |
  synchronized. |
  dynamic. |
  uses(" object_identifier "). |
  calls(" protocol_identifiers "). |
  info(" info_list "). |
  predicate_directives
```

Category directives

```
category_directives ::=
  category_directive |
  category_directive category_directives
```

```
category_directive ::=
  initialization_directive |
  synchronized. |
  dynamic. |
```

```

":- uses(" object_identifier ")." |
":- calls(" protocol_identifiers ")." |
":- info(" info_list ")." |
predicate_directives

```

Protocol directives

```

protocol_directives ::=
  protocol_directive |
  protocol_directive protocol_directives

```

```

protocol_directive ::=
  initialization_directive |
  ":- dynamic." |
  ":- info(" info_list ")." |
  predicate_directives

```

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 |
  info_directive |
  dynamic_directive |
  discontinuous_directive |
  multifile_directive |
  operator_directive

```

```

alias_directive ::=
  ":- alias(" entity_identifer ", " predicate_indicator ", " predicate_indicator ")." |
  ":- alias(" entity_identifer ", " non_terminal_indicator ", " non_terminal_indicator ")."

```

```

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

```

```

uses_directive ::=
  ":- uses(" object_identifer ", " 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_solutions ")."

meta_predicate_directive ::=
  ":- meta_predicate(" meta_predicate_mode_indicator ")."

info_directive ::=
  ":- info(" predicate_indicator | non_terminal_indicator ", " info_list ")."

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

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

multifile_directive ::=
  ":- multifile(" predicate_indicator_term ")." |

predicate_indicator_term ::=
  predicate_indicator |
  predicate_indicator_sequence |
  predicate_indicator_list

predicate_indicator_sequence ::=
  predicate_indicator |
  predicate_indicator " ," predicate_indicator_sequence

predicate_indicator_list ::=
  "[" predicate_indicator_sequence "]"

predicate_indicator_alias ::=
  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 "]"

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 ":" 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 "]"

predicate_mode_term ::=
    atom "(" mode_terms ")"

non_terminal_mode_term ::=
    atom "(" mode_terms ")"

mode_terms ::=
    mode_term |
    mode_term ", " mode_terms

mode_term ::=
    "@" [type] | "+" [type] | "-" [type] | "?" [type]

type ::=
    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_solutions ::=
    "zero" | "zero_or_one" | "zero_or_more" | "one" | "one_or_more" | "error"

meta_predicate_mode_indicator ::=
    atom "(" meta_predicate_terms ")"

meta_predicate_terms ::=
    meta_predicate_term |
    meta_predicate_term "," meta_predicate_terms

meta_predicate_term ::=
    ":" | "*" | integer

info_list ::=
    "]" |
    "[" info_item "is" nonvar "]" info_list "]"

info_item ::=
    "comment" | "remarks" |
    "author" | "version" | "date" |
    "copyright" | "license" |
    "parameters" | "parnames" |
    "arguments" | "argnames" |
    "definition" | "redefinition" | "allocation" |
    "examples" | "exceptions" |
    atom
```

Clauses and goals

```
clause ::=
    object_identifier ":" head ":" body |
    head ":" body |
    fact

goal ::=
    message_call |
    external_call |
    direct_call |
    context_call |
    callable

message_call ::=
    message_to_object |
    message_to_self |
```

```

message_to_super

message_to_object ::=
  receiver ":" messages

message_to_self ::=
  ":" messages

message_to_super ::=
  "^" message

messages ::=
  message |
  "(" message "," messages ")" |
  "(" message ";" messages ")" |
  "(" message "->" messages ")"

message ::=
  callable |
  variable

receiver ::=
  "{" callable "}" |
  object_identifier |
  variable

external_call ::=
  "{" callable "}"

direct_call ::=
  ":" message

context_call ::=
  object_identifier "<<" goal

```

Lambda expressions

```

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

lambda_free_variables ::=
  "{" term "}" |
  "{}"

lambda_parameters ::=
  list_of_terms |
  "["

```

Entity properties

```
category_property ::=  
  "static" |  
  "dynamic" |  
  "built_in" |  
  "synchronized" |  
  "file(File, Path)" |  
  "lines(Start, End)" |  
  "events"
```

```
object_property ::=  
  "static" |  
  "dynamic" |  
  "built_in" |  
  "synchronized" |  
  "threaded" |  
  "file(File, Path)" |  
  "lines(Start, End)" |  
  "context_switching_calls" |  
  "dynamic_declarations" |  
  "events" |  
  "complements"
```

```
protocol_property ::=  
  "static" |  
  "dynamic" |  
  "built_in" |  
  "file(File, Path)" |  
  "lines(Start, End)"
```

