Predicates

Predicate directives and clauses can be encapsulated inside objects and categories. Protocols can only contain predicate directives.

Declaring predicates

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

Predicate directives should always precede the corresponding predicate definitions and/or calls in the source files in order to ensure proper compilation.

Scope directives

A predicate can be public, protected, or private. Public predicates can be called from any object. Protected predicates can only be called from the container object or from a container descendant. Private predicates can only be called from the container object.

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

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

:- protected(valid_init_option/1).

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

Note that we do not need to write scope declarations for all defined predicates. If a predicate does not have a scope declaration, it is assumed that the predicate is private, although it will be invisible to the reflection methods and to the message and error handling mechanisms. One exception is local, dynamic predicates: explicitly declaring them as private predicates allows the Logtalk compiler to generate optimized code for them.

Mode directive

Many predicates cannot be called with arbitrary arguments with arbitrary instantiation status. The valid arguments and instantiation modes can be documented by using the `mode/2` directive. For instance:

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

The first argument describes a valid calling mode. The minimum information will be the instantiation mode of each argument. There are four possible values (described in [ISO 95]):

`+`
: Argument must be instantiated.

`-`
: Argument must be a free (non-instantiated) variable.

`?`
: Argument can either be instantiated or free.

`@`
: Argument will not be modified.

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

The second argument documents the number of proofs (or solutions) for the specified mode. The possible values are:

`zero`
: Predicate always fails.

`one`
: Predicate always succeeds once.

`zero_or_one`
: Predicate either fails or succeeds.

`zero_or_more`
: Predicate has zero or more solutions.

`one_or_more`
: Predicate has one or more solutions.

`error`
: Predicate will throw an error (see below).

Mode declarations can also be used to document that some call modes will throw an error. For instance, regarding the `arg/3` ISO Prolog built-in predicate, we may write:

```
:- mode(arg(-, -, +), error).
```

Note that most predicates have more than one valid mode implying several mode directives. For example, to document the possible use modes of the `atom_concat/3` ISO built-in predicate we would write:

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

Some old Prolog compilers supported some sort of mode directives to improve performance. To the best of my knowledge, there is no modern Prolog compiler supporting these kind of directive. The current version of the Logtalk compiler just parses and than discards this directive. Nevertheless, the use of mode directives is a good starting point to the documentation of your predicates.

Meta-predicate directive

Some predicates may have arguments that will be called as goals or closures that will be used for constructing a call. To ensure that these calls and closures will be executed in the correct scope (i.e. in the calling context, not in the predicate definition context) we need to use the `meta_predicate/1` directive. For example:

```
:- meta_predicate(findall(*, ::, *)).
```

The predicate arguments in this directive have the following meaning:

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

`N`
: Meta-argument that will be a closure used to construct a call by appending `N` arguments at the end. The value of `N` must be a positive integer.

`*`
: Normal argument.

This is similar to the declaration of meta-predicates in the ISO standard for Prolog modules except that we use the atom `::` instead of `:` to be consistent with the message sending operators. To the best of my knowledge, the use of non-negative integers to specify closures has first introduced on Quintus Prolog for prividing information for the cross-reference tools.

The `meta_predicate/1` directive must precede the meta-predicate definition and any local calls to the meta-predicate in order to ensure proper compilation. In addition, as each Logtalk entity is independently compiled, this directive must be included in every object or category that contains a definition for the described predicate, even if the predicate declaration is inherited from another entity, to ensure proper compilation of meta-arguments.

Discontiguous directive

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

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

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

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

Dynamic directive

An object (or category) predicate can be static or dynamic. By default, all object predicates are static. To declare a dynamic predicate we use the `dynamic/1` directive:

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

This directive may also be used to declare dynamic grammar rule non-terminals. As each Logtalk entity is compiled independently from other entities, this directive must be included in every object or category that contains a definition for the described predicate (even if the predicate declaration is inherited from other entity). If we omit the dynamic declaration

then the predicate definition will be compiled to static code. Note that any static object may declare and define dynamic predicates.

Operator directive

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

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

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

Uses directive

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

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

Logtalk provides a directive, `uses/2`, which allows us to simplify the code above. The usage template for this directive is:

```
:- uses(Object, [Functor1/Arity1, Functor2/Arity2, ...]).
```

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

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

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

Logtalk supports an extended version of this directive that allows the declaration of predicate alias using the notation `Predicate::Alias`. For example:

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

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

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

In the current Logtalk version, the omission of the `Object::` prefix is not supported when the predicate call occurs as an argument of a user-defined meta-predicate (Logtalk specified meta-predicates and Prolog non-standard meta-predicates declared in the config files pose no problem).

