Chapter 2: Manipulating data

(Local variables • Values • Unification • Partial lists and recursion)

Local variables

Recall that a dollar sign on its own is a wildcard. However, a dollar sign that is immediately followed by a word, with no whitespace in between, is a *local variable*. When incoming parameters have names, it becomes possible to pass them on to subqueries.

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

```
(program entry
point)
```

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

(descr $Thing)
                %% Here, $Thing is a variable that is passed to another query.
                (The $Thing) looks yummy. (line)

(The #apple) The green apple
(The #pear)  The juicy pear
(The $)      That    %% Here, $ is a wildcard. Its value is ignored.
```

[Copy to clipboard]

Since there is no explicit rule definition for (The #orange), the program will print:

```
The green apple looks yummy.
That looks yummy.
The juicy pear looks yummy.
```

It is also possible to print the value of a variable. In the case of objects, this will print the actual hashtag as it appears in the source code, so it is mainly useful for debugging:

```
(program entry
point)
```

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

(descr $Tag)
 No description for $Tag!
```

[Copy to clipboard]

This will print:

```
No description for #apple!
```

With the help of variables, we can refine the process of rule matching. Recall that when a query is made, Dialog considers every rule in program order, and attempts to match the parameters. If the match is successful, the rule body starts to execute. In case of a failure, Dialog proceeds to try the next rule in the program.

Ordinary parameter matching is a rather blunt instrument, so queries (sometimes called *guard conditions*) can be placed at the very beginning of a rule body to perform more sophisticated checks. If the guard condition succeeds, the rule applies, otherwise the search continues. Here is a very simplistic approach to world modelling:

```
%% A rule with a
blank body will
```

```
%% A rule with a blank body will succeed (assuming the parameters match).
%% The (fruit $) predicate will succeed for #apple and #orange, but fail for
%% anything else.

(fruit #apple)
(fruit #orange)

(descr #door)          The oaken door is oaken.
(descr $Obj)           (fruit $Obj) Yummy!
```

```
(descr $)                   It seems harmless.
(program entry point)

                            Apple: (descr #apple) (line)
                            Door: (descr #door) (line)
                            Pencil: (descr #pencil) (line)
```

[Copy to clipboard]

The output is:

```
Apple: Yummy!
Door: The oaken door is oaken.
Pencil: It seems harmless.
```

The scope of a local variable is limited to the rule in which it appears.

Values

So far, we have seen one kind of value: the object. There are three more kinds of value in the Dialog programming language: *Number*, *dictionary word*, and *list*.

Number

A number is a non-negative integer in the range 0–16383 inclusive. The printed representation of a number is always in decimal form, with no unnecessary leading zeros. Numbers that appear in the source code must also adhere to this format: For instance, 007 in the source code is regarded as a word of text. This makes a difference inside rule heads and queries, where 007 would be considered part of the predicate name, and not a parameter.

Dictionary word

While objects represent elements of the game world, dictionary words represent input typed by the player. Dictionary words are prefixed with @ instead of #. Unlike object names, they are case-insensitive.

A dictionary word is internally separated into two parts: an *essential part* at the beginning of the word, and an *optional part* at the end. When two dictionary words are compared to each other, only the essential part is considered.

Usually, the optional part is blank. However, on the Z-machine, long dictionary words are split according to an internal, low-level dictionary format. This usually means that the first nine letters of the word are essential, while the rest are optional. For instance, @northeast and @NorthEastern are considered to be the same value. Non-alphabetical characters require multiple slots of storage in the Z-machine, so that e.g. @north-east is considered the same as @north-ea.

The printed representation of a dictionary word is the essential part followed by the optional part. No @ prefix is printed, and there is usually no visible seam between the two parts of the word. However, during tracing, dictionary words are displayed with a plus sign (+) separating the essential part from the optional part.

Dictionary words may contain any characters supported by the underlying platform, e.g. the *ZSCII character set* in the case of the Z-machine.

A handful of characters are used to separate words when parsing input. These cannot appear together with other characters in a dictionary word—they have to stand on their own, as a single-character word. They are: . , ; * " ( )

Characters that have special significance in Dialog source code, such as parentheses, can be escaped by a backslash:

```
        @\(
```

```
 @\(
```

[Copy to clipboard]

Dictionary words are case insensitive as a general rule, although the Z-machine compiler backend has limited support for case conversion of international characters.