Predicate properties

```
predicate_property ::=  
  "static" | "dynamic" |  
  "private" | "protected" | "public" |  
  "logtalk" | "prolog" |  
  "multifile" |  
  "synchronized" |  
  "built_in" |  
  "declared_in(" entity_identifier ")" |  
  "defined_in(" object_identifier | category_identifier ")" |  
  "meta_predicate(" meta_predicate_mode_indicator ")" |  
  "alias_of(" callable ")" |  
  "non_terminal(" non_terminal_indicator ")"
```

Directives

encoding/1

Description

```
encoding(Encoding)
```

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. 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 config file must define a table that translates between the Logtalk and Prolog-specific terms 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

```
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 `set_logtalk_flag/2` built-in predicate within an `initialization/1` directive.

Template and modes

```
set_logtalk_flag(+atom, +atom)
```

Errors

Flag is a variable:

```
in instantiation_error
```

Value is a variable:

```
in instantiation_error
```

Flag is not an atom:

```
type_error(atom, Flag)
```

Flag is neither a variable nor a valid flag:

```
domain_error(logtalk_flag, Flag)
```

Value is not a valid value for flag Flag:

```
domain_error(flag_value, Flag + Value)
```

Flag is a read-only flag:

```
permission_error(modify, flag, Flag)
```

Examples

```
:- set_logtalk_flag(xmldocs, on).
```

if/1

Description

```
if(Goal)
```

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

```
if(+callable)
```

Examples

```
:- if(current_prolog_flag(double_quotes, atom)).
```

elif1/1

Description

```
elif1(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

```
elif(+callable)
```

Examples

```
:- elif(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.
```

calls/1

Description

```
calls(Protocol)
calls(Protocol1, Protocol2, ...)
calls([Protocol1, Protocol2, ...])
```

Declares the protocol(s) that are called by predicates defined in an object or category.

Template and modes

```
calls(+protocol_identifiers)
```

Examples

```
:- calls(comparingp).
```

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

```
:- category(monitring).

:- category(monitring,
  implements(monitringp)).

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

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

:- category(logging,
  implements(monitring),
  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

```
:- 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.
```

info/1

Description

```
info(List)
```

Documentation directive for objects, protocols, and categories. The directive argument is a list of pairs using the format *Key is Value*.

Template and modes

```
info(+info_list)
```

Examples

```
:- info([
    version is 1.0,
    author is 'Paulo Moura',
    date is 2000/4/20,
    comment is 'List protocol.']).
```

initialization/1

Description

```
initialization(Goal)
```

When used within an entity (object, category, or protocol), this directive defines a goal to be called immediately after the container entity 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(@goal)
```

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.

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)

```
object(Object)

object(Object,
  implements(Protocols))

object(Object,
  imports(Categories))

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

Prototype extensions

```
object(Object,
  extends(Objects))

object(Object,
  implements(Protocols),
  extends(Objects))

object(Object,
  imports(Categories),
  extends(Objects))

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

Class instances

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

Classes

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

Classes with metaclasses

```
object(Object ,
  instantiates(Classes) ,
  specializes(Classes))

object(Object ,
  implements(Protocols) ,
  instantiates(Classes) ,
  specializes(Classes))

object(Object ,
  imports(Categories) ,
  instantiates(Classes) ,
  specializes(Classes))

object(Object ,
  implements(Protocols) ,
  imports(Categories) ,
  instantiates(Classes) ,
  specializes(Classes))
```

Starting object directive.

Template and modes

Stand-alone objects (prototypes)

```
object(+object_identifier)

object(+object_identifier ,
  implements(+implemented_protocols))

object(+object_identifier ,
  imports(+imported_categories))

object(+object_identifier ,
  implements(+implemented_protocols) ,
  imports(+imported_categories))
```

Prototype extensions

```
object(+object_identifier,  
      extends(+extended_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      extends(+extended_objects))  
  
object(+object_identifier,  
      imports(+imported_categories),  
      extends(+extended_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      imports(+imported_categories),  
      extends(+extended_objects))
```

Class instances

```
object(+object_identifier,  
      instantiates(+instantiated_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      instantiates(+instantiated_objects))  
  
object(+object_identifier,  
      imports(+imported_categories),  
      instantiates(+instantiated_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      imports(+imported_categories),  
      instantiates(+instantiated_objects))
```

Classes

```
object(+object_identifier,  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      imports(+imported_categories),  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      imports(+imported_categories),  
      specializes(+specialized_objects))
```

Class with metaclasses

```
object(+object_identifier,  
      instantiates(+instantiated_objects),  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      instantiates(+instantiated_objects),  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      imports(+imported_categories),  
      instantiates(+instantiated_objects),  
      specializes(+specialized_objects))  
  
object(+object_identifier,  
      implements(+implemented_protocols),  
      imports(+imported_categories),  
      instantiates(+instantiated_objects),  
      specializes(+specialized_objects))
```

Examples

```
:- object(list).

:- object(list,
    implements(listp)).

:- object(list,
    extends(compound)).

:- object(list,
    implements(listp),
    extends(compound)).

:- object(object,
    imports(initialization),
    instantiates(class)).

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

:- object(agent,
    imports(private::attributes)).
```

