Chapter 6: Dynamic predicates

(Global flags • Per-object flags • Global variables • Per-object variables • Has parent)

So far, every example of world modelling that we have seen has been eerily static. An apple is yummy, and will remain so forever. The description of a steak might depend on whether the player eats meat or not, but we haven't seen any language facility that would allow us to adjust the vegetarianism of the player during gameplay, or even to change who the current player is.

Dialog supports four kinds of *dynamic predicates*. They are called *global flags*, *per-object flags*, *global variables*, and *per-object variables*.

Global flags

A *global flag* is a predicate with no parameters, and no side effects. It can either succeed (once) or fail. To set a global flag, use the (now) keyword:

```
	(now) (player
eats meat)
```

(now) (player eats meat)
[Copy to clipboard]

To clear it, use (now) together with a negation:

```
	(now) ~(player
eats meat)
```

(now) ~(player eats meat)
[Copy to clipboard]

We are allowed to define static rules for the global flag, but they won't be accessible at runtime, because queries will merely check the current value of the flag.

However, the compiler will look at the static rules to determine the initial value of the global flag. If there are no rules defined, the flag will be off at the start of the program.

```
(player eats meat)
%% Flag is initially
```

(player eats meat)	%% Flag is initially set.
[Copy to clipboard]

Note that there is no need to declare anywhere that (player eats meat) is a dynamic predicate. That follows implicitly from the fact that we try to use (now) on it. However, if—due to the use of (now)—a predicate is dynamic, then the compiler will enforce certain restrictions on the static rules that define its initial value, for instance that they have no side-effects, and that they don't depend on the value of other dynamic predicates.

Per-object flags

A *per-object flag* is a predicate with one parameter, and no side effects. When queried with a particular object as parameter, it will either succeed (once) or fail. To set the flag for a particular object, use the (now) keyword:

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

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

Per-object flags may only be set for objects (not e.g. lists or numbers). If an attempt is made to set the flag when the parameter isn't bound to an object, a fatal runtime error occurs.

To clear the flag, use (now) together with a negated query:

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

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

It is safe to attempt to clear the flag for a non-object: Nothing happens in that case.

It is also possible to clear the flag for every object in the game, by supplying an anonymous variable as parameter. This

is faster than iterating over each object in turn:

```
      (now) ~($ is
open)
```

(now) ~($ is open)
[Copy to clipboard]

But it is not possible to set the flag for every object in the game in this way; that triggers a runtime error. The rationale for this is that an unbound variable (such as $) may take on any value, e.g. a number or list, and per-object flags may only be set for objects.

To check if the flag is set for a particular object, simply query the predicate:

```
      The door is (if)
(#door is open)
```

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

If the predicate is queried with a parameter that's bound to something other than an object, it fails without generating a fatal error: A per-object flag may only be set for objects, but it can be queried for any value.

If the parameter is unbound, the query binds it to an object for which the flag is set (failing if there are none). With a multi-query, it is possible to backtrack over all objects that have the flag set:

```
(#reddoor is open)
(#bluedoor is open)
```

(#reddoor is open)
(#bluedoor is open)
(#greendoor is open)

(program entry point)
(now) ~(#bluedoor is open)
(collect $Thing)
*($Thing is open)
(into $List)
The open things are: $List
[Copy to clipboard]

This produces the output:

The open things are: [#reddoor #greendoor]

Let's pause for a while and consider a matter of design philosophy. I have claimed that in Dialog, objects are nothing but names, and the world is modelled using relations, represented by predicates. When it comes to per-object flags (and per-object variables, to be introduced below), that standpoint is starting to look tenuous. To be sure, dynamic predicates are designed to look and behave just like ordinary predicates when you query them. And in order to change the state of the game world, you issue now-commands that appear to modify predicates rather than objects. But at the same time, there are restrictions on how you can update those predicates, so that, in effect, what you can do is more or less exactly what you could do by storing flags (and properties) inside objects.

In the end, whether you choose to regard a dynamic per-object flag as something that resides inside the object, or in a separate data structure that represents the dynamic predicate, is entirely up to you. The actual runtime representation is irrelevant, and the compiler will in fact choose among different representations based on how the predicate is accessed from various parts of the program.

Global variables

Global variables are dynamic predicates with a single parameter. To distinguish them from per-object flags, global variables have to be *declared*, using special syntax. This is what it looks like:

```
(global variable <i>
(signature)</i>)
```

(global variable *(signature)*)
[Copy to clipboard]

starting at the leftmost column of a line. For instance:

```
(global variable
(current player $))
```

(global variable (current player $))

[Copy to clipboard]

The initial value of the global variable can be specified with an ordinary rule definition:

```
(current player #bob)
```

(current player #bob)
[Copy to clipboard]

But it is also possible to provide the initial value inside the global variable declaration itself:

```
(global variable
(current player
```

(global variable (current player #bob))
[Copy to clipboard]

To change the value of a global variable, use the (now) keyword:

```
(now) (current
player #alice)
```

(now) (current player #alice)
[Copy to clipboard]

Note that the above is a low-level operation. When working with the standard library, the predicate (select player $) should be used to change the current player. But we'll get to that in Part II.

It is also possible to *unset* the global variable, using ~, so that subsequent queries to the predicate will fail. The parameter must be an anonymous variable. The following line of code could be pronounced “Now, the current player is nobody”, or, in case you're a logician, “Now, it is not the case that the current player is anybody”:

```
      (now) ~(current
player $)
```

(now) ~(current player $)
[Copy to clipboard]

This is also the default state of a global variable, if no initial value is defined.

