Chapter 2: Objects and state

(Populating the game world • Descriptions, appearances, and synonyms • Defining new predicates • Object locations • Dynamic predicates • Hidden objects)

Populating the game world

Things in the game world are represented by objects (hashtags) in the Dialog language. Dialog objects are *thin*; they are identifiers without any code or data inside them. They exist, as names, simply by virtue of being mentioned somewhere in the source code. But they have no contents. Instead, the game world is modelled using predicates, where a predicate is a collection of rule definitions.

We will now extend our minimal example with two objects: A table (called #table) and a chair (#chair). We will give them names and descriptions, categorize them using traits (a kind of inheritable properties), and place them in their initial locations. All of this is achieved by defining rules:

```
%% Rules describing the table:

%% Rules describing the table:
(name #table)          wooden table
(descr #table)         It's a sturdy wooden table.
(supporter #table)
(#table is #in #room)

%% Rules describing the chair:
(name #chair)          chair
(descr #chair)         It's a plastic chair, painted white.
(on-seat #chair)
(#chair is #in #room)

%% These are the rules from the minimal story in the previous chapter:
(current player #player)
(#player is #in #room)
(room #room)
```

[Copy to clipboard]

Every rule definition starts with a *rule head* (the part within parentheses), which must begin at the leftmost column of a line in the source code. The rule head is optionally followed by a *rule body* (that's the non-parenthesized text in the above example), which may continue onto subsequent lines, as long as those lines are indented by at least one space or tab character.

A double-percent in the source code begins a *comment* that lasts to the end of the line.

The rule body is a piece of code that can be executed. Plain text is an instruction to print that text. Some special characters (#, $, @, ~, *, |, \, parentheses, brackets, and braces) need to be prefixed by a backslash (\). No special treatment is required for apostrophes or double quotes.

Other instructions may appear within parentheses in rule bodies: For instance, (line) inserts a line break, and (par) inserts a paragraph break. Text in the source code can be formatted freely, because it is broken down internally into a stream of words and punctuation characters, and then reassembled, with space characters automatically inserted in all the right places. This process eliminates the risk of duplicate or missing spaces, line breaks, and paragraph breaks.

```
(descr #table)
      It's a sturdy

(descr #table)
 It's a sturdy wooden table.
 %% The following instruction inserts a paragraph break (a blank line).
 (par)
 %% The following instruction would normally insert a line break, but it
 %% has no effect here because of the preceding paragraph break.
 (line)
 You got this table from your late aunt Margareth.
```

[Copy to clipboard]

The resulting object description would look like this:

```
> EXAMINE TABLE
```

It's a sturdy wooden table.

You got this table from your late aunt Margareth.

To reduce the repetitiveness of coding by rules, Dialog provides a bit of syntactic sugar: It is possible to set a current topic, which is an object name, by mentioning that object at the leftmost column of a line. Then, whenever an asterisk (*) appears in a position where an object name could go, it is understood as a reference to the current topic. This feature makes the code more concise, and easier to type out:

```
#table
(name *)
```

```
#table
(name *) wooden table
(descr *) It's a sturdy wooden table.
(supporter *)
(* is #in #room)

#chair
(name *) chair
(descr *) It's a plain chair, painted white.
(on-seat *)
(* is #in #room)

(current player #player)
(#player is #in #room)
(room #room)
```

[Copy to clipboard]

Thus, we have created two tangible objects in our room, given them source code names (#table, #chair), printed names (“wooden table”, “chair”), and descriptions to be revealed by the EXAMINE verb. We have also categorized the objects: The table is a *supporter*, which means that the player is allowed to put things on top of it. The chair is an *on-seat*, which is a special kind of supporter that the player may sit on. The standard library defines many other standard categories that may be used by stories, and we'll take a closer look at them in the upcoming chapter on traits. The predefined categories include *containers* and *in-seats* (such as armchairs), as well as *rooms*.

We have also defined an *initial location* for each object. A location has two parts: A *relation* (#in) and a *parent object* (#room). In addition to #in, the standard library supports the relations #on, #under, #behind, #heldby, #wornby, and #partof.

Descriptions, appearances, and synonyms

The (descr $) predicate prints the *external description* of an object; what it looks like when viewed from the outside. Another predicate, (look $), prints the *internal description* of an object, i.e. what it looks like from the inside. This is used for room descriptions.

Let's add a room description to our example game:

```
#player
(current player *)
```

```
#player
(current player *)
(* is #in #room)

#room
(name *) tutorial room
(room *)
(look *)  This is a very nondescript room, dominated by a wooden table.
          (notice #table)    %% Binds the word “it” to the table.

#table
(name *) wooden table
(descr *) It's a sturdy wooden table.
(supporter *)
(* is #in #room)

#chair
(name *) chair
(descr *) It's a plain chair, painted white.
(on-seat *)
(* is #in #room)
```

[Copy to clipboard]

Try this game! You can LOOK, and then X IT, and SIT (on what? Answer CHAIR).

