Chapter 7: Syntactic sugar

(Access predicates • The current topic • Nested queries in rule heads • Alternatives in rule heads • Automated object generation)

Dialog provides a small amount of syntactic sugar, i.e. optional syntax variations that can help make certain code more readable and succinct.

Access predicates

Whereas normal predicates describe what will happen when the program is running, *access predicates* transform the code at compile-time. As a general rule of thumb, they should be used sparingly, and **only for the purpose of increasing source code readability**. This is of course a subjective call.

The rule head of an access predicate definition is prefixed by an @ character. Here is an example from the standard library:

```
@($Obj is open)
~($Obj is closed)
```

@($Obj is open) ~($Obj is closed)
[Copy to clipboard]

The rule body must be a straightforward conjunction of queries: There can be more than one query, and they can be regular, negated, or multi-queries, but no special syntax such as if-statements is allowed.

Access predicates transform queries. Thus, for instance:

```
    (if) (#door is
open) (then) ...
```

(if) (#door is open) (then) ... (endif)
[Copy to clipboard]

is reinterpreted, at compile-time, as:

```
    (if) ~(#door is
closed) (then) ...
```

(if) ~(#door is closed) (then) ... (endif)
[Copy to clipboard]

When an access predicate is part of a (now) statement, the (now) operation is applied to each query appearing inside the rule body of the access predicate. In other words,

```
    (now) (#door is
open)
```

(now) (#door is open)
[Copy to clipboard]

behaves exactly like:

```
    (now) ~(#door
is closed)
```

(now) ~(#door is closed)
[Copy to clipboard]

The standard library uses the following access predicate extensively:

```
@($Obj is $Rel
$Parent)
```

@($Obj is $Rel $Parent)
*($Obj has parent $Parent)
*($Obj has relation $Rel)
[Copy to clipboard]

Relations in the standard library are #in, #on, #heldby, etc. The predicate ($ has relation $) is an ordinary per-object variable, and ($ has parent $) is the special built-in predicate that abstracts the Z-machine object tree operations. Thus, a statement such as:

```
	(now) (#pizza
is #in #oven)
```

```
(now) (#pizza is #in #oven)
```

[Copy to clipboard]

behaves like the following block of statements:

```
	{
		(now)
```

```
{
(now) (#pizza has parent #oven)
(now) (#pizza has relation #in)
}
```

[Copy to clipboard]

which makes the pizza a child of the oven in the object tree, and sets the “has relation” property to #in.

Negative now-statements are allowed with access predicates, but only if the body of the access predicate definition is a single query. This inverts the sense of that query, so that:

```
	(now) ~(#door
is open)
```

```
(now) ~(#door is open)
```

[Copy to clipboard]

is equivalent to:

```
	(now) (#door is
closed)
```

```
(now) (#door is closed)
```

[Copy to clipboard]

Negative now-statements are not allowed for access predicates with more than one query in the body, because a negated conjunction is under-specified: In the statement (now) ~(#pizza is $ $), do we unset the parent, the relation, or both? Incidentally, the standard library sidesteps this thorny philosophical issue by providing a separate access predicate:

```
	(now) (#pizza
is nowhere)
```

```
(now) (#pizza is nowhere)
```

[Copy to clipboard]

When the name of an access predicate appears in rule-head position, it behaves like a collection of rule definitions, lined up vertically. Any body statements in the original clause affect each of the expanded rule definitions. Consider the following definition:

```
($Obj is #in #oven)
	*(edible $Obj)
```

```
($Obj is #in #oven)
 *(edible $Obj)
```

[Copy to clipboard]

Because that rule-head matches an access predicate (defined in the standard library), the code above is equivalent to the following pair of rule definitions:

```
($Obj has parent
#oven)
```

```
($Obj has parent #oven)
 *(edible $Obj)

($Obj has relation #in)
 *(edible $Obj)
```

[Copy to clipboard]

These rule definitions then contribute to the compile-time computation of initial values for the ($ has parent $) and ($ has relation $) dynamic predicates.