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)).
```

synchronized/0

Description

```
synchronized
```

Declares that all object (or category) predicates and non-terminals will be synchronized (i.e. all predicates and non-terminals will use the same mutex for thread synchronization). Synchronized predicates and non-terminals are silently compiled as normal predicates and normal non-terminals when using back-end Prolog compilers that don't support multi-threading programming.

Template and modes

```
synchronized
```

Examples

```
:- synchronized.
```

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.
```

uses/1

Description

```
uses(Object)
```

Declares an object that receives messages from predicates defined in the category or object containing the directive. In the current Logtalk version, this is a non-operational directive, which may be used for documentation. Nevertheless, the compiler will warn you of any unknown object referenced using this directive when an entity containing it is compiled. In a forthcoming release, this directive will likely be used to support object namespaces.

Template and modes

```
uses(+object_identifier)
```

Examples

```
:- uses(list).
```

alias/3

Description

```
alias(Entity, Predicate, Alias)
alias(Entity, NonTerminal, Alias)
```

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. This directive may be used in objects, protocols, and categories.

Template and modes

```
alias(@entity_identifier, +predicate_indicator, +predicate_indicator)
alias(@entity_identifier, +non_terminal_indicator, +non_terminal_indicator)
```

Examples

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

:- alias(words, singular//0, peculiar//0).
```

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

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

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

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

dynamic/1

Description

```
dynamic(Functor/Arity)
dynamic((Functor1/Arity1, Functor2/Arity2, ...))
dynamic([Functor1/Arity1, Functor2/Arity2, ...])

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

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

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

Declares dynamic predicates and dynamic grammar rule non-terminals. Note that an object can be static and have both static and dynamic predicates/non-terminals. Dynamic predicates cannot be declared as synchronized. When the dynamic predicates are local to an object, declaring them also as private predicates allows the Logtalk compiler to generate optimized code for asserting and retracting predicate clauses. Categories can also contain dynamic predicate directives but cannot contain clauses for dynamic predicates.

The predicate indicators (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

```
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*.

Template and modes

```
info(+predicate_indicator, +info_list)
info(+non_terminal_indicator, +info_list)
```

Examples

```
:- info(empty/1, [
    comment is 'True if the argument is an empty list.',
    argnames is ['List']]).

:- info(sentence//0, [
    comment is 'Rewrites a sentence into a noun phrase and a verb phrase.']).
```

meta_predicate/1

Description

```
meta_predicate(MetaPredicate)
```

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 `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 context of the *sender*, i.e. in the meta-predicate calling context, not in the meta-predicate definition context.

Template and modes

```
meta_predicate(+meta_predicate_mode_indicator)
```

Examples

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

mode/2

Description

```
mode(Mode, Number_of_solutions)
```

Most predicates can be used with several instantiations modes. This directive enables the specification of each instantiation mode and the corresponding number of solutions (not necessarily distinct). 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_solutions)  
mode(+non_terminal_mode_term, +number_of_solutions)
```

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(+string, -list(atom)), zero_or_one).
```

op/3

Description

```
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

```
op(+integer, +associativity, +atom_or_atom_list)
```

Examples

```
:- op(950, fx, +).  
:- op(950, fx, ?).  
:- op(950, fx, @).  
:- op(950, fx, -).
```

private/1

Description

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

private(Functor//Arity)
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

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

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

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

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

:- protected((print/2, convert/4)).

:- protected([load/1, save/3]).
```

public/1

Description

```
public(Functor/Arity)
public((Functor1/Arity1, Functor2/Arity2, ...))
public([Functor1/Arity1, Functor2/Arity2, ...])

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

public(op(Precedence, Associativity, Operator))
```

Declares public predicates, public grammar rule non-terminals, and public operators. A public predicate can be called from any object. A public non-terminal can be used as an argument in `phrase/2` and `phrase/3` 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

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

synchronized(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.

Synchronized predicates are silently compiled as normal predicates when using back-end Prolog compilers that don't support multi-threading programming. 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

```
uses(Object, Predicates)
uses(Object, PredicatesAndAliases)
```

```
uses(Object, NonTerminals)
uses(Object, NonTerminalsAndAliases)
```

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 config files).

It is possible to specify a predicate alias using the notation `Functor/Arity::Alias/Arity`. Aliases may be used either for avoiding conflicts between predicates specified in several `uses/2` directives or for giving more meaningful names considering the using context of the predicates.

It is also possible to include operator declarations, `op(Precedence, Associativity, Operator)`, in the second argument.

Template and modes

```
uses(+object_identifier, +predicate_indicator_list)
uses(+object_identifier, +predicate_indicator_alias_list)
```

```
uses(+object_identifier, +non_terminal_indicator_list)
uses(+object_identifier, +non_terminal_indicator_alias_list)
```

Examples

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

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)
    ...
```

