# 7

# Preventing Backtracking

## Chapter Aims

After reading this chapter you should be able to:

- Use the cut predicate to prevent unwanted backtracking
- Use 'cut with failure' to specify exceptions to general rules.

## Introduction

Backtracking (as described in Chapter 3) is a fundamental part of the process by which the Prolog system satisfies goals. However, it can sometimes be too powerful and lead to inappropriate results. This chapter is about preventing the Prolog system from backtracking using a built-in predicate called *cut*, which is written as an exclamation mark **!** character.

Before going on, it is worth issuing a warning. Many Prolog users see preventing backtracking using 'cut' as being 'against the spirit of the language' and some would even like to see it banned altogether! It is included in the language (and in this book) because it can sometimes be very useful. When used badly, it can also be a cause of programming errors that are very hard to find. The best advice is probably to use it only sparingly and with care.

## 7.1 The Cut Predicate

We start by giving two examples of predicate definitions that appear correct but give erroneous results when used with backtracking.

Example 1

The **larger** predicate takes the value of the larger of its first two arguments (which are assumed to be numbers) and returns it as the value of the third.

```
larger(A,B,A):-A>B.
larger(A,B,B).
```

With the usual 'top to bottom' searching of clauses, the second clause can reasonably be assumed to apply only when *A* is less than or equal to *B*. Testing the definition with 8 and 6 as the first two arguments gives the correct answer.

**?- larger(8,6,X).**
**X = 8**

However, if the user forces the system to backtrack at this stage, it will go on to examine the second clause for **larger** and generate an incorrect second answer.

**?- larger(8,6,X).**
**X = 8 ;**
**X = 6**
**?-**

Example 2

The definition of predicate **sumto/2** given below is a slightly modified version of the one given in Chapter 6. It still appears to be correct, but has a serious flaw.

The goal **sumto(N,S)** causes the sum of the integers from 1 to N to be calculated and returns the answer as the value of *S*.

```
sumto(1,1).
sumto(N,S):-N1 is N-1,sumto(N1,S1),
   S is S1+N.
```

**?- sumto(3,S).**
**S = 6**

However, forcing backtracking will now cause the system to crash with a cryptic error message, such as 'stack overflow'. Whilst evaluating the goal **sumto(3,S)** the Prolog system will try to find a solution for the goal **sumto(1,S)**. The first time it does this the first clause is used and the second argument is correctly bound to 1. On backtracking the first clause is rejected and the system attempts to satisfy the goal using the second clause. This causes it to subtract one from one and then evaluate the goal **sumto(0,S)**. Doing this will in turn require it

to evaluate **sumto(-1,S1)**, then **sumto(-2,S1)** and so on, until eventually the system runs out of memory.

Examples 1 and 2 could both be remedied by using additional goals in the definition of the predicates, e.g. by changing the second clause of the definition of **larger** to

```
larger(A,B,B):-A=<B.
```

and the second clause in the definition of **sumto** to

```
sumto(N,S):-N>1,N1 is N-1,sumto(N1,S1),S is
S1+N.
```

However, in other cases identifying such additional terms can be considerably more difficult.

A more general way to avoid unwanted backtracking is to use a *cut*. The goal **!** (pronounced 'cut') in the body of a rule always succeeds when first evaluated. On backtracking it always fails and prevents any further evaluation of the current goal, which therefore fails.

Example 1 (revised)

```
larger(A,B,A):-A>B,!.
larger(A,B,B).
```

**?- larger(8,6,X).**
**X = 8**
**?-**

Example 2 (revised)

```
sumto(1,1):-!.
sumto(N,S):-N1 is N-1,sumto(N1,S1),
   S is S1+N.
```

**?- sumto(6,S).**
**S = 21**
**?-**

Note that backtracking over a cut not only causes the evaluation of the current clause of **larger** or **sumto** to be abandoned but also prevents the evaluation of any other clauses for that predicate.

Example 3

The following incorrect program defines a predicate **classify/2** that classifies a number (its first argument) as either positive, negative or zero. The first clause deals explicitly with the case where the first argument is zero. The second deals with a negative value, leaving the third to deal with positive values.

```
/* classify a number as positive, negative or zero */
classify(0,zero).
classify(N,negative):-N<0.
classify(N,positive).
```

However, as before, the absence of a specific test for a positive argument causes problems when the user forces the system to backtrack.

**?- classify(0,N).**
**N = zero ;**
**N = positive**

**?- classify(-4,X).**
**X = negative ;**
**X = positive**
**?-**

This can be rectified either by changing the third clause to

```
classify(N,positive):-N>0.
```

or by using cuts.

Example 3 (revised)

```
classify(0,zero):-!.
classify(N,negative):-N<0,!.
classify(N,positive).
```

**?- classify(0,N).**
**N = zero**

**?- classify(-4,N).**
**N = negative**
**?-**

So far all the incorrect programs could have been rectified by adding an additional goal to one of the clauses rather than using cuts, and that would probably have been the better approach. The following example shows a more difficult case.