As usual, we can query the predicate with a bound parameter (to check if the global variable has that particular value), or with an unbound variable (to obtain the current value). Either query will fail if the global variable is unset.

The value that is stored in a global variable must be bound. Attempts to set a global variable to an unbound variable, or a list with an unbound variable inside, will result in a fatal runtime error.

In the following example, a complex global variable is used to implement an AGAIN command:

```
(global variable (last
input $))
```

```
(global variable (last input $))
(ask for command $Result)
 > (get input $Words)
 (if) ($Words = [again]) (or) ($Words = [g]) (then)
 (last input $Result)
 (else)
 ($Result = $Words)
 (now) (last input $Result)
 (endif)
```

[Copy to clipboard]

Per-object variables

A *per-object variable* (sometimes referred to as an object property) is a predicate with two parameters, and no side effects. The first parameter is always an object, and the second parameter can be any bound value. A per-object variable can also be unset for a given object.

When a per-object variable is queried with the first parameter bound to an object, the second parameter will be unified with the current value of the variable. As usual, this can be used to check for a particular value:

```
(if) (#troll
wields #axe) (then)
```

```
(if) (#troll wields #axe) (then) ... (endif)
```
[Copy to clipboard]

or to read the current value:

```
(narrate fight with
$Enemy) %%
```

```
(narrate fight with $Enemy)    %% Assume $Enemy is bound
 ($Enemy wields $Weapon)
 (The $Enemy) swings (the $Weapon) at you.
```
[Copy to clipboard]

The query fails if the variable is unset.

When a per-object variable is queried with an unbound first parameter, Dialog backtracks over every object in the game, and attempts to unify the second parameter with the current value of the corresponding per-object variable. This is potentially a very slow operation, at least on vintage hardware, and the compiler will print a warning if you attempt to do it. To get rid of the warning, you can explicitly backtrack over all objects yourself, by querying *(object $), and then checking the property.

```
($X wields
#axe) The axe is
```

```
($X wields #axe) The axe is currently in the hands of (the $X).
```
[Copy to clipboard]

To change the value of a per-object variable, use the (now) keyword:

```
(now) (#troll
wields #club)
```

```
(now) (#troll wields #club)
```
[Copy to clipboard]

Again, the first parameter must be an object, and the second parameter must be bound. If this is not the case, a fatal runtime error occurs.

To unset the per-object variable, use the following syntax, where the second parameter must be an anonymous variable:

```
(now) ~(#troll
wields $)
```

```
(now) ~(#troll wields $)
```
[Copy to clipboard]

The following is also allowed, and faster than doing it explicitly for each object:

```
(now) ~($
wields $) %%
```

```
(now) ~($ wields $)    %% Nobody wields anything anymore.
```
[Copy to clipboard]

As with the other dynamic predicates, the initial value of a per-object variable is defined with ordinary rules:

```
(#troll wields #club)
```

```
(#troll wields #club)
```
[Copy to clipboard]

Has parent

There is one built-in per-object variable with special properties. This is the ($ has parent $) predicate. It is used to track where in the game world objects are located. In other words, it is an abstraction of the low-level Z-machine object tree operations.

In many ways, ($ has parent $) works just like any other per-object variable: It is queried and modified in the same way, and when it is modified, the first parameter must always be an object. But it has the additional restriction that the second parameter must also be an object. The benefit of this is that reverse lookup operations—backtracking over every child of a particular object—can be implemented very efficiently.

The ($ has parent $) property can be unset in order to detach an object from the object tree.

Here are some examples:

(#troll has parent $Room)

Determine where the troll is.

*($Obj has parent #library)

Backtrack over every object in the library.

(now) (#troll has parent #cave)

Set the parent object of the troll. Under the hood, this will also update the linked structures representing the children of the cave, and the children of the previous parent of the troll.

(now) ~(#axe has parent $)

Remove the axe from play (i.e. detach it from the object tree).

You are responsible for maintaining a well-formed object tree. This means that you're not allowed to create cycles, such as a pair of boxes inside each other. Compiled Dialog code cannot detect violations of this rule at runtime, but the interactive debugger does.

Be wary of updating the object tree while there is an ongoing iteration: An untimely change of a sibling pointer could easily divert the iterating code into a different part of the object tree.

That being said, Dialog guarantees that you can safely iterate over all objects with a particular parent, and move them (or a subset of them) to a different part of the object tree:

```
    (exhaust) {
        *($Obj has
```

```
(exhaust) {
*($Obj has parent #safe)
(now) ($Obj has parent #knapsack)
}
```

[Copy to clipboard]

Initial object locations

The initial value of ($ has parent $) is computed by making a multi-query to the predicate at compile-time, and noting down the first parent encountered for each object.

The following definitions:

```
(edible #apple)
(edible #lettuce)
```

```
(edible #apple)
(edible #lettuce)
(#apple has parent #bowl)
(*(edible $) has parent #fridge)
```

[Copy to clipboard]

cause the multi-query to succeed three times, first with (#apple has parent #bowl), then with (#apple has parent #fridge), and finally with (#lettuce has parent #fridge). In the initial object tree, the apple will be located in the bowl, and the lettuce in the fridge.

Onwards to “Chapter 7: Syntactic sugar” • Back to the Table of Contents