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(monitring).
```

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.

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 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.

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 valid object property:

```
domain_error(object_property, Property)
```

Examples

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

protocol_property/2

Description

```
protocol_property(Protocol, Property)
```

Enumerates, by backtracking, the properties associated with the currently defined protocols.

Template and modes

```
protocol_property(?protocol_identifier, ?protocol_property)
```

Errors

Protocol is neither a variable nor a valid protocol identifier:

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

Property is neither a variable nor a valid protocol property:

```
domain_error(protocol_property, Property)
```

Examples

```
| ?- 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:

```
in 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:

```
| ?- create_object(hovercraft, [specializes(vehicle)], [public([propeller/2,  
fan/2])]), []).
```

Creating a new object and defining its initialization goal:

```
| ?- create_object(runner, [instantiates(runners)], [initialization(start)],  
[length(22), time(60)]).
```

Creating a new empty object with dynamic predicate declarations support:

```
| ?- create_object(database, [], [set_logtalk_flag(dynamic_declarations, allow)],  
[]).
```

create_protocol/3

Description

```
create_protocol(Identifier, Relations, Directives)
```

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

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_protocol(?protocol_identifier, +list, +list)
```

Errors

Either Relations or Directives is a variable:

```
in instantiation_error
```

Identifier is neither a variable nor a valid protocol identifier:

```
type_error(protocol_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)
```

Examples

```
| ?- create_protocol(  
    logging,  
    [extends(monitoring)],  
    [public([log_file/1, log_on/0, log_off/0])]  
    ).
```

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(monitored).
```

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:

```
in instantiation_error
```

Object is neither a variable nor a valid object identifier:

```
in type_error(object_identifier, Object)
```

Object is an identifier of a static object:

```
in permission_error(modify, static_object, Object)
```

Object does not exist:

```
in 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

```
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

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

Errors

Prototype is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Prototype)
```

Parent is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Parent)
```

Scope is neither a variable nor a valid entity scope:

```
type_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 a valid entity scope:

```
type_error(scope, Scope)
```

Examples

```
| ?- extends_protocol(listp, Protocol).
| ?- extends_protocol(Protocol, temp, 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:

```
type_error(category_identifier, Category1)
```

Category2 is neither a variable nor a valid protocol identifier:

```
type_error(category_identifier, Category2)
```

Scope is neither a variable nor a valid entity scope:

```
type_error(scope, Scope)
```

Examples

```
| ?- extends_category(basic, Category).
| ?- extends_category(Category, extended, private).
```

implements_protocol/2-3

Description

```
implements_protocol(Object, Protocol)
implements_protocol(Category, Protocol)

implements_protocol(Object, Protocol, Scope)
implements_protocol(Category, Protocol, Scope)
```

Enumerates, by backtracking, all pairs of entities such that an object or a category implements a protocol. The relation scope is represented by the atoms `public`, `protected`, and `private`.

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 a valid entity scope:

```
type_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 a valid entity scope:

```
type_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 a valid entity scope:

```
type_error(scope, Scope)
```

Examples

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

specializes_class/2-3

Description

```
specializes_class(Class, Superclass)
specializes_class(Class, Superclass, Scope)
```

Enumerates, by backtracking, all pairs of objects such that the first one specializes the second. The relation scope is represented by the atoms `public`, `protected`, and `private`.

Template and modes

```
specializes_class(?object_identifier, ?object_identifier)
specializes_class(?object_identifier, ?object_identifier, ?scope)
```

Errors

Class is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Class)
```

Superclass is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Superclass)
```

Scope is neither a variable nor a valid entity scope:

```
type_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).
```

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:

```
type_error(event, Event)
```

Object is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Object)
```

Message is neither a variable nor a callable term:

```
type_error(callable, Message)
```

Sender is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Sender)
```

Monitor is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Monitor)
```

Examples

```
| ?- abolish_events(_, list, _, _, debugger).
```

current_event/5

Description

```
current_event(Event, Object, Message, Sender, Monitor)
```

Enumerates, by backtracking, all defined events. The two types of events are represented by the atoms `before` and `after`.

Template and modes

```
current_event(?event, ?object_identifier, ?callable, ?object_identifier, ?object_identifier)
```

Errors

Event is neither a variable nor a valid event identifier:

```
type_error(event, Event)
```

Object is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Object)
```

Message is neither a variable nor a callable term:

```
type_error(callable, Message)
```

Sender is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Sender)
```

Monitor is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Monitor)
```

Examples

```
| ?- current_event(Event, Object, Message, Sender, debugger).
```

define_events/5

Description

```
define_events(Event, Object, Message, Sender, Monitor)
```

Defines a new set of events. The two types of events are represented by the atoms `before` and `after`. The object `Monitor` must define the event handler methods required by the `Event` argument.

Template and modes

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

Errors

Event is neither a variable nor a valid event identifier:

```
type_error(event, Event)
```

Object is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Object)
```

Message is neither a variable nor a callable term:

```
type_error(callable, Message)
```

Sender is neither a variable nor a valid object identifier:

```
type_error(object_identifier, Sender)
```

Monitor is a variable:

```
in instantiation_error
```

Monitor is neither a variable nor a valid object identifier:

```
existence_error(object_identifier, Monitor)
```

Monitor does not define the required `before/3` method:

```
existence_error(procedure, before/3)
```

Monitor does not define the required `after/3` method:

```
existence_error(procedure, after/3)
```

Examples

```
| ?- define_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

```
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

```
threaded_call(@callable)
threaded_call(@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_call(Goal)
```

Prove `::Message` asynchronously in a new thread:

```
threaded_call(::Message)
```

Prove `Object::Message` asynchronously in a new thread:

```
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/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_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

```
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

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

Errors

(none)

Examples

Wait until the `data_available` notification is received:

```
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 configuration file. The Logtalk source file name extension (by default, `.lgt`) must be omitted. Note that the argument is a source file name or a list of source file names, not file paths. In other words, the files must exist in the current working directory, unless library notation is used.

Note that only the errors related to problems in the predicate argument are listed below. Other exceptions may be thrown by the compiler if any of the compiled entities contain syntax errors.

Template and modes

```
logtalk_compile(@source_file_names)
```

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, is a compound term but not a library name:

```
type_error(library_name, File)
```

File or an element File of the Files list does not exist in the current working directory or in the specified library directory:

```
existence_error(file, File)
```

Source file library does not exist:

```
existence_error(library, Library)
```

Examples

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

logtalk_compile/2

Description

```
logtalk_compile(File, Flags)
logtalk_compile(Files, Flags)
```

Compiles to disk a source file or a list of source files using a list of flag values. The Logtalk file name extension (by default, `.lgt`) must be omitted. Note that the first argument is a source file name or a list of source file names, not file paths. In other words, the files must exist in the current working directory, unless library notation is used.

For a description of the available compiler flags, please consult the User Manual.

Note that only the errors related to problems in the predicate arguments are listed below. Other exceptions may be thrown by the compiler if any of the compiled entities contain syntax errors.

Template and modes

```
logtalk_compile(@source_file_names, @list)
```

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 an entity file name nor a library entity file name:

```
type_error(source_file_name, File)
```

File, or an element File of the Files list, is a compound term but not a library entity file name:

```
type_error(library_source_file_name, Entity)
```

File or an element File of the Files list does not exist in the current working directory or in the specified library directory:

```
existence_error(file, File)
```

Entity library does not exist:

```
existence_error(library, Library)
```

Flags 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 valid:

```
type_error(compiler_flag, Flag)
```

Examples

```
| ?- logtalk_compile(list, []).  
  
| ?- logtalk_compile(types(tree), [xmlspec(xsd)]).  
  
| ?- logtalk_compile([listp, list], [xml(off), plredf(warning)]).
```

logtalk_load/1

Description

```
logtalk_load(File)
logtalk_load(Files)
```

Compiles to disk and then loads to memory a source file or a list of source files using the default compiler flags specified in the Logtalk configuration file. The Logtalk file name extension (by default, `.lgt`) must be omitted. Note that the argument is a source file name or a list of source file names, not file paths. In other words, the files must exist in the current working directory, unless library notation is used.

Note that only the errors related to problems in the predicate argument are listed below. Other exceptions may be thrown by the compiler if any of the loaded entities contain syntax errors.

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

Template and modes

```
logtalk_load(@source_file_names)
```

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 nor a library entity file name:

```
type_error(source_file_name, File)
```

File, or an element File of the Files list, is a compound term but not a library source file name:

```
type_error(library_source_file_name, File)
```

File or an element File of the Files list does not exist in the current working directory or in the specified library directory:

```
existence_error(file, File)
```

Entity library does not exist:

```
existence_error(library, Library)
```

Examples

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

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 flag values. The Logtalk file name extension (by default, `.lgt`) must be omitted. Note that the first argument is a source file name or a list of source file names, not file paths. In other words, the files must exist in the current working directory, unless library notation is used.

For a description of the available compiler flags, please consult the User Manual.

Note that only the errors related to problems in the predicate arguments are listed below. Other exceptions may be thrown by the compiler if any of the loaded entities contain syntax errors.

Template and modes

```
logtalk_load(@source_file_names, @list)
```

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, is a compound term but not a library source file name:

```
type_error(library_source_file_name, Entity)
```

File or an element File of the Files list does not exist in the current working directory or in the specified library directory:

```
existence_error(file, File)
```

Entity library does not exist:

```
existence_error(library, Library)
```

Flags 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 valid:

```
type_error(compiler_flag, Flag)
```

Examples

```
| ?- logtalk_load(list, []).  
  
| ?- logtalk_load(types(tree), [xmlspec(xsd)]).  
  
| ?- logtalk_load([listp, list], [xml(off), plredf(warning)]).
```

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 ("/").

Template and modes

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

Errors

(none)

Examples

```
| ?- logtalk_library_path(viewpoints, Path).  
  