List

A list is an ordered sequence of values. Lists are enclosed in square brackets in the source code:

```
(program entry
point)
```

```
(program entry point)
 This is a list containing three integers: [1 2 3]
 This is an empty list: []
 A list may contain values of different kinds: [@hello #world [2 3] 4]
```

[Copy to clipboard]

In the final expression above, the third element of the list is itself a list, containing the two values 2 and 3.

When a dictionary word appears inside a list, it is possible (and recommended) to omit the @ character.

Printing a list is possible, and highly useful during debugging. The program:

```
(program entry
point)
```

```
(program entry point)
 Have a look at [#this inscrutable list]!
```

[Copy to clipboard]

will print out:

```
Have a look at [#this inscrutable list]!
```

Unbound variables

In addition to the four kinds of value we have seen (object, dictionary word, number, and list), there is a fifth kind, which is more of a pseudo-value: the *unbound variable*. If a local variable appears in a rule body without first being mentioned in the rule head, for instance, it will be unbound. Unbound variables are allowed wherever values are allowed. They can appear as parameters to queries, and even inside lists.

The wildcard that we saw earlier ($) is in fact an unbound variable, although it also has the special property of being anonymous: Two instances of $ do not refer to the same variable.

The printed representation of an unbound variable is always $, regardless of its name.

```
(program entry
point)
```

```
(program entry point)
 This list contains an unbound variable: [one $Two three]
```

[Copy to clipboard]

The output is:

```
This list contains an unbound variable: [one $ three]
```

Unification

At the heart of Dialog is a mechanism called *unification*. This is an operation that takes two values, and ensures that they are the same afterwards. If this cannot be done, the operation fails.

Unification is provided by a built-in predicate in Dialog, with the signature ($ = $). The equals sign has no special properties; it is treated as a regular word.

Two identical values, e.g. #apple and #apple, unify successfully. Two values that differ, e.g. #apple and #orange, do not unify, so the operation fails. Thus, unification can be used to check for equality:

```
(program entry
point)
```

```
(program entry point)
 (#apple = #apple)
 This text will be printed.
 (#apple = #orange)
 This will not, because the rule has failed by now.
```

[Copy to clipboard]

An unbound variable successfully unifies with any value, but this also has the effect of *binding* the variable to that value. Thus:

```
(program entry
point)
```

```
(program entry point)
 ($X = #apples)
 (#oranges = $Y)

 I like $X and $Y.
```

[Copy to clipboard]

will print:

I like #apples and #oranges.

Observe that the same operation that was used to check for equality can be used for assignment. It is also symmetrical: The unbound variable can appear either to the left or to the right of the equals sign.

But once the variable is bound, it sticks to that value:

(program entry point)

```
(program entry point)
 ($X = #apples)
 I like $X
 ($X = #oranges)
 and $X.
```

[Copy to clipboard]

will print:

I like #apples

and then the second unification fails, because $X resolves to #apples, which is different from #oranges. As a result, the top-level predicate of this example fails, and the program terminates.

Two lists unify if their elements unify, at each and every position. This may have the side-effect of binding unbound variables inside the lists. Consider:

(program entry point)

```
(program entry point)
 ($X = [#apples #pears $])
 ($X = [$ #pears #oranges])
 I like $X.
```

[Copy to clipboard]

The first unification operation will bind $X to the list [#apples #pears $], the last element of which is an anonymous unbound variable. The second unification operation will attempt to unify that list with another list, [$ #pears #oranges]. This will succeed, and by now three bindings are in place: The first anonymous variable is bound to #oranges, the second anonymous variable is bound to #apples, and $X is bound to [#apples #pears #oranges]. The output of the program is:

I like [#apples #pears #oranges].

Finally, it is possible to unify two unbound variables with each other. This creates a hidden link between them, entangling them, so that if one of them is later bound to a value, the other one will also become bound to the same value:

(program entry point)

```
(program entry point)
 ($X = $Y)
 ([spooky action at a distance] = $X)
 This is $Y.
```

[Copy to clipboard]

The output of the program is:

This is [spooky action at a distance].

