Chapter 3: Choice points

(Disjunctions • Backtracking • Multi-queries • Visiting all solutions • Collecting values • Collecting words • Accumulating numbers • Just • Infinite loops)

Disjunctions

So far, we have seen that rule bodies can contain text and values to be printed, as well as queries to be made to predicates. As soon as a query fails, the entire rule fails. Hence, a rule body can be regarded as a *conjunction* (boolean "and") of queries.

On the other hand, the rules that make up a predicate are tried one at a time until one succeeds—they form a *disjunction* (boolean "or").

It is also possible, using special syntax, to put disjunctions inside rule bodies. The (or) keyword may look like a query, but is in fact an infix operator. Here is an example:

```
(program entry
point)
```

```
(program entry point)
 Apple: (descr #apple) (line)
 Steak: (descr #steak) (line)
 Door: (descr #door) (line)
(descr $Obj)
 (tasty $Obj) Yummy!
(descr $)
 You see nothing unexpected about it.
(tasty $Obj)
 (fruit $Obj)
 (or)
 ($Obj = #steak) (player eats meat)
(fruit #apple)
(player eats meat)
```

[Copy to clipboard]

Note that this part of the code:

```
(tasty $Obj)
      (fruit $Obj)
```

```
(tasty $Obj)
 (fruit $Obj)
 (or)
 ($Obj = #steak) (player eats meat)
```

[Copy to clipboard]

is exactly equivalent to:

```
(tasty $Obj)     (fruit
$Obj)
```

```
(tasty $Obj) (fruit $Obj)
(tasty $Obj) ($Obj = #steak) (player eats meat)
```

[Copy to clipboard]

The output from the program is:

```
Apple: Yummy!
Steak: Yummy!
Door: You see nothing unexpected about it.
```

There can be more than two subexpressions in a disjunction; just separate them all with (or) keywords, e.g. a (or) b (or) c. When Dialog reaches a disjunction, it will attempt to enter the first subexpression. If that subexpression succeeds, the entire disjunction succeeds. But if it fails, Dialog tries the second subexpression, and so on.

Note that the (or) operator has low precedence: It includes everything to the left and right of itself. Therefore, if we try to move the tasty condition verbatim into the Yummy rule itself, we run into trouble:

```
%% This will not
work as desired:
```

```
%% This will not work as desired:
(descr $Obj)
 (fruit $Obj) (or) ($Obj = #steak) (player eats meat)
 Yummy!
(descr $)
 You see nothing unexpected about it.
(fruit #apple)
(player eats meat)
```
[Copy to clipboard]

Now if we try to describe the apple, only the first leg of the (or) expression will execute, and the query to (descr $) will succeed without printing anything.

Another special syntax comes to the rescue: Curly braces { ... } can be used to organize Dialog code into *blocks*. The (or) operation extends as far as it can to the left and right, but it won't go beyond the limits of its containing block. Thus:

```
%% The following
version is correct:
```

```
%% The following version is correct:
(descr $Obj)
 { (fruit $Obj) (or) ($Obj = #steak) (player eats meat) }
 Yummy!
(descr $)
 You see nothing unexpected about it.
(fruit #apple)
(player eats meat)
```
[Copy to clipboard]

Backtracking

Now consider the following predicate, which determines who is considered royalty in some fictional world:

```
(royalty #king)
(royalty #queen)
```

```
(royalty #king)
(royalty #queen)
(royalty $Person)
 {
 (the mother of $Person is $Parent)
 (or)
 (the father of $Person is $Parent)
 }
 (royalty $Parent)
```
[Copy to clipboard]

To check whether a person is royalty, unless they're the actual king or queen, we first consider their mother (in the first leg of the disjunction). Let's say the query succeeds, binding $Parent to whoever is the mother of $Person. Because the first leg succeeded, so does the entire disjunction. And if the parent turns out to be royalty, then the entire rule succeeds. But otherwise, we're in a position where $Parent is bound, and the recursive call to (royalty $) has failed. When this happens, Dialog needs to *unbind* $Parent, before going back to explore the second leg of the disjunction. This is called *backtracking*.

