List predicates

In this example, we will illustrate the use of:

- objects
- protocols

by using common list utility predicates.

Defining a list object

We will start by defining an object, `list`, containing predicate definitions for some common list predicates like `append/3`, `length/2` and `member/2`:

```
:- object(list).

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

    append([], List, List).
    append([Head| Tail], List, [Head| Tail2]) :-
        append(Tail, List, Tail2).

    length(List, Length) :-
        length(List, 0, Length).

    length([], Length, Length).
    length([_| Tail], Acc, Length) :-
        Acc2 is Acc + 1,
        length(Tail, Acc2, Length).

    member(Element, [Element| _]).
    member(Element, [_| List]) :-
        member(Element, List).

:- end_object.
```

What is different here from a regular Prolog program? The definitions of the list predicates are the usual ones. We have two new directives, `object/1` and `end_object/0`, that encapsulate the object's code. In Logtalk, by default, all object

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

predicates are private; therefore, we have to explicitly declare all predicates that we want to be public, that is, that we want to call from outside the object. This is done using the `public/1` scope directive.

After we copy the object code to a text file and saved it under the name `list.lgt`, we need to change the Prolog working directory to the one used to save our file (consult your Prolog compiler reference manual). Then, after starting Logtalk (see the Installing and running Logtalk session on the User Manual), we can compile and load the object using the `logtalk_load/1` Logtalk built-in predicate:

```
| ?- logtalk_load(list).

object list loaded
yes
```

We can now try goals like:

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

X = 1;
X = 2;
X = 3;
no
```

or:

```
| ?- list::length([1, 2, 3], L).

L = 3
yes
```

The infix operator `::/2` is used in Logtalk to send a message to an object. The message must match a public object predicate. If we try to call a non-public predicate such as the `length/3` auxiliary predicate an exception will be generated:

```
| ?- list::length([1, 2, 3], 0, L).

uncaught exception:
    error(
        existence_error(predicate_declaration, length([1, 2, 3], 0, L)),
        list::length([1, 2, 3], 0, L),
        user)
```

The error term describes the type of error, the message that caused the exception, and the sender of the message (in this case, the pseudo-object `user` because we are sending the message from the top-level interpreter).

Defining a list protocol

As we saw in the above example, a Logtalk object may contain predicate directives and predicate definitions (clauses). The set of predicate directives defines what we call the object's *protocol* or interface. An interface may have several implementations. For instance, we may want to define a new object that implements the list predicates using difference lists. However, we do not want to repeat the predicate directives in the new object. Therefore, what we need is to split the object's

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

protocol from the object's implementation by defining a new Logtalk entity known as a protocol. Logtalk protocols are compilations units, at the same level as objects and categories. That said, let us define a `listp` protocol:

```
:- protocol(listp).

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

:- end_protocol.
```

Similar to what we have done for objects, we use the directives `protocol/1` and `end_protocol/0` to encapsulate the predicate directives. We can improve this protocol by documenting the call/return modes and the number of solutions of each predicate using the `mode/2` directive:

```
:- protocol(listp).

    :- public(append/3).
    :- mode(append(?list, ?list, ?list), zero_or_more).

    :- public(length/2).
    :- mode(length(?list, ?integer), zero_or_more).

    :- public(member/2).
    :- mode(member(?term, ?list), zero_or_more).

:- end_protocol.
```

We now need to change our definition of the `list` object by removing the predicate directives and by declaring that the object implements the `listp` protocol:

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

   append([], List, List).
   append([Head| Tail], List, [Head| Tail2]) :-
       append(Tail, List, Tail2).

   ...

:- end_object.
```


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

The protocol declared in `listp` may now be alternatively implemented using difference lists by defining a new object, `difflist`:

```
:- object(difflist,
    implements(listp).

    append(L1-X, X-L2, L1-L2).
    ...

:- end_object.
```

Summary

- It is easy to define a simple object: just put your Prolog code inside starting and ending object directives and add the necessary scope directives. The object will be self-defining and ready to use.

- Define a protocol when you may want to provide or enable several alternative definitions to a given set of predicates. This way we avoid needless repetition of predicate directives.

Copyright © Paulo Moura — Logtalk.org
Last updated on: October 26, 2005

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