Example 4

A very common requirement is to prompt the user for an answer to a question until a valid answer (e.g. yes or no) is entered. The following program does this using a **repeat** loop, but unhelpfully will continue to prompt for valid answers on backtracking.

```
get_answer(Ans):-
   write('Enter answer to question'),nl,
   repeat,write('answer yes or no'),read(Ans),
   valid(Ans),write('Answer is '),write(Ans),nl.
valid(yes).
valid(no).
```

**?- get_answer(X).**
**Enter answer to question**
**answer yes or no: maybe.**
**answer yes or no: yes.**
**Answer is yes**
**X = yes ;**

**answer yes or no: no.**
**Answer is no**
**X = no ;**

**answer yes or no: unsure.**
**answer yes or no: yes.**
**Answer is yes**
**X = yes**

(and so on indefinitely).

Example 4 (revised)

Adding a final cut to the definition of **get_answer** will prevent the unwanted backtracking.

```
get_answer(Ans):-
   write('Enter answer to question'),nl,
```

```
   repeat,write('answer yes or no'),read(Ans),
   valid(Ans),write('Answer is '),write(Ans),nl,!.
valid(yes).
valid(no).
```

**?- get_answer(X).**
**Enter answer to question**
**answer yes or no: maybe.**
**answer yes or no: unsure.**
**answer yes or no: yes.**
**Answer is yes**
**X = yes**
**?-**

The above example, like all the other examples in this chapter so far, illustrates the solution to a problem that could have been avoided if the user had not chosen to force the system to backtrack. In practice, it is most unlikely that anyone would want to do so. The prevention of unwanted backtracking is of much more practical importance when one predicate 'calls' another (i.e. makes use of another as a goal in the body of one of its clauses). This can lead to apparently inexplicable results.

Example 5

The **go** predicate in the following program uses a **repeat** loop to prompt the user for input until a positive number is entered. However, the lack of cuts in the definition of the **classify** predicate leads to incorrect answers.

```
classify(0,zero).
classify(N,negative):-N<0.
classify(N,positive).
go:-write(start),nl,
   repeat,write('enter a positive value'),read(N),
   classify(N,Type),Type=positive,
   write('positive value is '),write(N),nl.
```

**?- go.**
**start**
**enter a positive value: 28.**
**positive value is 28**
**yes**

**?- go.**
**start**
**enter a positive value: -6.**
**positive value is -6**
**yes**

**?- go.**
**start**
**enter a positive value: 0.**
**positive value is 0**
**yes**

Changing the definition of **classify** to the one given in Example 3 (revised) above gives the expected behaviour for **go**.

Example 5 (revised)

```
classify(0,zero):-!.
classify(N,negative):-N<0,!.
classify(N,positive).
go:-write(start),nl,
   repeat,
   write('enter a positive value'),read(N),
   classify(N,Type),
   Type=positive,
   write('positive value is '),write(N),nl.
```

**?- go.**
**start**
**enter a positive value: -6.**
**enter a positive value: -7.**
**enter a positive value: 0.**
**enter a positive value: 45.**
**positive value is 45**
**yes**

## 7.2 Cut with Failure

Another use of 'cut' that can sometimes be helpful is to specify exceptions to general rules.

Suppose that we have a database of the names of birds, such as

```
bird(sparrow).
bird(eagle).
bird(duck).
bird(crow).
bird(ostrich).
bird(puffin).
bird(swan).
bird(albatross).
bird(starling).
bird(owl).
bird(kingfisher).
bird(thrush).
```

A natural rule to add to this would be

```
can_fly(X):-bird(X).
```

corresponding to 'all birds can fly'.

Unfortunately this rule is over general. There are a few exceptions, notably that ostriches cannot fly. How can we ensure that the goal **can_fly(ostrich)** will always fail? The obvious approach is to change the definition of the **can_fly** predicate to

```
can_fly(ostrich):-fail.
can_fly(X):-bird(X).
```

However this does not give the desired result:

**?- can_fly(duck).**
**yes**

**?- can_fly(ostrich).**
**yes**

The **can_fly(ostrich)** goal is matched with the head of the first **can_fly** clause. Attempting to satisfy the goal in the body of that clause (i.e. **fail**) obviously fails, so the system next looks at the second **can_fly** clause. The goal matches with the head, and the goal in the body of the clause, i.e. **bird(X)** is also satisfied, so the **can_fly(ostrich)** goal succeeds. This is obviously not what was intended.

The desired effect can be achieved by replacing the **can_fly** clauses by

```
can_fly(ostrich):-!,fail.
can_fly(X):-bird(X).
```

**?- can_fly(duck).**
**yes**

**?- can_fly(ostrich).**
**no**

As before, the **can_fly(ostrich)** goal is matched with the head of the first **can_fly** clause. Attempting to satisfy the goal in the body of that clause (i.e. **fail**) fails, but the cut prevents the system from backtracking and so the **can_fly(ostrich)** goal fails.

The combination of goals **!,fail** is known as *cut with failure*.

## Chapter Summary

This chapter describes how the 'cut' predicate can be used to prevent undesirable backtracking and how 'cut' can be used in conjunction with the 'fail' predicate to specify exceptions to general rules.