Alias directive

Logtalk allows the definition of an alternative name for an inherited or imported predicate (or for an inherited or imported grammar rule non-terminal) through the use of the `alias/3` directive:

```
:- alias(Entity, Predicate, Alias).
```

This directive can be used in objects, protocols, or categories. The first argument, `Entity`, must be an entity referenced in the opening directive of the entity contain the `alias/3` directive. It can be an implemented protocol, an imported category, an extended prototype, an instantiated class, or a specialized class. The second and third arguments are predicate indicators (or grammar rule non-terminal indicators).

A common use for the `alias/3` directive is to give an alternative name to an inherited predicate in order to improve readability. For example:

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

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

    ...

:- end_object.
```

The directive allows both `width/1` and `side/1` to be used as messages to the object `square`. Thus, using this directive, there is no need to explicitly declare and define a "new" `side/1` predicate. Note that the `alias/3` directive does not rename a predicate, only provides an alternative, additional name; the original name continues to be available.

Another common use for this directive is to solve conflicts when two inherited predicates have the same functor and arity. We may want to call the predicate which is masked out by the Logtalk lookup algorithm (see the Inheritance section) or

we may need to call both predicates. This is simply accomplished by using the `alias/3` directive to give alternative names to masked out or conflicting predicates. Consider the following example:

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

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

    ...

:- end_object.
```

Assuming that both `list` and `set` objects define a `member/2` predicate, without the `alias/3` directives, only the definition of `member/2` predicate in the object `list` would be visible on the object `my_data_structure`, as a result of the application of the Logtalk predicate lookup algorithm. By using the `alias/3` directives, all the following messages would be valid (assuming a public scope for the predicates):

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

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

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

When used this way, the `alias/3` directive provides functionality similar to programming constructs of other object-oriented languages which support multi-inheritance (the most notable example probably being the renaming of inherited features in Eiffel).

Note that the `alias/3` directive never hides a predicate which is visible on the entity containing the directive as a result of the Logtalk lookup algorithm. However, it may be used to make visible a predicate which otherwise would be masked by another predicate, as illustrated in the above example.

The `alias/3` directive may also be used to give access to an inherited predicate, which otherwise would be masked by another inherited predicate, while keeping the original name as follows:

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

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

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

    ...

:- end_object.
```

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

Documenting directive

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

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

The second argument is a list of `Key is Value` terms. See the Documenting Logtalk programs session for details.

Defining predicates

Object predicates

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

```
:- object(list).

    :- public(append/3).
    :- public(member/2).

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

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

:- end_object.
```

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

Category predicates

Because a category can be imported by several different objects, dynamic private predicates must be called using the `::/1` message sending operator. This ensures that the correct predicate definition will be used. For example, if we want to define a category implementing variables using destructive assignment we could write:

```
:- category(variable).

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

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

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

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

:- end_category.
```

This way, each importing object will have its own definition for the `value_/2` private predicate. Furthermore, the `get/2` and `set/2` predicates will always access/update the correct definition, contained in the object receiving the messages.

A category may only contain clauses for static predicates. Nevertheless, as the example above illustrates, there are no restrictions in declaring and calling dynamic predicates from inside a category.

Meta-predicates

Meta-predicates may be defined inside objects (and categories) as any other predicate. A meta-predicate is declared using the `meta_predicate/1` directive as described earlier on this section. When defining a meta-predicate, the arguments in the clause heads corresponding to the meta-arguments must be variables. All meta-arguments are called in the context of the *sender* of the message that resulted in the execution of the meta-predicate.

Some meta-predicates have meta-arguments which are not goals but closures. Logtalk supports the definition of meta-predicates that are called with closures instead of goals as long as the definition uses the Logtalk built-in predicates `call/N` to call the closure with the addtional arguments. For example:

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

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

Note that the meta-predicate directive specifies that the closure will be extended with one extra argument.

Definite clause grammars

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

```
:- object(calculator).

    :- public(parse/2).

