Chapter 1: Flow of execution

(Predicates and rules • Printing text • Parameters, objects, and wildcards • Success and failure)

Dialog is heavily inspired by *Prolog*. Unless you've programmed in Prolog before, you may find the material in the first three or so chapters to be increasingly counter-intuitive, odd, or even downright crazy. But don't worry, we'll take it in small steps, and hopefully the concepts will click together in the end. If at any point you feel that the discussion is too abstract, don't hesitate to head over to Part II for a while, and then come back.

If you already know Prolog, you're still going to have to pay attention to the details, as there are both obvious and subtle differences between the two languages.

Predicates and rules

This is what “hello world” looks like in Dialog:

```
(program entry
point)
```

```
(program entry point)
  Hello, world!
```

[Copy to clipboard]

A Dialog program is a list of *rule definitions*, or *rules* for short. A rule definition always begins at the very first column of a line, but it may span several lines. Subsequent lines of code that belong to the same rule have to be indented by at least one space or tab character.

There can be multiple rules with the same name. If so, they are said to belong to the same *predicate*. The program above contains one predicate, built from a single rule definition. The head of the rule is (program entry point), and this is also the name of the predicate. The body of the rule is some text to be printed verbatim, in this case “Hello, world!” Program execution always begins with the predicate called (program entry point). Sometimes we will refer to this as the top-level predicate.

A predicate may call upon another predicate, similar to subroutine calls in other programming languages. This is referred to as making a *query*. The following program produces exactly the same output as the previous one:

```
(program entry
point)
```

```
(program entry point)
  Hello (my own rule)!

(my own rule)
  , world
```

[Copy to clipboard]

The parentheses indicate that a query is to be made. When a query is made to a predicate, all rules defined for that predicate are consulted in program order, until one of them *succeeds*; we will return to the concept of success at the end of the present chapter. If the program comprises several source code files, the order of the rules is determined by the order in which the filenames appeared on the command line.

Predicate names are case-sensitive.

Printing text

Rule bodies contain statements to be executed. Plain text in a rule body 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.

Comments in the source code are prefixed with %% and last to the end of the line.

Thus:

```
(program entry
point)
```

```
(program entry point)
  Hello y'all \( and "welcome" \) !     %% This is a comment.
```

[Copy to clipboard]

produces the output:

Hello y'all (and "welcome")!

Dialog is smart about punctuation and whitespace: It interprets the source code text as a stream of words and punctuation characters, and rebuilds the output from that stream, inserting whitespace as appropriate. It knows that no space should precede a comma, that a space should go before an opening parenthesis but not after it, and so on.

```
(program entry
point)
```

```
(program entry point)
 For instance
 :This
 text
 \( which,to all intents          and purposes,is silly\( indeed
 \) \)
 , prints properly.
```

[Copy to clipboard]

The output of that is:

For instance: This text (which, to all intents and purposes, is silly (indeed)), prints properly.

It is possible to override the automatic whitespace decisions on a case-by-case basis using the *built-in predicates* (space) and (no space):

```
(program entry
point)
```

```
(program entry point)
 To (no space) gether (space) , apart.
```

[Copy to clipboard]

The output of that is:

Together , apart.

Line breaks are inserted with the (line) built-in predicate; paragraph breaks with (par). On the Z-machine, a paragraph break is implemented as a blank line of output.

Several adjacent line breaks are merged into one, preventing accidental paragraph breaks. Likewise, several adjacent paragraph breaks are merged into one, along with any adjacent line breaks, preventing accidental runs of more than one blank line.

The following program:

```
(note)
      (line) This goes
```

```
(note)
 (line) This goes on a line of its own. (line)
(program entry point)
 (note)
 (note)
 (par) This goes in a paragraph of its own. (par)
 This
 is
 not broken up.
 (note)
```

[Copy to clipboard]

produces the following output:

This goes on a line of its own.
This goes on a line of its own.

This goes in a paragraph of its own.

