# Chapter 4: More control structures

(If-statements • Negation • Selecting among variations • Closures • Stoppable environments)

## If-statements

We have seen how execution can be directed into different branches based on arbitrary conditions, with the help of conjunctions and disjunctions. It is certainly possible to code up a simple if-then-else routine using these constructs alone:

```
(eat $Obj)
      You take a
```

```
(eat $Obj)
 You take a large bite, and conclude that it is
 {
 { (fruit $Obj) (or) (pastry $Obj) }
 sweet
 (or)
 ($Obj = #steak) (player eats meat)
 savoury
 (or)
 inedible
 }.
```

[Copy to clipboard]

But this quickly becomes unreadable as the code grows in complexity. Dialog provides traditional if, then, elseif, else, and endif keywords:

```
      (if)
<i>condition</i>
```

```
(if) condition (then)
statements
(elseif) other condition (then)
statements
(elseif) other condition (then)
statements
...
(else)
statements
(endif)
```

[Copy to clipboard]

The body of code between (if) and (then) is called a *condition*. If the condition succeeds, possibly binding variables in the process, then any choice points created by the condition are discarded, and execution proceeds with the statements in the corresponding then-clause. Should the condition fail, no variables are bound, and Dialog considers the next (elseif) condition, and so on, eventually falling back on the else-clause if none of the conditions were successful. The (elseif) and (else) clauses are optional; if there is no else clause, it is assumed to be empty (i.e. succeeding). The entire if-statement succeeds if and only if the chosen branch succeeds. Note that if the chosen (then) or (else) block creates choice points, those remain in effect. It is only the conditions that are limited to a single solution.

```
(eat $Obj)
      You take a
```

```
(eat $Obj)
 You take a large bite, and conclude that it is
 (if) (fruit $Obj) (or) (pastry $Obj) (then)
 sweet
 (elseif) ($Obj = #steak) (player eats meat) (then)
 savoury
 (else)
 inedible
 (endif).
```

[Copy to clipboard]

There are subtle differences between the if-statement above and the disjunction shown earlier: An if-condition is evaluated at most once, even if it creates choice points. As soon as one of the then-clauses is entered, all the remaining then-clauses and the else-clauses become inaccessible. And, finally, if-statements without else-clauses succeed when none of the conditions are met (i.e. a blank else-clause is assumed).

In the disjunction-based version of the rule, there are several lingering choice points, so if a failure is encountered further down in the rule (or even in the calling rule, if this was a multi-query), then execution might resume somewhere in the middle of this code, printing half-sentences as a result. When that happens, it is almost always due to a bug elsewhere in the program.

Technically the disjunction-based version works just as well as the if-based version. But in the spirit of defensive programming, it's generally a good idea to stick to if-statements when writing code with side-effects, such as printing.

# Negation

Is there a way to test whether a query fails? We can certainly do it with an if-else construct:

```
(if) (my little
query) (then) (fail)
```

(if) (my little query) (then) (fail) (endif) My little query failed.
[Copy to clipboard]

But Dialog provides a shorthand syntax for this very common operation. By prefixing a query with a tilde character (~, pronounced “not”), the query succeeds if there is no solution, and fails if there is at least one solution. The following code is equivalent to the if-statement above:

```
~(my little
query) My little
```

~(my little query) My little query failed.
[Copy to clipboard]

It also works for blocks:

```
~{ (my little
query) (my other
```

~{ (my little query) (my other little query) }
At least one of the little queries failed.
[Copy to clipboard]

which is equivalent to:

```
(if) (my little
query) (my other
```

(if) (my little query) (my other little query) (then) (fail) (endif)
At least one of the little queries failed.
[Copy to clipboard]

Dialog also allows us to define rules with negated rule heads. When such a rule succeeds, the query fails immediately, and none of the remaining rules are considered. Negated rules could be thought of as having an implicit (just) (fail) at the end.

```
(fruit #apple)
(fruit #banana)
```

(fruit #apple)
(fruit #banana)
(fruit #orange)
(fruit #pumpkin)
(sweet #cookie)
~(sweet #pumpkin)  %% Equivalent to: (sweet #pumpkin) (just) (fail)
(sweet $Obj)
  (fruit $Obj)
[Copy to clipboard]

# Selecting among variations

When writing interactive fiction, it can be nice to be able to add a bit of random variation to the output, or to step through a series of responses to a particular command. Dialog provides this functionality through a mechanism that is

respectfully stolen from the Inform 7 programming language.