Parameters are passed by unification

Now, here's the kicker: In Dialog (and Prolog, for that matter), parameters to predicates are passed by unification. Remember that when a predicate is queried, the query is compared to each of the rule heads, in program order, until a match is found. That comparison is in fact carried out by attempting to unify each parameter of the query with the corresponding parameter of the rule head.

This has a very interesting and useful consequence, which is that parameters can be used interchangeably as inputs or outputs:

```
(#rock beats
#scissors)
```

```
(#rock beats #scissors)
(#scissors beats #paper)
(#paper beats #rock)

(program entry point)
 (#rock beats $X)    %% Parameters are: Input, output.
 When your opponent plays rock, you'd better not play $X.
 ($Y beats #rock)    %% Parameters are: Output, input.
 When your opponent plays rock, you should play $Y.
```

[Copy to clipboard]

The first query (#rock beats $X) tells Dialog to search for a rule head with the signature ($ beats $), and attempt to unify #rock with the first parameter in the rule head, and the unbound variable $X with the second. This succeeds on the very first rule encountered, and as a side effect, $X is now bound to #scissors. That rule has no body, so it succeeds, and control returns to the top-level predicate.

For the second query ($Y beats #rock), Dialog searches for a rule head with the signature ($ beats $), and attempts to unify the unbound variable $Y with the first parameter, and #rock with the second. It gets to the first rule: $Y would unify successfully will #rock from the rule head, because $Y is unbound. But #rock does not equal #scissors, so this unification fails. Hence, the first rule was not a match. Now the second rule is considered: $Y is still unbound, and would unify perfectly fine with #scissors. But #rock from the query doesn't unify with #paper from the rule head, so the unification operation fails again. Finally, Dialog tries the third rule: $Y unifies successfully with #paper, and #rock unifies with #rock. This time the operation is successful! The rule body is empty, so it succeeds too, and control returns to the top-level predicate.

The output of the program is:

When your opponent plays rock, you'd better not play #scissors. When your opponent plays rock, you should play #paper.

Partial lists and recursion

Dialog has a special syntax for matching the head (first element) and tail (remaining elements) of a list. Recall that a list is usually a sequence of values in brackets:

```
    [1 2 3 4]
```

```
 [1 2 3 4]
```

[Copy to clipboard]

Such a sequence can be unified with a special expression, called a *partial list*:

```
    [$Head | $Tail]
```

```
 [$Head | $Tail]
```

[Copy to clipboard]

The unification succeeds if $Head unifies with the first element of the list, and $Tail unifies with a list containing the rest of the elements. Unification of a partial list and an empty list fails, because there is no head element to extract. Here is an example of extracting the head and tail of a list:

```
(program entry
point)
```

```
(program entry point)
 ([1 2 3 4] = [$A | $B])
 A is $A.
 B is $B.
```

[Copy to clipboard]

Here is the output:

A is 1. B is [2 3 4].

Unification works both ways, so the same syntax can be used to construct a list from a head and a tail:

```
(program entry
point)
```

```
(program entry point)
 ($A = 1)
 ($B = [2 3 4])
 Tacking on a new head: [$A | $B]
```
[Copy to clipboard]

The output is:

```
Tacking on a new head: [1 2 3 4]
```

The syntax is not limited to just a single head: Any number of elements can be matched (or tacked on) from the beginning of the list. However, the tail is always represented by a single expression (usually a variable) after the vertical bar (|). So:

```
(program entry point)
 ([$First $Second | $Rest] = [a b c d e])
 ([$Second $First | $Rest] = $Result)
 The result is $Result.
```
[Copy to clipboard]

produces the following output:

```
The result is [b a c d e].
```

Partial lists are very useful in combination with *recursive code*, i.e. predicates that query themselves. Here is an example that considers each element of a list in turn using recursive calls:

```
(program entry point)
        (observe objects [#banana #orange #apple #apple])

(observe objects [])
        You don't see any more fruit.

(observe objects [$Head | $Tail])
        You see (a $Head). (line)
        (observe objects $Tail)

(a #banana) a banana
(a #apple)  an apple
(a $)       an unknown fruit
```
[Copy to clipboard]

The output is:

```
You see a banana.
You see an unknown fruit.
You see an apple.
You see an apple.
You don't see any more fruit.
```

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