Chapter 8: More built-in predicates

(Checking the type of a value • Numbers and arithmetic • List-related predicates • Manipulating dictionary words • System control)

Checking the type of a value

Dialog contains a set of built-in predicates for checking if a value is of a particular type. They are:

(number $X)

Succeeds if and only if $X is bound to a number.

(word $X)

Succeeds if and only if $X is bound to a dictionary word.

(unknown word $X)

Succeeds if and only if $X is bound to a word that wasn't found in the game dictionary.

(empty $X)

Succeeds if and only if $X is bound to an empty list.

(nonempty $X)

Succeeds if and only if $X is bound to a non-empty list.

(list $X)

Succeeds if and only if $X is bound to a list (empty or non-empty).

(bound $X)

Succeeds if and only if $X is bound to anything. That is, the predicate only fails if $X is an unbound variable. It succeeds for numbers, dictionary words, and lists, including lists with one or more unbound variables inside.

(fully bound $X)

Like (bound $), but performs a full recursive check. It succeeds if and only if $X is bound, and—in case of a list—contains only fully bound elements.

The last one comes with an extra feature:

(object $X)

If $X is bound to an object, the query succeeds. If it is bound to anything else, the query fails. But if $X is unbound, *(object $X) backtracks over every object in the game.

Numbers and arithmetic

The Dialog language is designed for symbolic manipulation, predicate logic, and storytelling. Arithmetic is possible, but the syntax is rather clunky.

($A plus $B into $C)

A and B must be bound to numbers; C is unified with their sum. If the result is outside the valid range of numbers, the query fails.

($A minus $B into $C)

A and B must be bound to numbers; C is unified with their difference. If the result is outside the valid range of numbers, the query fails.

($A times $B into $C)

A and B must be bound to numbers; C is unified with their product. If the product is outside the valid range of numbers, the query succeeds, but the numeric result is unpredictable (i.e. it depends on the interpreter).

($A divided by $B into $C)

A and B must be bound to numbers; C is unified with the (integer) quotient after dividing A by B. The query fails if B is zero.

($A modulo $B into $C)

A and B must be bound to numbers; C is unified with the remainder after dividing A by B. The query fails if B is zero.

(random from $A to $B into $C)

A and B must be bound to numbers, such that B is greater than or equal to A. A random number in the range A to B (inclusive) is picked, and then unified with C.

($A < $B)

This predicate succeeds if and only if A is numerically less than B.

($A > $B)

This predicate succeeds if and only if A is numerically greater than B.

Common to all of the above predicates is that they fail if A or B is unbound, or bound to a non-number. C may be bound or unbound; it is unified with the result of the computation.

To check for numerical equality, use regular unification, i.e. ($ = $).

All numbers in Dialog are restricted to the range 0–16383 (inclusive). This range directly supports four-digit numbers such as years and PIN codes. Pocket money should be fairly straightforward to implement by counting in cents; story authors (or library developers) that require more sophisticated number crunching will have to get creative.

List-related predicates

The Dialog programming language provides the following built-in predicates for working with lists:

($Element is one of $List)

Succeeds if $Element appears in the $List. If a multi-query is made, and $Element is unbound, the predicate will backtrack over each member of the list.

(append $A $B $AB)

Unifies $AB with the concatenation of $A and $B. The first parameter ($A) must be bound.

(split $List by $Keyword into $Left and $Right)

See below.

Splitting input by keywords

During parsing, it is often necessary to scan a list for certain keywords, and then split it into two sublists, representing the elements on either side of the matched keyword. It is straightforward to implement this using ordinary rules in Dialog. However, for performance reasons the language also provides a built-in predicate:

(split $Input by
$Keyword into $Left

```
(split $Input by $Keyword into $Left and $Right)
```

[Copy to clipboard]

$Input must be a list of simple values, i.e. it mustn't contain sublists. $Keyword must be a simple value, or a list of simple values.

The $Input list will be scanned, starting at its head, until the first element that is equal to (or appears in) $Keyword is found. A list of the elements that came before the keyword is unified with $Left, and a list of the elements that follow it is unified with $Right. That is, neither $Left nor $Right includes the keyword itself.

When invoked as a multi-query, the predicate backtracks over each matching position. Thus:

*(split [the
good , the bad and

```
*(split [the good , the bad and the ugly]
by [and ,]
into $Left and $Right)
```

[Copy to clipboard]

will succeed twice: First, binding $Left to [the good] and $Right to [the bad and the ugly], and then a second time binding $Left to [the good , the bad] and $Right to [the ugly].