To select randomly among a number of code branches, use the expression (select) *...alternatives separated by (or)...* (at random):

```
(descr #bouncer)
	The bouncer
```

```
(descr #bouncer)
 The bouncer
 (select)
 eyes you suspiciously
 (or)
 hums a ditty
 (or)
 looks at his watch
 (at random).
```

[Copy to clipboard]

Note that (or) just revealed itself to be an overloaded operator: When it occurs immediately inside a select expression, it is used to separate alternatives. When it is used anywhere else, it indicates disjunction.

Select-at-random never picks the same branch twice in succession, to avoid jarring repetitions in the narrative. If a uniform distribution is desired, e.g. for implementing a die, an alternative form is available: (select) ... (purely at random).

To advance predictably through a series of alternatives, and then stick with the last alternative forever, use (select) ... (stopping):

```
(report)
	(select)
```

```
(report)
 (select)
 This is printed the first time.
 (or)
 This is printed the second time.
 (or)
 This is printed ever after.
 (stopping)
 (line)
(program entry point)
 (report)
 (report)
 (report)
 (report)
```

[Copy to clipboard]

The output of that program is:

```
This is printed the first time.
This is printed the second time.
This is printed ever after.
This is printed ever after.
```

A combination of predictability and randomness is offered by the following two forms, where Dialog visits each alternative in turn, and then falls back on the specified random behaviour:

(select) *...alternatives separated by (or)...* (then at random)

(select) *...alternatives separated by (or)...* (then purely at random)

To advance predictably through a series of alternatives, and then start over from the beginning, use:

(select) *...alternatives separated by (or)...* (cycling)

The three remaining variants from Inform 7 are currently not supported by Dialog.

# Closures

A *closure* is an anonymous bit of code that can be kept as a value, and invoked at a later time.

A closure definition in curly braces can appear at any place where a value is expected. Once a closure has been created, it can be invoked using the built-in predicate (query $).

Example:

```
      ($X = { Hello,
world! })
```

```
 ($X = { Hello, world! })
 %% Nothing is printed yet, but $X is bound to the code in braces.
 (query $X)    %% This will print “Hello, world!”
```

[Copy to clipboard]

A closure captures the environment surrounding its definition. This means that the same local variables are accessible both inside and outside of the brace-expression. In the following example, the closure is created with a reference to the local variable $X. Afterwards, the same variable is bound to a dictionary word.

```
(program entry
point)
```

```
(program entry point)
 ($Closure = { Hello, $X! })
 ($X = @world)
 (query $Closure)
```

[Copy to clipboard]

The output is:

Hello, world!

It is possible to make multi-queries to closures. The following program:

```
(program entry
point)
```

```
(program entry point)
 (exhaust) {
 *(query { Veni (or) Vidi (or) Vici })
 !
 }
```

[Copy to clipboard]

produces the following output:

Veni! Vidi! Vici!

It is also possible to invoke a closure with a single parameter, using an alternative form of the query builtin: (query $Closure $Parameter). The parameter is accessible from within the closure, by means of the special variable $_. Here is an example:

```
(program entry
point)
```

```
(program entry point)
 ($Greeter = { Hello, $_! })
 (query $Greeter @world)
 (query $Greeter @indeed)
```

[Copy to clipboard]

The output is:

Hello, world! Hello, indeed!

Under the hood, closures are actually lists. The first element is a number, assigned by the compiler to differentiate between the various closure definitions appearing in the program. The remaining elements, if any, are bound to local variables from the environment surrounding the closure definition.

Thus, there is no way to check at runtime whether a value is a closure, or just an ordinary list that happens to begin with a number.

## Stoppable environments

Dialog provides a mechanism for non-local returns, similar to exceptions in other programming languages. By prefixing

a statement (such as a query or a block) with the keyword (stoppable), that statement will execute in a *stoppable environment*. If the built-in predicate (stop) is queried from within the statement, at any level of nesting, execution immediately breaks out of the (stoppable) environment. If the statement terminates normally, either by succeeding or failing, execution also resumes after the (stoppable) construct; (stoppable) never fails. Regardless of how the stoppable environment is left, any choice points created while inside it are discarded.

Stoppable environments can be nested, and (stop) only breaks out of the innermost one. A stop outside of any stoppable environment terminates the program.

Here is a convoluted example:

```
(routine)
	this (stop) (or)
```

```
(routine)
 this (stop) (or) that
(program entry point)
 { Let's (or) now. (stop) }
 (stoppable) {
 take
 (routine)
 another
 }
 shortcut
 (fail)
```

[Copy to clipboard]

The printed output is:

Let's take this shortcut now.

The standard library uses stoppable environments to allow action-handling predicates to stop further actions from being processed. For instance, TAKE ALL may result in several actions being processed, one at a time. If taking the booby-trapped statuette triggers some dramatic cutscene, the code for that cutscene can invoke (stop) to prevent spending time on taking the other items.

Onwards to “Chapter 5: Input and output” • Back to the Table of Contents