When you are modelling your game world, and you wish to create a flag that can be accessed with a pair of antonyms as

in the open/closed example, you have to decide which one of them is the real dynamic predicate (closed in the example), and which one is the access predicate (open). A good guiding principle is to choose a representation where the actual per-object flag is initially unset for most objects. That's because you're not allowed to have a catch-all rule (for the initial value) such as:

```
($ is closed)     %%
Error! This is not
```

($ is closed) %% Error! This is not allowed for dynamic predicates.
[Copy to clipboard]

The reason is that dynamic per-object flags can only be set for objects. However, it's perfectly all right to have a rule that says:

```
($Obj is closed)
(door $Obj)
```

($Obj is closed) (door $Obj)
[Copy to clipboard]

as long as (door $) only succeeds for objects.

An access predicate can have multiple rules. Each rule is tried in order, and the formal parameters are matched against program source code. This is not unification, but a simple form of structural matching:

- A variable (in the access predicate rule head) matches any source-code expression.
- A simple constant, like an object name or the empty list, matches only that constant.
- A list (or partial list) matches a list (or partial list) if the heads match, and the tails match.

Note in particular that this behaviour is assymetric: A variable in the program source code does not match a constant in the access predicate rule.

The result of the transformation is again subjected to access predicate matching, so with recursion it is possible to transform complex source code expressions. The standard library uses this technique to deal with grammar declarations.

Like normal rule definitions, access predicate definitions can appear anywhere in the source code, i.e. before or after the rules in which they are used.

The current topic

Rules that belong to the same predicate form a kind of disjunction, but unlike a single, big (or) statement, the rule definitions can be scattered all over the source code. This allows a kind of aspect-oriented programming, where rules are organized according to their high-level purpose.

In object-oriented languages, source code that is specific to a particular object or class tends to be nested inside a common lexical structure, such as a class definition. Since Dialog objects are just names, we can't organize our code in that particular way. But we may still want to put rules pertaining to a particular object close together:

```
(name #apple)
green apple
```

(name #apple) green apple
(dict #apple) yummy
(fruit #apple)
[Copy to clipboard]

To make such code a little less repetitive, Dialog maintains a *current topic*. The current topic is always an object, and we select it by placing the desired object name on a line of its own, beginning in the very first column (as if it were a rule head):

```
#apple
```

#apple
[Copy to clipboard]

Then, when we want to use that object in a place where a value is expected, we simply type an asterisk (*) instead:

```
#apple
(name *)  green
```

#apple
(name *) green apple
(dict *) yummy

```
(fruit *)
```
[Copy to clipboard]

Use of the current topic is not restricted to rule heads. It works equally well inside queries and list expressions in the rule bodies. Thus, something like this is allowed:

```
#apple
(descr *)  Your eyes
```

```
#apple
(descr *) Your eyes feast upon the (name *).
```
[Copy to clipboard]

It is possible to change the topic at any time, and even to return to an earlier topic in a different part of the source code.

Nested queries in rule heads

As we have seen in many of the examples, predicates are often used to categorize objects. For instance, if (fruit $) is defined for some of the objets in the game, then it's straightforward to query that predicate in order to check whether a particular object is a fruit or not. In addition, a multi-query such as *(fruit $F) can be used to backtrack over every fruit in the game.

We have also seen several examples of rules that employ such a category check as a guard condition:

```
(descr #door)  The
oaken door is oaken.
```

```
(descr #door) The oaken door is oaken.
(descr $Obj)   (fruit $Obj) Yummy!
(descr $)      It seems harmless.
```
[Copy to clipboard]

Dialog provides syntactic sugar to make this look even cleaner: Nested query-expressions in rule heads. These queries are automatically inserted at the beginning of the rule body, in left-to-right order as they appear in the rule head. The nested rules must have at least one parameter, and that (first) parameter is copied into the rule head, replacing the nested query.