Here is how backtracking works: When Dialog first encounters a disjunction, it creates a *choice point*. This is a bit like saving the state of a game: A snapshot is made of all relevant variables, and later on we can restore the snapshot and explore a different path forward. And we have already seen the condition that causes Dialog to revert back to the last choice point: It is failure (to satisfy a query).

On failure, all variable bindings are rolled back to their state at the time when the choice point was created, including any variable bindings made from within subroutine calls. But of course, side-effects cannot be undone: What's printed is printed. This can be used to narrate what's going on while the program is searching for a solution:

```
(program entry
point)
```

```
(program entry point)
        {
        ($X = #door)
        (or)
        ($X = #foot)
        (or)
        ($X = #apple)
        (or)
        ($X = #pencil)
        }
        Checking (the $X).
        (fruit $X)    %% If the query fails, the most recent choice point is restored.
        Yes, it's a fruit!

(fruit #apple)

(the #apple)  the green apple
(the #door)   the oaken door
(the #foot)   my left foot
(the #pencil) the pencil
```

[Copy to clipboard]

The output is:

Checking the oaken door. Checking my left foot. Checking the green apple. Yes, it's a fruit!

Backtracking during rule matching

Recall that when a query is made, Dialog considers each rule definition in turn, in program order, unifying the parameters of the query with the parameters of the rule head. In case of a match, the rule body starts to execute. We've also seen, that if a failure occurs while executing the body, Dialog proceeds to check the next rule in the predicate, and it keeps doing this until one of the rules succeeds.

Now we are in a position to understand what is actually going on: Before attempting to unify the parameters with those of a rule head, Dialog creates a choice point. In case of failure, the query parameters (and all other variables that have been bound since) are restored to their earlier state, and the next rule is attempted.

However, there is one critical difference between queries and disjunctions: As soon as a rule succeeds, that query is considered over and done. So on success, Dialog simply discards any lingering choice points that were created as part of the query.

Multi-queries

Sometimes when we query a predicate, we want to be able to go back and reconsider every matching rule, even if we already found one that was successful. That is, we want to inhibit the default behaviour of discarding choice points as soon as a rule succeeds. This is done by putting an asterisk (*) before the query, which turns it into a *multi-query*.

No whitespace is allowed between the asterisk and the opening parenthesis.

Here is an example:

```
(program entry
point)
```

```
(program entry point)
 *(fruit $Obj)    %% This is a multi-query. Also, $Obj is unbound here.
 $Obj is a fruit.
 (colour $Obj)

 We found a fruit that is also a colour!

(colour #blue)
(colour #orange)

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

[Copy to clipboard]

The output of the program is:

#apple is a fruit. #orange is a fruit. We found a fruit that is also a colour!

A multi-query behaves like a disjunction, in that it installs a choice point for going back and trying something else (in this case, attempting to match the query with the next rule in the program). Unlike a normal query, it doesn't resemble a traditional subroutine call, because it can effectively “return” more than once.

The multi-query may set up choice points of its own, for instance using (or) expressions or by making nested multi-queries. All of these choice points remain in effect after the multi-query returns. This provides a very powerful mechanism for searching through a database of relations:

```
(#lisa is a child of
#marge)
```

```
(#lisa is a child of #marge)
(#lisa is a child of #homer)
(#bart is a child of #marge)
(#bart is a child of #homer)
(#homer is a child of #mona)
(#homer is a child of #abraham)
(#herb is a child of #mona)
(#herb is a child of #abraham)

(male #bart)
(male #homer)
(male #herb)
(male #abraham)

($X is the father of $Y)
 *($Y is a child of $X)
 (male $X)

($X is a grandfather of $Y)
 *($Y is a child of $Parent)
 *($X is the father of $Parent)