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

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

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

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

:- end_object.
```

The predicate `phrase/2` called in the definition of predicate `parse/2` above is a Logtalk built-in method, similar to the predicate with the same name found on most Prolog compilers that support definite clause grammars. After compiling and loading this object, we can test the grammar rules with calls such as the following one:

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

Result = -9
yes
```

In most cases, the predicates resulting from the translation of the grammar rules to regular clauses are not declared. Instead, these predicates are usually called by using the built-in methods `phrase/2` and `phrase/3` as shown in the example above. When we want to send the messages `phrase/2` and `phrase/3` to *self* or to another object, the non-terminal used as first argument must be within the scope of the *sender*. For the above example, assuming that we want the predicate corresponding to the `expr//1` non-terminal to be public, the corresponding scope directive would be:

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

The `//` infix operator used above tells the Logtalk compiler that the scope directive refers to a grammar rule non-terminal, not to a predicate. The idea is that the predicate corresponding to the translation of the `expr//1` non-terminal will have

a number of arguments equal to one plus the number of additional arguments necessary for processing the subjacent lists of tokens.

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

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

    :- public(parse/2).

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

    sentence --> noun_phrase, verb_phrase.

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

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

:- end_object.
```

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

```
:- category(determiners).

    :- private(determiner//0).

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

:- end_category.
```

Along with the message sending operators (`::/1` and `::/2`), we may also use other control constructs such as `\+/1`, `!/0`, `;/2`, `->/2`, and `{}/1` in the body of a grammar. In addition, grammar rules may contain meta-calls (a variable taking the place of a non-terminal), which are translated to calls of the built-in method `phrase/3`.

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

call a goal from a grammar rule body, while bypassing at the same time the Logtalk compiler. Consider the following example:

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


:- object(bypass).

    :- public(foo//0).

    foo --> {{bar}}.

:- end_object.
```

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

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

bar predicate called
yes
```

This is the expected result as the expansion of the grammar rule into a clause leaves the `{bar}` goal untouched, which, in turn, is converted into the goal `bar` when the clause is compiled.

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

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

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

Built-in object predicates (methods)

Logtalk defines a set of built-in object predicates or methods to access message execution context, to find sets of solutions, to inspect objects and for database handling. Similar to Prolog built-in predicates, these built-in methods should not be redefined.

Execution context methods

Logtalk defines four built-in methods to access an object execution context. These methods (with the possible exception of `parameter/2`) are translated to a single unification performed at compile time with a clause head context argument. Therefore, they can be freely used without worrying about performance penalties. When called from inside a category, these methods refer to the execution context of the object importing the category.

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

The method `this/1` enables us to retrieve the name of the object that contains the code that is being executed instead of using the name directly. This helps to avoid breaking the code if we decide to change the object name and forget to change the name references.

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

```
:- object(test).

    :- public(test/0).

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

:- end_object.


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

:- end_object.
```

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

```
| ?- descendant::test.

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

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

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

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

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

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

    :- public(test/0).

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

:- end_object.
```

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

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

Color parameter value is blue
yes
```

The method `parameter/2` is only translated to a compile time unification when used inside objects with its first argument instantiated at compile time. When the first argument is not known at compile time, or when the method is used inside categories, its call implies a call to the built-in Prolog predicate `arg/3`. Nevertheless, note that calls to `parameter/2` from inside categories are inherently problematic: a category may be implemented by several objects, both parametric (with different number of parameters) and non-parametric. Care must be taken to ensure that a parametric object importing such a category match the interpretation of its parameters used in the category.

Database methods

Logtalk provides a set of built-in methods for object database handling similar to the usual database Prolog predicates: `abolish/1`, `asserta/1`, `assertz/1`, `clause/2`, `retract/1`, and `retractall/1`. These methods always operate on the database of the object receiving the corresponding message.

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

Meta-call methods

Logtalk supports the generalizaed `call/N` predicate as a built-in. This built-in predicate must be used in the implementation of meta-predicates which work with closures instead of goals.

All solutions methods

The usual all solutions meta-predicates are pre-defined methods in Logtalk: `bagof/3`, `findall/3`, and `setof/3`. There is also a `forall/2` method that implements generate and test loops.

Reflection methods

Logtalk provides two built-in methods for inspecting object predicates: `predicate_property/2`, which returns predicate properties and `current_predicate/1`, which enables us to query about predicate definitions. See below for a more detailed description of both methods.