Path = examples('viewpoints/')  
yes  
  
| ?- logtalk_library_path(Library, Path).  
  
Library = lgtuser  
Path = '$LOGTALKUSER/' ;  
  
Library = library  
Path = lgtuser('library/') ;  
  
Library = examples  
Path = lgtuser('examples/') ;  
  
Library = viewpoints  
Path = examples('viewpoints/')  
yes
```

current_logtalk_flag/2

Description

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

Enumerates, by backtracking, the current Logtalk flag values.

Template and modes

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

Errors

Flag is neither a variable nor an atom:

```
type_error(atom, Flag)
```

Flag is not a valid flag:

```
domain_error(logtalk_flag, Value)
```

Examples

```
| ?- current_logtalk_flag(xml, Value).
```

set_logtalk_flag/2

Description

```
set_logtalk_flag(Flag, Value)
```

Sets Logtalk flag values.

Template and modes

```
set_logtalk_flag(+atom, +atom)
```

Errors

Flag is a variable:

```
in instantiation_error
```

Value is a variable:

```
in instantiation_error
```

Flag is not an atom:

```
in type_error(atom, Flag)
```

Flag is neither a variable nor a valid flag:

```
in domain_error(logtalk_flag, Flag)
```

Value is not a valid value for flag Flag:

```
in domain_error(flag_value, Flag + Value)
```

Flag is a read-only flag:

```
in permission_error(modify, flag, Flag)
```

Examples

```
| ?- set_logtalk_flag(xmldocs, on).
```

forall/2

Description

```
forall(Generator, Test)
```

This predicate is true if, for all solutions of Generator, Test is true (some Prolog compilers already define this or a similar predicate).

Template and modes

```
forall(+callable, +callable)
```

Errors

Generator is not a callable term:

```
type_error(callable, Generator)
```

Test is not a callable term:

```
type_error(callable, Test)
```

Examples

```
| ?- forall(member(X, [1, 2, 3]), write(X)).
```

```
123
```

```
yes
```

retractall/1

Description

```
retractall(Head)
```

Logtalk adds this built-in predicate, with the usual definition, to a Prolog compiler if it is not already defined.

Template and modes

```
retractall(+callable)
```

Errors

Head is not a callable term:

```
type_error(callable, Head)
```

Examples

```
| ?- retractall(foo(_)).
```

Built-in methods

parameter/2

Description

```
parameter(Number, Term)
```

Used in parametric objects (and parametric categories), this private method provides runtime access to parameter values by using the argument number in the object (or category) 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. For parametric objects, this unification occurs at the clause head, not at the clause body. See also [this/1](#).

Template and modes

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

Errors

Number is a variable:

```
instantiation_error
```

Number is neither a variable nor an integer value:

```
type_error(integer, Number)
```

Object identifier is not a compound term:

```
type_error(compound, Object)
```

Number is a negative integer value:

```
domain_error(not_less_than_zero, Number)
```

Examples

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

...

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

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 that contains the predicate clause whose body is being executed (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_predicate/1

Description

```
current_predicate(Predicate)
```

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

Template and modes

```
current_predicate(?predicate_indicator)
```

Errors

Predicate is neither a variable nor a valid predicate indicator:

```
type_error(predicate_indicator, Predicate)
```

Examples

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

```
current_predicate(Predicate)
```

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

```
::current_predicate(Predicate)
```

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

```
Object::current_predicate(Predicate)
```

predicate_property/2

Description

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

Enumerates, by backtracking, the properties of a visible predicate.

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 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

```
asserta(Head)
asserta((Head:-Body))
```

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 the compiler option `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

```
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 is that of a private predicate:

```
permission_error(modify, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(modify, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(modify, static_predicate, Head)
```

Target object was created/compiled with support for dynamic declaration of predicates turned off:

```
permission_error(create, predicate_declaration, Head)
```

Examples

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

```
asserta(Clause)
```

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

```
::asserta(Clause)
```

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

```
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 the compiler option `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 is that of a private predicate:

```
permission_error(modify, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(modify, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(modify, static_predicate, Head)
```

Target object was created/compiled with support for dynamic declaration of predicates turned off:

```
permission_error(create, predicate_declaration, Head)
```

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 neither a variable nor a callable term:

```
type_error(callable, Head)
```

Body is neither a variable nor a callable term:

```
type_error(callable, Body)
```

The predicate indicator of Head is that of a private predicate:

```
permission_error(access, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(access, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(access, static_predicate, Head)
```

Head is not a declared predicate:

```
existence_error(predicate_declaration, Head)
```

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(Clause)
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 is that of a private predicate:

```
permission_error(modify, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(modify, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(modify, static_predicate, Head)
```

The predicate indicator of Head is not declared:

```
existence_error(predicate_declaration, Head)
```

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 is that of a private predicate:

```
permission_error(modify, private_predicate, Head)
```

The predicate indicator of Head is that of a protected predicate:

```
permission_error(modify, protected_predicate, Head)
```

The predicate indicator of Head is that of a static predicate:

```
permission_error(modify, static_predicate, Head)
```

The predicate indicator of Head is not declared:

```
existence_error(predicate_declaration, Head)
```

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.

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, ...)
```

once/1

Description

```
once(Goal)
```

This predicate behaves as `call(Goal)` but it is not re-executable. This built-in meta-predicate is declared as a private method and thus cannot be used as a message to an object.

Template and modes

```
once(+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 deterministically in the context of the object or category containing the call:

```
once(Goal)
```

To send a goal as a non-backtracable message to *self*:

```
once(::Goal)
```

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

```
once(Object::Goal)
```

`\+/1`

Description

```
\+ Goal
```

Not-provable meta-predicate. True iff `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:

```
instantiation_error
```

Goal is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

Examples

Not-provable goal in the context of the object or category containing the call:

```
\+ Goal
```

Not-provable goal sent as a message to *self*:

```
\+ ::Goal
```

Not-provable goal sent as a message to an explicit object:

```
\+ Object::Goal
```

catch/3

Description

```
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

```
catch(?callable, ?term, ?term)
```

Errors

Goal is a variable:

```
instantiation_error
```

Goal is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

Examples

(none)

throw/1

Description

```
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

```
throw(+nonvar)
```

Errors

Exception is a variable:

```
instantiation_error
```

Exception does not unify with the second argument of any call of `catch/3`:

```
system_error
```

Examples

(none)

bagof/3

Description

```
bagof(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

```
bagof(@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:

```
bagof(Term, Goal, List)
```

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

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

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

```
bagof(Term, Object::Goal, List)
```

findall/3

Description

```
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

```
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:

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

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

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

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

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

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

```
before(Object, Message, Sender)
```

User-defined public method for handling `before` events. This public method is declared in the `monitoring` built-in protocol.

Template and modes

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

Errors

(none)

Examples

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

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

after/3

Description

```
after(Object, Message, Sender)
```

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

Template and modes

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

Errors

(none)

Examples

```
:- object(...,  
  implements(monitoring),  
  ...).  
  
after(Object, Message, Sender) :-  
  writeq(Object), write('::'), writeq(Message),  
  write(' from '), writeq(Sender), nl.
```

call//1

Description

```
call(Closure)
```

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)  
...
```

Errors

Closure is a variable:

```
in instantiation_error
```

Closure is neither a variable nor a callable term:

```
in 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/2

Description

```
phrase(NonTerminal, Input)
phrase(::NonTerminal, Input)
phrase(Object::NonTerminal, Input)
```

True if the list `Input` of tokens can be parsed using the specified non-terminal `NonTerminal`. This method also accepts grammar rule bodies in the first argument. 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:NonTerminal, Input)` may also be used.

This method is opaque to cuts in the first argument.

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)
```

Input is neither a partial list nor a list:

```
type_error(list, Input)
```

The grammar rule non-terminal NonTerminal is private:

```
permission_error(access, private_non_terminal, NonTerminal)
```

The grammar rule non-terminal NonTerminal is protected:

```
permission_error(access, protected_non_terminal, NonTerminal)
```

The grammar rule non-terminal NonTerminal is not declared:

```
existence_error(non_terminal_declaration, 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

```
phrase(NonTerminal, Input, Rest)
phrase(::NonTerminal, Input, Rest)
phrase(Object::NonTerminal, Input, Rest)
```

True if the list `Input` of tokens can be parsed using the specified non-terminal `NonTerminal`. The list `Rest` is what remains of the list `Input` after parsing succeeded. This method also accepts grammar rule bodies in the first argument. 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:NonTerminal, Input, Rest)` may also be used.

This method is opaque to cuts in the first argument.

Template and modes

```
phrase(+callable, ?list, ?list)
```

Errors

NonTerminal is a variable:

```
instantiation_error
```

NonTerminal is neither a variable nor a callable term:

```
type_error(callable, NonTerminal)
```

Input is neither a partial list nor a list:

```
type_error(list, Input)
```

Rest is neither a partial list nor a list:

```
type_error(list, Rest)
```

The grammar rule non-terminal NonTerminal is private:

```
permission_error(access, private_non_terminal, NonTerminal)
```

The grammar rule non-terminal NonTerminal is protected:

```
permission_error(access, protected_non_terminal, NonTerminal)
```

The grammar rule non-terminal NonTerminal is not declared:

```
existence_error(non_terminal_declaration, NonTerminal)
```

Examples

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

```
phrase(NonTerminal, Input, Rest)
```

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