(program entry point)
 ($X is a grandfather of #lisa)
 The answer is $X.
```

[Copy to clipboard]

Output:

The answer is #abraham.

Multi-queries provide a clean mechanism for going back and trying various options. This gives the code a declarative flavour, which can often improve readability.

Although the declarative style makes it easy to see what problem we're trying to solve, it may not be obvious at first how the code in the previous example works. The following detailed description may help:

A query is first made to ($ is a grandfather of $), with the second parameter bound to #lisa. That rule in turn makes a multi-query, *(#lisa is a child of $Parent), with the purpose of backtracking over Lisa's parents. At first, this matches the very first rule in the program, binding $Parent to #marge. However, because this was a multi-query, a choice point remains in effect for coming back and looking for another parent of Lisa later.

Next, an attempt is made to bind $X to the father of #marge. This invokes the father-rule, which in turn makes another multi-query, in this case *(#marge is a child of $X). But Marge's parents aren't in the database, so this multi-query fails. The failure makes Dialog backtrack to the last choice point, unbinding $Parent as it goes, and proceeding to look for another of Lisa's parents, starting with the second rule of the program. This succeeds, and $Parent is now bound to #homer.

A query is now made to determine the father of #homer. We're in the father-rule again, making a multi-query: *(#homer is a child of $X). This will first bind $X to #mona, but (male #mona) fails. Thanks to the choice point created by the recent multi-query, Dialog goes back and binds $X to #abraham instead. Now, starting with (male #abraham), everything succeeds, and we end up in the top-level rule again, with $X bound to #abraham. Because the original query ($X is a grandfather of #lisa) was a regular (non-multi) query, we also know that any choice points created inside it have now been discarded; a regular query is guaranteed to return at most once.

That example illustrates a general class of problems in the spirit of the pencil-and-paper game Sudoku: Dialog searches through the parameter space by tentatively binding variables to values and checking that all constraints are met, backtracking when they are not, until one or more solutions are found. The main difference between Sudoku and the family tree example is that in the former, all the variables are lined up on a grid from the start, whereas in the latter, the variables are parameters of recursive queries.

In interactive fiction, this technique can be useful for working with small relational databases, as in the example. But its

biggest strength is in parsing and disambiguation: It is possible to implement a parser in a clean, declarative style that takes input (such as NORTH, EAST), and backtracks over various possible interpretations (such as “Go north and east” and “Tell Mrs. North to go east”), which can then be weighed against each other based on their likelihood from a semantical point of view. This is indeed the approach taken by the Dialog standard library.

Now that we know about multi-queries, we are better equipped to deal with lists. There's a very handy built-in predicate called ($ is one of $) that takes two parameters—a value and a list—and attempts to unify the value with each member of the list in turn. If both parameters are bound, a regular query to ($ is one of $) can be used to check whether the value is in the list:

```
(descr $Obj)    ($Obj is one of [#apple
```

```
(descr $Obj) ($Obj is one of [#apple #orange #banana]) Yummy!
(descr $Obj) You see nothing unexpected about it.
```

[Copy to clipboard]

But with the help of a multi-query, the same predicate can be used as a list iterator. Here is a more elegant re-implementation of an earlier example:

```
(program entry point)
```

```
(program entry point)
        *($X is one of [#door #foot #apple #pencil])
        Checking (the $X).
        (fruit $X)
        Yes, it's a fruit!
(fruit #apple)
(the #apple)  the green apple
(the #door)   the oaken door
(the #foot)   my left foot
(the #pencil) the pencil
```

[Copy to clipboard]

The output is:

Checking the oaken door. Checking my left foot. Checking the green apple. Yes, it's a fruit!

Incidentally, ($ is one of $) is a built-in predicate for performance reasons only. We could have defined it ourselves like this:

```
($Element is one of [$Element | $])
```

```
($Element is one of [$Element | $])
($Element is one of [$ | $Tail])
 *($Element is one of $Tail)
```

[Copy to clipboard]

Note the asterisk in front of the recursive call. By making this a multi-query, $Element iterates over the entire $Tail, and we simply propagate each of those successful returns up to our caller. The base case of the recursion is implicit, in that neither of the rules will match an empty list.

Visiting all solutions

A Dialog statement can be prefixed with the (exhaust) keyword. This will cause Dialog to consider every nook and cranny of the search tree, without remaining on the first successful branch. The statement—usually a block—executes, backtracking is performed on success as well as on failure, and in the end the entire (exhaust) construct succeeds. Here is an example:

```
(program entry point)
```

```
(program entry point)
        (exhaust) {
        *($X is one of [#door #foot #apple #pencil])
        (line)
        Checking (the $X).
        (fruit $X)
        Yes, it's a fruit!
```

```
        }
        (line)
        The program continues, but $X is unbound again.
(fruit #apple)
(the #apple)  the green apple
(the #door)   the oaken door
(the #foot)   my left foot
(the #pencil) the pencil
```
[Copy to clipboard]

The output is:

```
Checking the oaken door.
Checking my left foot.
Checking the green apple. Yes, it's a fruit!
Checking the pencil.
The program continues, but $ is unbound again.
```

Incidentally,

```
    (exhaust)
<i>statement</i>
```
(exhaust) *statement*
[Copy to clipboard]

is exactly equivalent to

```
    {
<i>statement</i>
```
{ *statement* (fail) (or) }
[Copy to clipboard]

where (fail) is a built-in predicate that always fails.

Collecting values

Computing our way through all the solutions of a query is useful, but sometimes we would like to collect the results of those computations into a list, and then perform some work on the list as a whole. This can be done with the following special syntax:

```
    (collect
$Element)
```
(collect $Element)
...
(into $List)
[Copy to clipboard]

The ellipsis represents some code that is expected to bind $Element to a value. That value is remembered, and backtracking is performed until all possibilities have been exhausted. The collected values are placed into a list, in the order in which they were encountered, and that list is then unified with the output parameter, $List.

Example:

```
(program entry
point)
```
```
(program entry point)
 (collect $F)
 *(fruit $F)
 (into $FruitList)
 Come and buy! $FruitList!
(fruit #apple)
(fruit #orange)
(fruit #banana)
```
[Copy to clipboard]

The output is:

Come and buy! [#apple #orange #banana]!

Note that the query to (fruit $) must be a multi-query, otherwise only the first fruit is returned. The first statement inside a collect-expression is nearly always a multi-query.

Collecting words

There is also a special variant of the collect-into syntax:

```
(collect words)
...
```

(collect words)
...
(into $List)
[Copy to clipboard]

This makes Dialog execute the inner statements, while diverting all output into a list of dictionary words. That list is then unified with the output parameter. Typically, a game has a rule for printing the name of an object. This construct makes it possible to gather all the words that make up that name, in order to match them against player input.

It is assumed that words are collected for purposes of comparison. Thus, in the interest of performance, only the essential part of each dictionary word is reported. On the Z-machine, a long word such as north-west will come out as @north-we, but that will unify just fine with, say, a @north-west obtained from the player's input.

Example:

```
(name #apple)
green apple
```

(name #apple) green apple
(dict #apple) yummy %% Extra synonyms can be listed here.

(name #door) mysterious door
(dict #door) oaken oak

%% By default, include any words mentioned in the name rule:
(dict $Obj) (name $Obj)

(program entry point)
(exhaust) {
*($Obj is one of [#apple #door])
(collect words)
*(dict $Obj)
(into $List)
The (name $Obj) can be referred to using the words $List.
(line)
}
[Copy to clipboard]

The output is:

The green apple can be referred to using the words [yummy green apple].
The mysterious door can be referred to using the words [oaken oak mysteriou door].

Printed values (numbers, lists, objects) also end up in the collection, as themselves.

Just as when parsing player input, certain punctuation characters are treated as separate words. They are: . , ; * " ( )
When printed back, . , ; and) inhibit whitespace on their left, and (inhibits whitespace on its right:

```
(program entry
point)
```

(program entry point)
(collect words)
Hello, world!
(into $List)
The list is: $List (line)
Printing each word:
(exhaust) {
*($Word is one of $List)
$Word

```
 }
```
[Copy to clipboard]

produces the output:

```
The list is: [hello , world!]
Printing each word: hello, world!
```

Accumulating numbers

Instead of returning the collected values as a list, it is possible to add them together:

```
      (accumulate
$Element)
```

```
 (accumulate $Element)
 ...
 (into $Sum)
```
[Copy to clipboard]

The elements have to be numbers, and the resulting sum must be within the valid range of numbers (i.e. no more than 16383). If that is not the case, then the inner expression is still exhausted (including any side-effects), before the whole accumulate-statement fails.

The $Element can be a constant. If it is 1, the resulting sum will be the number of ways the inner expression succeeded. Thus:

```
(program entry
point)
```

```
(program entry point)
 (accumulate 1)
 *(fruit $)
 (into $Num)
 I know of $Num pieces of fruit.

(fruit #apple)
(fruit #orange)
(fruit #banana)
```
[Copy to clipboard]

would produce the output:

```
I know of 3 pieces of fruit.
```

Just

We have seen that the rules of a predicate work together as a disjunction. Each rule is tried in turn, and failure causes Dialog to backtrack and resume with the next rule. When a rule succeeds, the backtracking may or may not stop: This depends on whether the caller was making a multi-query or a regular query. But sometimes, it makes sense to give the callee, i.e. the set of rules that make up the predicate being queried, influence over when to stop the backtracking.

This is achieved with the (just) statement. When a rule invokes (just)—and this can happen anywhere inside the rule body—Dialog discards any choice points created since the beginning of the present query.

Consider again the example in the previous section, where a list of synonyms is provided for each object by a (dict $) rule. When a multi-query is made to that predicate, Dialog inevitably finds its way to the generic (dict $) rule that also throws in all of the words making up the object's printed name. It is conceivable that a game has a few objects that the player should not be able to refer to using their printed names. A common example is the object that represents the player character inside the game world. Suppose (although this is not necessarily a good idea in practice) that you want this object to print as “you”, but parse only as “me”:

```
(name #player)
you
```

```
(name #player)      you
(dict #player)      (just) me

(name #door)        mysterious door
(dict #door)        oaken oak    %% Extra synonyms.

%% By default, include any words mentioned in the name rule:
(dict $Obj)         (name $Obj)
```
[Copy to clipboard]

Here, when a multi-query is made for *(dict #player), Dialog sets up a choice point as usual, and enters the first matching rule. But this rule uses the (just) keyword, immediately discarding the choice point. The upshot of this is that the generic rule, (dict $Obj), is never considered for the #player.

To recap, (just) discards any choice points that have been created so far while dealing with the current query. This includes choice points created by inner multi-queries, and it even extends a little bit outside the present rule, to the backtracking-over-matching-rules mechanism. But it doesn't go any further than that. Code such as (just) (just) is redundant; the second (just) has no effect.

Infinite loops

The built-in predicate (repeat forever) provides an unlimited supply of choice points. This can be used to create infinite loops (such as the main game loop or the game-over menu).

(program entry point)

```
(program entry point)
 *(repeat forever)
  This gets printed over and over. (line)
 (fail)
```

[Copy to clipboard]

In the above example, the query to (fail) makes Dialog backtrack to the multi-query to (repeat forever), which will keep on returning, successfully, over and over again.

It is possible to break out of the infinite loop by discarding the choice-point created by (repeat forever). This can be done explicitly using (just), or implicitly by successfully returning from a regular (non-multi) query.

Onwards to “Chapter 4: More control structures” • Back to the Table of Contents