This is not broken up.
This goes on a line of its own.

Parameters, objects, and wildcards

Predicates can have *parameters*. The name of a predicate, called its *signature*, is written with dollar signs as placeholders for the parameters. These may appear anywhere in the predicate name. For instance, (descr $) is the signature of a predicate with one parameter, and in this case the parameter is at the end.

In rule heads and queries, *values* may appear in place of these dollar signs. A common kind of value is the *object*. Objects in Dialog are short, programmer-friendly names that start with a # character and refer to entities in the game world.

Consider the following three rule definitions for the (descr $) predicate:

```
(descr #apple) The apple looks yummy.

(descr #apple) The apple looks yummy.
(descr #door)  The oaken door is oaken.
(descr $)      It looks pretty harmless.
```

[Copy to clipboard]

When a value appears inside a rule head, the given parameter must have that particular value in order for the rule to succeed. It is also possible to use dollar signs as wildcards in rule heads.

Let's take a look at how the three rule definitions above might be used in a program. To print the description of an object, let's say the #door, one would make the query (descr #door). Dialog would consider each of the three rule definitions in program order. The first rule head doesn't match the query, but the second does. Thus, the text “The oaken door is oaken.” is printed. The query (descr #orange) would cause the text “It looks pretty harmless.” to be printed.

Note that the general rule, the one with the wildcard, appears last. This is crucial: If the general rule were to appear before e.g. the #door rule in the source code, it would supersede that rule every time, and the door would be described as harmless.

Signatures (predicate names) rarely appear explicitly in the source code. They are implied by rule heads and queries, where parameter values are typically used instead of dollar signs.

Objects are thin

Dialog objects are *thin*, in the sense that each hashtag is a mere identifier, without any inherent behaviour or properties. This is in contrast with object-oriented programming languages, where code and data are organized inside objects and classes. In Dialog, the world is modelled using predicates that specify relations between objects, but the objects themselves are just names.

Object names may contain alphanumeric characters (including a limited range of international glyphs), plus (+), minus (-), and underscore (_) characters. They are case sensitive.

Success and failure

If a query is made to a predicate, but there is no matching rule in the program, the query *fails*. When a rule makes a query, and that query fails, the rule also fails and is immediately abandoned. In this way, the failure condition might propagate to the calling rule, to its calling rule in turn, and so on, all the way to the top-level predicate. Here is a simple program that fails:

```
(program entry point)

(program entry point)
        You see an orange. (descr #orange) Now what do you do?
(descr #apple) The apple looks yummy.
(descr #door)  The oaken door is oaken.
```

[Copy to clipboard]

This program will print “You see an orange”. Then, because there is neither a rule for (descr #orange) nor a rule for (descr $), the query (descr #orange) fails. This causes the top rule, i.e. the program entry point, to fail, at which point the entire program terminates. Hence, “Now what do you do?” is never printed.

If failure would always propagate all the way to the top and terminate the program, it would be of little use. So of course, there's more to the story: Recall that a query to a predicate causes each of its rule definitions to be tried, in source code order, until a match is found. What happens when a rule fails, is that this search continues where it left off. Consider the following example:

```
(program entry point)

(program entry point)
 (descr #apple)
```

Over and out.

(descr #apple)
(the player dislikes #apple)
Yuck!

(descr $)
It looks yummy!

(the player dislikes #orange)
[Copy to clipboard]

A query is made: (descr #apple). There's a matching rule, and this rule makes a query in turn, to the predicate (the player dislikes $), with the parameter #apple. But this time, there is no matching rule definition, so the query fails. This aborts the execution of the (descr #apple) rule, and the quest to satisfy the original query, (descr #apple), resumes. And indeed there's another match: (descr $) prints “It looks yummy!” and succeeds. Thus, the program entry point rule will proceed to print “Over and out”.

The complete output is:

It looks yummy! Over and out.

Onwards to “Chapter 2: Manipulating data” • Back to the Table of Contents