Chapter 9: Beyond the program

Story metadata

When an interactive story is released into the wild, the community takes over, cataloguing it, preserving it for posterity, and making it available via archives and databases. To help streamline this work, authors are strongly encouraged to provide *metadata*, such as the title of the work and the name of the author, in a machine-readable format inside the story file.

Dialog strives to comply with the *Treaty of Babel*, which is an initiative to establish a common standard for interactive-fiction metadata. The information is supplied via a special set of predicates, that are queried at compile-time.

Each published story should at the very least include a unique identifier—called the *IFID*—to distinguish it from other works. Different versions of the same story (e.g. bug-fix releases) should be marked with the same IFID. To declare the IFID, use the following predicate:

```
(story ifid)
XXXXXXXX-XXXX-
```

(story ifid) XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
[Copy to clipboard]

where each X is an uppercase hexadecimal digit. There's an online tool on the Dialog website that lets you generate such declarations with fresh IFID numbers. You may also use any program capable of generating a universally unique identifier (UUID), such as the uuidgen command under Linux (available in the Debian package uuid-runtime).

The Dialog compiler will complain if the story contains more than one hundred lines of source code and no IFID is provided. The standard library is not included in the line count.

A story author may also want to specify one or more of the following:

(story title)

Put the title of the story in the body of this rule definition.

(story author)

The full name of the author, or a pseudonym, should go here.

(story blurb)

Put a brief, selling description of the story here.

(story noun)

The default story noun is “An interactive fiction”.

(story release $)

Specify the release number in the parameter; the rule body should be empty.

The story blurb is allowed to contain paragraph breaks, (par), but otherwise these rule bodies should consist entirely of plain, static text. The predicates are queried at compile-time, and the text is extracted and stored in the relevant locations in the output files. You may also query these predicates from your own code; the standard library invokes several of them while printing the banner, for instance.

When compiling to the zblorb file format, it is possible to include cover art with your story. This is not part of the Dialog language itself, but is an extra service provided by the Dialog compiler. Specify the name of a PNG file and, optionally, a short textual description of the image, using the commandline options -c and -a respectively. The cover image should be no larger than 1200 x 1200 pixels in size, and preferably square.

The raw z8 format has limited support for metadata, so if you select this output format, only the IFID and release number will be included in standard locations in the file. The other predicates are of course still accessible from within the program.

Two elements of metadata are supplied automatically by the compiler: The *compiler version string* and the *serial number*. These can be printed from within the program (and you are strongly encouraged to do so as part of the story banner), by querying the following built-in predicates:

(serial number)

(compiler version)

The serial number is the compilation date in YYMMDD format.

The standard library defines a rule for (library version), which prints the library version string. The compiler looks for this definition to check that a library appears as the last filename on the commandline, and prints a warning otherwise.

Interfaces

Predicates are versatile building blocks with many potential usage patterns. Their parameters can be inputs, outputs, or both, and they can be bound, unbound, or partially bound (i.e. lists with unbound variables inside).

Furthermore, the behaviour of a predicate can depend on multiple rule definitions, and some of the rules that deal with special cases might rely on pattern matching in the rule heads. Thus, even with access to source code, it may not be obvious how a predicate is supposed to be used. To alleviate this problem, Dialog lets you document the intended use of a predicate with an *interface declaration*:

```
(interface
(<i>predicate
```

(interface (*predicate name*))
[Copy to clipboard]

In an interface declaration, all parameters should be variables with sensible, self-explaining names. Furthermore, the first character of the variable name has special significance:

<

This parameter is supposed to be fully bound **before** querying the predicate.

>

This parameter is supposed to be fully bound **after** the query succeeds.

Here is an example:

```
(interface
(understand
```

(interface (understand $<Words as $>Action))
[Copy to clipboard]

From the above declaration, we learn that the predicate is supposed to be queried with a bound value—some words, presumably in a list—as the first parameter. When it succeeds, it will have unified the second parameter with a bound value, representing an action.

Note that < and > have no special significance in variable names in ordinary rule definitions, only in interface declarations.

Being bound before or after a query is subtly different from being an input parameter or an output parameter. For instance, the library provides a predicated called (length of $ into $) that counts the elements in a list. The list is allowed to contain unbound values, so the first parameter of the interface declaration cannot begin with a <, even though it is an input. The actual declaration is:

```
(interface (length of
$List into
```

(interface (length of $List into $>Number))
[Copy to clipboard]

Interface declarations are machine-readable documentation. They do not affect the generated story file in any way. However, the compiler will check the program for interface violations—unbound values that could end up in the wrong place at the wrong time—and warn you about them at compile-time.

Runtime errors

Attempts to violate the constraints imposed on dynamic predicates will result in fatal runtime errors.

Such errors can also occur if one of the heaps (main, auxiliary, and long-term) is ever exhausted. The heaps are three relatively large memory arrays that the Dialog runtime system uses to keep track of local variables, lists, activation records, choice points and other transient data structures. When one of these arrays fills up—which could happen at any time, in the middle of any operation—there's not much Dialog can do, except abandon all hope and re-initialize itself.

For compiled code, the main heap, auxiliary heap, and long-term heap occupy 1000, 500, and 500 words respectively by default, and this should be more than enough for most interactive stories. The size of each area can be adjusted by passing commandline options to the compiler. A built-in predicate, (display memory statistics), prints the peak usage of

each memory area so far. During debugging and testing, you may wish to invoke this predicate just before quitting, as it will tell you how close you are to the limits.