```
phrase(::NonTerminal, Input, Rest)
```

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

```
phrase(Object::NonTerminal, Input, Rest)
```

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 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 the Logtalk compiler to expand terms read from a source file.

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 using a scope directive for it. 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 used either by calling the `expand_term/2` method explicitly or by using *hook objects*. 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.

Template and modes

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

Errors

(none)

Examples

(none)

expand_goal/2

Description

```
expand_goal(Goal, Expansion)
```

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 called 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 the Logtalk compiler to expand the body of clauses read from a source file.

Template and modes

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

Errors

(none)

Examples

(none)

goal_expansion/2

Description

```
goal_expansion(Goal, ExpandedGoal)
```

Defines an expansion for a goal. The first argument is the goal to be expanded. The expanded goal is returned in the second argument. This predicate is called recursively on the expanded goal until 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 using a scope directive for it. This predicate is called when compiling source files.

Goal expansion may be used either by calling the `expand_goal/2` method explicitly or by using *hook objects*. 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.

Template and modes

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

Errors

(none)

Examples

```
goal_expansion(write(Term), (write_term(Term, []), nl)).
```


Control constructs

::/2**Description**

```
Object::Predicate  
{Proxy}::Predicate
```

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*.

The `{Proxy}::Predicate` syntax construct allows simplified access to parametric object *proxies*. Its operational semantics is equivalent to the goal conjunction `({Proxy}, Proxy::Predicate)`. I.e. `Proxy` is proved as a plain Prolog goal 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.

Template and modes

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

Errors

Either Object or Predicate is a variable:

```
instantiation_error
```

Object is not a valid object identifier:

```
type_error(object_identifier, Object)
```

Predicate is neither a variable nor a callable term:

```
type_error(callable, Predicate)
```

Predicate is declared private:

```
permission_error(access, private_predicate, Predicate)
```

Predicate is declared protected:

```
permission_error(access, protected_predicate, Predicate)
```

Predicate is not declared:

```
existence_error(predicate_declaration, Predicate)
```

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

```
| ?- list::member(X, [1, 2, 3]).  
  
X = 1 ;  
X = 2 ;  
X = 3  
yes
```

::/1**Description**

```
::Predicate
```

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 inside a category, or when using private inheritance.

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 is declared private:

```
permission_error(access, private_predicate, Predicate)
```

Predicate is not declared:

```
existence_error(predicate_declaration, Predicate)
```

Examples

```
area(Area) :-  
  ::width(Width),  
  ::height(Height),  
  Area is Width*Height.
```

^^/1**Description**

```
^^Predicate
```

Calls an inherited predicate definition. The call fails if there is no inherited predicate definition. This control construct may be used within objects or categories in the body of a predicate definition. When used within a category, the lookup for the overridden predicate definition is restricted to the extended categories. This control construct is optimized for the case where the inherited predicate being called is the same predicate being redefined.

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 is declared private:

```
permission_error(access, private_predicate, Predicate)
```

Predicate is not declared:

```
existence_error(predicate_declaration, Predicate)
```

Examples

```
init :-  
    assertz(counter(0)),  
    ^^init.
```

`{}/1`

Description

```
{Goal}
```

Calls external Prolog code. Can be used to bypass the Logtalk compiler. 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`.

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 is used as an object identifier in the message sending call.

Template and modes

```
{+callable}
```

Errors

Goal is a variable:

```
instantiation_error
```

Goal is neither a variable nor a callable term:

```
type_error(callable, Goal)
```

Examples

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

<</2

Description

```
Object<<Goal
```

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 is mainly used for debugging and for writing object unit tests. This control construct can only be used for objects compiled with the compiler option `context_switching_calls` set to `allow`. Set this compiler option to `deny` to disable this control construct.

Template and modes

```
+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, predicate, Goal)
```

Examples

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

:/1**Description**

```
:Predicate
```

Calls an **imported** predicate with the lookup for the predicate declaration beginning in *this* instead of *self* and with the lookup for the predicate definition restricted to the imported categories. As a consequence, any redeclaration or redefinition of the predicate in a descendant of the object containing the call will be ignored. The predicate is called with the same execution context (*sender*, *this*, and *self*) as the predicate whose body contains the call. When the predicate is defined in an imported category compiled using static binding, this control construct allows the predicate to be called with the same performance as a local object predicate.

Template and modes

```
:+callable
```

Errors

Predicate is a variable:

```
in instantiation_error
```

Predicate is neither a variable nor a callable term:

```
in type_error(callable, Predicate)
```

Predicate is not declared:

```
in existence_error(predicate_declaration, Predicate)
```

Examples

```
:- object(bounded_point,
  imports(bounded_coordinate),
  instantiates(class),
  specializes(point)).

move(X, Y) :-
  :check_bounds(x, X),      % defined in the "bounded_coordinate" category
  :check_bounds(y, Y),
  ^^move(X, Y).
```