Did you miss the chair in the room description? In Dialog, as a general design principle, game objects do not call attention to themselves. They are assumed to be part of the scenery, and it is up to the story author to mention (or subtly hint at) their existence in the prose, like we did with the table. Nevertheless, there is a way to furnish objects with *appearances*, which are displayed in separate paragraphs of their own after the room description:

```
#chair
(appearance *)
```

#chair
(appearance *) You notice a conspicuous chair here.
[Copy to clipboard]

The room description would then turn into:

> LOOK
Tutorial room
This is a very nondescript room, dominated by a wooden table.

You notice a conspicuous chair here.

Appearances can be very handy for objects that move around during gameplay. This includes objects that the player might pick up, and drop in another room. We will learn more about such objects—and appearances—when we get to the chapter about Items.

How does the game know that CHAIR refers to the chair object? By default, the standard library assumes that every word that appears in the printed name of an object, i.e. the body of the (name $) rule, can be used to refer to it. If the player types several words in succession, they must all refer to the same object, so WOODEN CHAIR does not match any object in this game. We can easily add extra synonyms to an object, using the (dict $) predicate:

```
(dict #chair)
white plain
```

(dict #chair) white plain
[Copy to clipboard]

Now the game would understand SIT ON THE WHITE CHAIR, for instance. Add some synonyms to the game and try them out!

What happens if you add “wooden” as a synonym for the chair, and type EXAMINE WOODEN? What about SIT ON WOODEN?

Noun phrase heads

To assist with disambiguation, it is also possible to declare certain words to be potential *heads* of the noun phrase for a given object. The head of a noun phrase is the main noun, such as “cap” in “the bright red bottle cap of doom”.

Thus, we might define:

```
#bottle
(name *)  red bottle
```

#bottle
(name *) red bottle
(dict *) crimson decanter
(heads *) bottle decanter

#cap
(name *) red bottle cap
(heads *) cap
[Copy to clipboard]

Now, if the player types EXAMINE BOTTLE, this is unambiguously interpreted as a request to examine the bottle, not the bottle cap, because one of the heads of #bottle was given. If the player types EXAMINE RED, the game will ask if they wanted to examine the red bottle or the red bottle cap. In response to that, the answer BOTTLE is unambiguously understood as the #bottle.

The list of noun heads is only consulted to resolve ambiguities. If the player attempts to TAKE BOTTLE while holding the bottle but not the cap, for instance, then that is interpreted as a request to take the bottle cap.

Typically, (heads $) definitions are added as needed, on a case-by-case basis, when ambiguities turn up during playtesting.

Defining new predicates

It's easy to conjure up new predicates. We simply define one or more rules for them. For instance, we might want to put the primary construction material of our objects in a separate predicate that we call (material):

```
(material)
caramelized sugar
```

(material) caramelized sugar
[Copy to clipboard]

This predicate can then be queried from within rule bodies, like so:

```
#table
(name *)
```

#table
(name *) (material) table
(descr *) It's a sturdy table made of (material).
[Copy to clipboard]

We can use variables to pass parameters around. In the following example, a generic object description calls out to object-specific material and colour predicates. The standard library doesn't know about these predicates; we just created them by defining rules for them. The library queries (descr $), and we take it from there:

```
#player
(current player *)
```

#player
(current player *)
(* is #in #room)
(descr *) It's you.

#room
(name *) tutorial room
(room *)
(look *) This is a very nondescript room, dominated by a wooden table. (notice #table)

#table
(name *) table
(material *) caramelized sugar
(colour *) dark brown
(supporter *)
(* is #in #room)

#chair
(name *) chair
(material *) plastic
(colour *) white
(on-seat *)
(* is #in #room)

(descr $Obj) It's (colour $Obj) and made of (material $Obj).
[Copy to clipboard]

Note that the rule for (descr #player), on line four, supersedes the generic rule for (descr $Obj), at the very last line. This is solely due to the order in which they appear in the source code. When coding in Dialog, make sure to always put specific rules before generic rules.

Variables are local to the rule definition in which they appear: In the example, $Obj is only available from within the last rule. If we were to use a variable named $Obj inside one of the other rules, that would be a completely unrelated variable.