The main rationale for throwing runtime errors when an invalid (now) operation is attempted, instead of merely failing, is that the compiler can do a better job of optimizing the code if it can assume that now-expressions always succeed.

A fatal runtime error will reset the Dialog evaluation state, clear the heaps, and restart the program from the (error $ entry point) predicate. The parameter is a small integer, representing the nature of the error:

1: Heap space exhausted.

2: Auxiliary heap space exhausted.

3: Type error: Expected object.

4: Type error: Expected bound value.

5: Invalid dynamic operation.

6: Long-term heap space exhausted.

7: Invalid output state.

After a fatal error, the game world could be in an inconsistent state, and there's not much one can do except print an error message and quit. Or is there? The standard library attempts to bring the game back to a known state via the undo facility of the Z-machine.

Some notes on performance

Both the Dialog language and its compiler have been designed with runtime performance in mind. In particular, the language lends itself well to static analysis, and the compiler performs a thorough global analysis of the program in order to choose the ideal way to represent each predicate. Often, what looks like a lengthy search through a series of rule heads can compile down to a simple property lookup or a bunch of comparison instructions.

Execution speed

The programmer can lay the foundation for a well-optimized program, by following two important design principles.

The first principle is: Always dispatch on the first parameter. Whenever a bound value is used to select among many different rule definitions, try to phrase the name of the predicate so that the bound value is the first parameter. That's because the Dialog compiler considers the parameters in left-to-right order when generating accelerated code for rule lookups. An example of a carefully named predicate is (from $Room go $Direction to $Target) from the standard library, which is always queried with a known $Room.

The second principle is: Make tail calls. The last statement of a rule body is said to be in tail position. Making a query in tail position is cheaper than making it elsewhere. Thus, the following implementation:

```
(say it with $Obj)
My hovercraft is full
```

```
(say it with $Obj) My hovercraft is full of (name $Obj).
(stash $Obj)       (say it with $Obj) (now) ($Obj is #in #hovercraft)
```

[Copy to clipboard]

is not as efficient as this one:

```
(say it with $Obj)
My hovercraft is full
```

```
(say it with $Obj) My hovercraft is full of (name $Obj).
(stash $Obj)       (now) ($Obj is #in #hovercraft) (say it with $Obj)
```

[Copy to clipboard]

In the first version, the code for (stash $) needs to create an activation record, and set things up so that execution can resume at the (now) statement, with $Obj still bound to the correct value, after the inner query succeeds. In the second version, there is no need for an activation record: The (now) statement is handled locally, and then control is simply passed to (say it with $Obj).

If the final statement of a rule is a disjunction or an if-statement, then the last statement inside every branch is also in tail position.

It is especially fruitful to place recursive calls in tail position, as we can then avoid creating an activation record for every step in the recursion.

Memory footprint

Per-object variables have a relatively large memory footprint. If you would like your game to be playable on vintage hardware, try to minimize the number of per-object variables used by your design, especially if they will remain unset for most objects.

For instance, looking back at the (#troll wields #axe) example from the chapter on dynamic predicates, if the troll and the player are the only characters that wield weapons, it would be much better to use a pair of global variables:

```
(global variable (troll wields $))
(global variable (player wields $))
```

[Copy to clipboard]

If desired, wrapper predicates could be defined for querying or updating those variables as though they were per-object variables:

```
(#troll wields $Weapon)
 (troll wields $Weapon)
(#player wields $Weapon)
 (player wields $Weapon)
(now #troll wields $Weapon)
 (now) (troll wields $Weapon)
(now #player wields $Weapon)
 (now) (player wields $Weapon)
```

[Copy to clipboard]

To conserve heap memory, use backtracking whenever possible. Identify places in the code where you can implement loops with backtracking instead of recursion. For instance, to iterate over a list and print something for each element, use exhaust:

```
(exhaust) {
 *($X is one of $List)
 (report on $X)
 }
```

[Copy to clipboard]

Tail-call optimization is backend-dependent. This means that infinite loops must be implemented using (repeat forever). If they are implemented using recursion, some memory might leak with every iteration, and the program will eventually crash.

Asymptotic complexity of per-object flags

Per-object flags are implemented in one of two ways. Most flags are stored in some array-like structure indexed by object number, which means that checking or updating the flag is a constant-time operation. Some flags are also tracked by a separate data structure on the side: A single-linked list of all objects for which the flag is currently set.

The linked list is enabled for any flag predicate that the program might query with an unbound parameter, as this allows Dialog to loop efficiently over every object for which the flag is set. Such per-object flags can also be checked or set in constant time, but **clearing them is linear** in the number of objects having the flag set, because it is necessary to traverse the list at runtime in order to unlink the object in question.

The linked list is also enabled for flags that the program might clear for every object, e.g. (now) ~($ is marked). That operation is always linear in the number of objects that have the flag set.

Limitations and the future of Dialog

The support for numerical computation in Dialog is quite minimal at the moment. A future version of Dialog might include more built-in predicates, and an extended numerical range. However, this must be balanced against the stated design goal of a small, elegant language.

There is currently no way to provide a fallback if the interpreter fails to print a particular unicode character, although the Z-machine backend has built-in fallbacks for certain common characters. Unsupported characters come out as question marks, which is decidedly ugly. A future version of the language may introduce functionality for dealing with unsupported characters.

The (uppercase) built-in predicate is currently limited to English letters (A–Z) for the Z-machine backend. It should at least be upgraded to support the so called default extra characters in the ZSCII character set, which cover the needs of several European languages. The Å-machine backend and the debugger have full support.

Onwards to “Appendix: Quick reference” • Back to the Table of Contents