The split-by predicate can also be used to check whether a list contains one or more of a set of keywords. The standard library uses it that way in the following rule definition:

($X contains one of
$Y)

```
($X contains one of $Y)
 (split $X by $Y into $ and $)
```

[Copy to clipboard]

Manipulating dictionary words

Dictionary words are usually treated as atomic units, but it is possible to extract their constituent characters using the built-in predicate (split word $ into $). The output is a list of single-character dictionary words and/or single-digit numbers. Thus:

(split word
@fission into $List)

```
(split word @fission into $List)
$List
```

[Copy to clipboard]

will print:

```
[f i s s i o n]
```

Conversely, it is possible to construct a dictionary word from a list of words (single-character or otherwise), using (join words $ into $):

(join words [f u
s i o n] into $Word)

```
(join words [f u s i o n] into $Word)
$Word
```

[Copy to clipboard]

will print:

```
fusion
```

The join-words predicate fails if any of the following is true:

- The input is anything other than a list of dictionary words and/or numbers.
- The input consists of more than a single character, and one of those characters is either a word representing a special keystroke (like backspace), or one of the word-separating characters (see Input): . , ; " * ( )
- The resulting word would exceed a backend- or interpreter-imposed length limit. If there is a limit, it is guaranteed to be at least 64 characters.

It is possible to split and join numbers as though they were words:

> (get input
[$W])

```
> (get input [$W])
(split word $W into $Chars)
(split $Chars by 5 into $LeftChars and $RightChars)
$LeftChars, $RightChars. (line)
(join words $LeftChars into $Left)
(join words $RightChars into $Right)
($Left plus $Right into $Sum)
The sum is $Sum.
```

[Copy to clipboard]

This could result in the following interaction:

```
> 11522
[1 1], [2 2].
The sum is 33.
```

System control

The following built-in predicates offer low-level control over the interpreter and the Dialog runtime. This is decidedly in the domain of library code, so story authors rarely need to worry about these predicates.

(quit)

Immediately terminates the program. This predicate neither fails nor succeeds.

(restart)

Resets the program to its initial state. The only part of the game state that may survive a restart is the state of the output machinery (including the current style and on-screen contents, and whether the transcript feature is on or off). If the operation succeeds, execution resumes from the start of the program. If there is an error, or the interpreter doesn't support restarting, execution continues normally, i.e. the query succeeds.

(save $ComingBack)

Attempts to save the current game state to a file. The interpreter takes care of asking the player for a filename. In the event of a save error, or if the operation was cancelled, the query fails. On success, the parameter is unified with 0 if we just saved the state, and with 1 if we just restored the state from a file saved by this query.

(restore)

Attempts to restore the current game state from a file. The interpreter takes care of asking the player for a filename. The only part of the game state that may survive a restore is the state of the output machinery (including the current style and on-screen contents, and whether the transcript feature is on or off). If the operation succeeds, execution resumes after the query from which the save file was written. Otherwise, in the event of a load error or if the operation was cancelled, execution continues normally, i.e. the query succeeds.

(save undo $ComingBack)

Works just like (save $), but stores the game state in a buffer in memory. This operation is typically invoked once per move.

(undo)

Works like (restore), but restores the game state from the undo buffer. If there is no saved undo state, the predicate fails. If there's some other problem—such as the interpreter imposing a limit on the number of undo states that are retained in memory—the predicate succeeds, and execution continues normally.

(interpreter supports quit)

Succeeds if and only if the current interpreter handles (quit) in a way that is meaningful to the player. For instance, it fails under the Å-machine web interpreter, because a web page cannot close itself.

(interpreter supports undo)

Succeeds if and only if the current interpreter declares that it supports undo functionality.

(transcript on)

Enables the transcript feature. The interpreter takes care of asking the player for a filename. If the operation succeeds, the query suceeds. In case of an error, or if the operation was cancelled, the query fails.

(transcript off)

Disables the transcript feature. This predicate always succeeds.

(display memory statistics)

Prints a line of information specific to the compiler backend, about the peak memory usage in the heap, auxiliary heap, and long-term heap areas. This only works for compiled code (Z-machine or Å-machine). The size of these areas can be adjusted by passing commandline options to the compiler. 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.

Onwards to “Chapter 9: Beyond the program” • Back to the Table of Contents