Queries either fail or succeed. When a predicate is queried, each rule definition is tried in program order, until a match is found. If there is no matching rule, the query fails. As a general rule of thumb, predicates that print text should be designed to always succeed. Therefore, we'll often want to put a catch-all rule at the end of the program, with a wildcard ($) in the rule head:

```
(material $)
an unknown
```

(material $) an unknown material

```
(colour $)    beige
```
[Copy to clipboard]

The standard library provides a catch-all rule for (descr $), printing a stock message along the lines of “It seems to be harmless”. But for our material and colour predicates, we have to provide our own catch-all rules.

Object locations

At runtime, objects of the game world are organized into *object trees*. Every object has (at most) one parent, and a relation (in, on, under, behind, held by, or worn by) to that parent. The *root* of a tree has no parent. Sometimes you will run into the expression “the object tree of the game”, as though every object were part of a single, huge tree with only one root. Technically, it is more correct to say “the object forest of the game”, because there can be more than one root object. Rooms don't have parents, so every room is the root of a tree.

In the Dialog standard library, the object forest is encoded using two predicates: ($Object has parent $Parent) and ($Object has relation $Relation). For brevity, there's an access predicate that combines them into a single expression: ($Object is $Relation $Parent). We have already seen that one defines the initial location of an object like this:

```
(#chair is #in #room)
```

```
(#chair is #in #room)
```
[Copy to clipboard]

and, due to the access predicate, the above is equivalent to the following pair of definitions:

```
(#chair has parent
#room)
```

```
(#chair has parent #room)
(#chair has relation #in)
```
[Copy to clipboard]

From within a rule body, you may query the access predicate to determine the current location of an object:

```
(descr #apple)
      The apple
```

```
(descr #apple)
 The apple
 (if) (#apple is #in $Parent) (then)
 in (the $Parent)
 (endif)
 looks yummy!
```
[Copy to clipboard]

The output of the above code might be:

The apple in the fruit basket looks yummy!

With a multi-query, you can backtrack over every object that has a particular location:

```
(descr #basket)
      It's a plain fruit
```

```
(descr #basket)
 It's a plain fruit basket.
 (exhaust) {
 *($Child is #in $Obj)
 There's (a $Child) inside.
 }
```
[Copy to clipboard]

Dynamic predicates

The location of an object is dynamic, which means that it can be modified at runtime using the (now) keyword:

```
(descr #apple)
      (if) (#apple is
```

```
(descr #apple)
 (if) (#apple is #in #basket) (then)
```

```
 Yummy!
 (else)
 The apple seems to be very shy. As soon as you look at it, it
 jumps of its own accord into the fruit basket.
 (now) (#apple is #in #basket)
 (endif)
```
[Copy to clipboard]

The standard library also uses dynamic predicates to track the *internal state* of game world objects. For instance, ($ is closed) succeeds when a particular openable object (such as a container, or a door) is currently closed. The access predicate ($ is open) is defined as its negation, ~($ is closed), allowing both forms to appear as queries, now-expressions, and initial value definitions.

A convenience predicate, (open or closed $), prints the word “open” if the given object is open, and “closed” otherwise. The same thing can of course be coded explicitly with an if statement.

```
#box
(openable *)
```

```
#box
(openable *)
(* is closed)
(descr *)
 The box is (open or closed *).
(intro)
 Pandora looks at her box. (descr #box)
 (now) ~(#box is closed)
 She looks away for five seconds, and then looks at it again,
 just to check. (descr #box)
 (game over { That's just life. })
```
[Copy to clipboard]

The following dynamic predicates are used by the library:

Dynamic predicate	**Negated version**	
($ is closed)	($ is open)	Changed by: OPEN, CLOSE.
($ is locked)	($ is unlocked)	Changed by: LOCK, UNLOCK.
($ is off)	($ is on)	Changed by: SWITCH ON, SWITCH OFF.
($ is broken)	($ is in order)	Never changed by the library.
($ is handled)	($ is pristine)	Object has been moved by the player.
($ is visited)	($ is unvisited)	The player has been inside this room.
($ is hidden)	($ is revealed)	Not suggested during disambiguation.

Hidden objects

Hidden objects, ($ is hidden), can be in scope (meaning that the parser will recognize them as nouns), but the library is careful not to mention them. Thus, if the player carries a pink slip, and the current room contains a pink elephant that is hidden, then EXAMINE PINK will print the description of the slip, with no disambiguating questions asked. EXAMINE PINK ELEPHANT will examine the elephant, as would EXAMINE PINK if the slip weren't there. The idea is to improve the experience of replaying a game, while avoiding spoilers on the first playthrough.

Hidden objects can be revealed either by directly updating the flag, (now) (#elephant is revealed), or by querying (reveal $) for the given object. A hidden object is also revealed implicitly when its name is printed, or when (notice $) is invoked on it.

Onwards to “Chapter 3: Traits” • Back to the Table of Contents