Definite clause grammar parsing methods

Logtalk supports two definite clause grammar parsing built-in methods, `phrase/2` and `phrase/3`, with definitions similar to the predicates with the same name found on most Prolog compilers that support definite clause grammars.

Term expansion methods

Logtalk supports a `expand_term/2` built-in method for expanding a term into another. This method is mostly used to translate grammar rules into Prolog clauses. It can be customized, e.g. for bypassing the default Logtalk grammar rule translator, by defining clause for the predicate `term_expansion/2`.

Predicate properties

We can find the properties of visible predicates by calling the `predicate_property/2` built-in method. For example:

```
| ?- bar::predicate_property(foo(_), Property).
```

Note that this method respects the predicate's scope declarations. For instance, the above call will only return properties for public predicates.

An object's set of visible predicates is the union of all the predicates declared for the object with all the built-in methods and all the Logtalk and Prolog built-in predicates.

Possible predicate properties values are:

- `public`, `protected`, `private`
- `static`, `dynamic`
- `built_in`
- `meta_predicate(Mode)`
- `declared_in(Entity)`
- `defined_in(Entity)`
- `non_terminal(NonTerminal//Arity)`
- `alias(Predicate)`

The properties `declared_in/1` and `defined_in/1` do not apply to built-in methods and Logtalk or Prolog built-in predicates. Note that if a predicate is declared in a category imported by the object, it will be the category name — not the object name — that will be returned by the property `declared_in/1`. The same goes for protocol declared predicates.

The property `non_terminal/1` only applies to predicates that result from the compilation of grammar rule non-terminals.

The property `alias/1` is returned for predicates that are an alias to other predicate (which is returned in the property argument).

Finding declared predicates

We can find, by backtracking, all visible user predicates by calling the `current_predicate/1` built-in method. This method respects the predicate's scope declarations. For instance, the following call:

```
| ?- some_object::current_predicate(Functor/Arity).
```

will only return user predicates that are declared public. The predicate property `non_terminal/1` may be used to retrieve all grammar rule non-terminals declared for an object. For example:

```
current_non_terminal(Object, NonTerminal//Args) :-
    Object::current_predicate(Functor/Arity),
    functor(Predicate, Functor, Arity),
    Object::predicate_property(Predicate, non_terminal(NonTerminal//Args)).
```

Usually, the non-terminal and the corresponding predicate share the same functor but users should not rely on this always being true.

Calling Prolog built-in predicates

In predicate definitions, predicate calls which are not prefixed with a message sending operator (either `::` or `^^`), are compiled to either calls to local predicates or as calls to Logtalk/Prolog built-in predicates. A predicate call is compiled as a call to a local predicate if the object (or category) contains a definition for the called predicate or a dynamic declaration for it. When the object (or category) does not contain either a definition of the called predicate or a corresponding dynamic declaration, Logtalk tests if the call corresponds to a Logtalk or Prolog built-in predicate. Calling a predicate which is neither a local predicate nor a Logtalk/Prolog built-in predicate results in a compile time warning. This means that, in the following example:

```
foo :-
    ...,
    write(bar),
    ...
```

the call to the predicate `write/1` will be compiled as a call to the corresponding Prolog built-in predicate unless the object (or category) encapsulating the above definition also contains a predicate named `write/1` or a dynamic declaration for the predicate.

When calling non-standard Prolog built-in predicates, you may run into portability problems when trying your applications with other Prolog compilers which might not have the same set of built-in predicates. You may use the Logtalk compiler flag `portability/1` to help you check for problematic calls in your code.

Calling Prolog non-standard meta-predicates

Compiling calls to non-standard, Prolog built-in meta-predicates can be tricky for two reasons: first, there is no standard way of checking if a built-in predicate is also a meta-predicate and finding out which are its meta-arguments; second, in some cases, the meta-arguments of a meta-predicate are not goals but closures, used for constructing goals. The way the goals are constructed is specific to the meta-predicate and cannot be reliable inferred by the Logtalk compiler. For meta-

predicates whose meta-arguments are directly called as goals, the solution is to explicitly declare them in the corresponding Prolog configuration file using the predicate `'$lgt_pl_meta-predicate'/1`. For example:

```
'$lgt_pl_meta_predicate'(call_with_depth_limit(::, *, *)).
```

Currently, there is no clean workaround for calling non-standard, Prolog built-in meta-predicates whose meta-arguments are used as closures instead of called as goals directly.

XHTML + CSS Last updated on: September 17, 2006