Thus,

```
(descr (fruit $Obj))
      Yummy!
```

```
(descr (fruit $Obj))
  Yummy!
```
[Copy to clipboard]

is exactly equivalent to:

```
(descr $Obj)
      (fruit $Obj)
```

```
(descr $Obj)
  (fruit $Obj) Yummy!
```
[Copy to clipboard]

Nested queries can appear anywhere in rule heads, and both negative rules and multi-queries are allowed. The following:

```
(prevent [give
(edible $Obj) to ~
```

```
(prevent [give (edible $Obj) to ~(animate $Target)])
  You can't feed something inanimate.
```
[Copy to clipboard]

is exactly equivalent to:

```
(prevent [give $Obj
to $Target])
```

```
(prevent [give $Obj to $Target])
  (edible $Obj)
```

```
~(animate $Target)
You can't feed something inanimate.
```
[Copy to clipboard]

If a non-anonymous variable appears only once in a rule, the compiler prints a warning about it, because it is likely a typo. Thus, to avoid this warning, it is recommended to simplify:

```
(descr (fruit $Obj))
        Yummy!
```

```
(descr (fruit $Obj))
 Yummy!
```
[Copy to clipboard]

into:

```
(descr (fruit $))
        Yummy!
```

```
(descr (fruit $))
 Yummy!
```
[Copy to clipboard]

It will still be treated as:

```
(descr $Obj)
        (fruit $Obj)
```

```
(descr $Obj)
 (fruit $Obj) Yummy!
```
[Copy to clipboard]

but with some unique, internally-generated variable name instead of “Obj”.

Nested rule-expressions may only appear in rule heads, never inside rule bodies.

Alternatives in rule heads

Dialog provides a shorthand syntax for specifying alternatives in rule heads. A set of simple values (dictionary words, objects, numbers, or the empty list) separated by forward slashes is called a *slash expression*. It is transformed into a nested multi-query to the built-in predicate ($ is one of $):

```
(descr
#apple/#banana/#o
```

```
(descr #apple/#banana/#orange)
  Yummy!
```
[Copy to clipboard]

is equivalent to

```
(descr *($ is one of
[#apple #banana
```

```
(descr *($ is one of [#apple #banana #orange]))
   Yummy!
```
[Copy to clipboard]

which in turn is equivalent to

```
(descr $X)
        *($X is one of
```

```
(descr $X)
 *($X is one of [#apple #banana #orange])
 Yummy!
```
[Copy to clipboard]

where X represents some internally generated name.

Slash expressions are very useful when dealing with user input and synonyms. Here is an example from the standard library:

```
(grammar
[pull/yank/drag/tug/t
```

```
(grammar [pull/yank/drag/tug/tow [object]] for [pull $])
```

[Copy to clipboard]

Because these expressions expand into multi-queries, they can also function as output parameters:

```
(bird
#blackbird/#duck/#
```

```
(bird #blackbird/#duck/#penguin)

(program entry point)
 (exhaust) {
 *(bird $B)
 $B is a bird.
 }
```

[Copy to clipboard]

Slash-expressions may only appear in rule heads, never inside rule bodies.

Automated object generation

Sometimes it is desirable to instantiate several identical objects in a game. It is possible to create each object manually, like this:

```
(green grape
#ggrape1)
```

```
(green grape #ggrape1)
(green grape #ggrape2)
(green grape #ggrape3)

(blue grape #bgrape1)
(blue grape #bgrape2)
(blue grape #bgrape3)
(blue grape #bgrape4)
(blue grape #bgrape5)

(fruit *(green grape $))
(fruit *(blue grape $))

(program entry point)
 (exhaust) {
 *(fruit $F)
 $F is a fruit. (line)
 }
```

[Copy to clipboard]

However, Dialog provides a convenient mechanism for automating the process. The following is functionally equivalent to the above example, although the printed representations of these objects will be different:

```
(generate 3 (green
grape $))
```

```
(generate 3 (green grape $))

(generate 5 (blue grape $))

(fruit *(green grape $))
(fruit *(blue grape $))

(program entry point)
 (exhaust) {
 *(fruit $F)
 $F is a fruit. (line)
 }
```

[Copy to clipboard]

The printed representation of a generated object is a hash character followed by some unique number, since these objects have no source-code names.

Onwards to “Chapter 8: More built-in predicates” • Back to the Table of Contents