

Chapter 5

Backtrack control

Prolog will automatically backtrack to satisfy a goal. This relieves the programmer of the burden of putting in backtrack explicitly. On the negative side, this might lead to a less efficient program.

As an example, consider the following double-step function:

$$y = \begin{cases} 0 & \text{if } x < 3 \\ 2 & \text{if } 3 \leq x < 6 \\ 4 & \text{if } x \geq 6. \end{cases}$$

A straightforward Prolog program would be the following:

`f(X,0):-X<3.`

`f(X,2):-3=<X,X<6.`

`f(X,4):-6=<X.`

A few cases

1. If we pose the following question:

$?-f(1, Y), 2 < Y.$

When satisfying $f(1, Y)$, Y is instantiated with 0, and we know, by the definition, that this is the only applicable clause. The second goal fails.

Note that since the three cases for X are exclusive, it won't help to use another value of X .

But, because of the automatic backtrack, Prolog makes two more useless attempts to satisfy the second goal.

Question: How do we know?

Answer: Use trace.

A useful feature

The debugging facility `trace` will show the actual process of goal resolution.

```
[debug] 8 ?- trace.
```

Yes

```
[trace] 8 ?- f(1, Y), 2<Y.  
    Call: (8) f(1, _G441) ? creep  
^ Call: (9) 1<3 ? creep  
^ Exit: (9) 1<3 ? creep  
    Exit: (8) f(1, 0) ? creep  
    Redo: (8) f(1, _G441) ? creep  
^ Call: (9) 3=<1 ? creep  
^ Fail: (9) 3=<1 ? creep  
    Redo: (8) f(1, _G441) ? creep  
^ Call: (9) 6=<1 ? creep  
^ Fail: (9) 6=<1 ? creep
```

No

In the above, the `Redo` shows backtrack.

What do we do?

We can overcome this inefficiency by utilizing a mechanism called a “cut” as follows:

$f(X,0) : -X < 3, ! .$

$f(X,2) : -3 \leq X, X < 6, ! .$

$f(X,4) : -6 \leq X .$

Now, once the cut is met, Prolog will commit to this clause, and no decision made prior to this point will be backtracked.

When we call

$?-f(1, Y) .$

Once $1 < 3$ succeeds, i.e., a fitting case is found, no further attempts will be made, since by definition, none of the other cases will fit.

2. Given the modified program, if we now pose a new question

?-f(7,Y).

we will get back $Y=4$. What happened is that Prolog tries the first rule and failed; then back-track (Since the cut is not seen yet.) tried the second: this time, the first subgoal succeeds, but the second fails. Again, since the cut is not seen yet, it backtracks to try the third and succeeds.

Here we see another piece of inefficiency: since it finds out $x < 3$ fails, the first subgoal, $x \geq 3$, must succeed and Prolog should not try to evaluate it again. Thus, it can be further put into the following form:

f(X,0):-X<3,!.

f(X,2):-X<6,!.

f(X,4):-6=<X.

3. Given the further modified program, if we drop the cut operators,

$f(X,0) :- X < 3.$

$f(X,2) :- X < 6.$

$f(X,4) :- 6 \leq X.$

and pose the following goal $?-f(1,Y)$, we will get multiple answers, $Y=0$, $Y=2$.

Thus, this form not only affects the procedural meaning, but also the declarative meaning of the program. Sometimes, it is called a *red cut*.

We should really be careful in using the cut operator.

“Cut” in General

Let's call the *parent goal* the goal that matched with the head of the clause containing the cut operator. When the cut is seen, it succeeds immediately, and also commits Prolog to all the choices it has made between the time the parent goal is called, and the time the cut is seen. All the alternative choices between them are simply discarded.

For example, given the following code:

```
C:-P,Q,R,! ,S,T,U.
```

```
C:-V.
```

```
A:-B,C,D.
```

```
?-A.
```

Once the ‘!’ is seen, all the alternative ways to solve P,Q,R, as well as that for C, are discarded.

“Cut” examples

1. Computing maximums. $\text{max}(X, Y, \text{Max})$ holds if Max is the bigger one between X and Y .

$\text{max}(X, Y, X) : -X > Y, ! .$

$\text{max}(X, Y, Y) .$

The above will not work if Max is instantiated. For example, the goal $\text{max}(3, 1, 1)$ succeeds. (?)

Answer: The first clause fails, where the cut is not seen yet, hence the second succeeds by default.

An improved one could be

$\text{max}(X, Y, \text{Max}) : -X > Y, !, \text{Max} = X .$

$\text{max}(X, Y, Y) .$

2. Finding the first occurrence of X in L.

```
First(X, [X|L]) :- !.
```

```
First(X, [_|L]) :- First(X, L).
```

3. Adding in X to L if it is not there already.

```
add(X, L, L) :- member(X, L), !.
```

```
add(X, L, [X|L]).
```

Homework: Finish off §5.2 and then complete Exercises 5.1(c) and 5.3.

Negation as failure

How can we say “Mike likes all animals, but not snakes” in Prolog? One way to say it is

If X is a snake then ‘Mike likes X ’ is not true, otherwise, if X is an animal, then Mike likes X .

In Prolog, the above becomes the following:

```
likes(mike,X):-snake(X),!,fail.  
like(mike,X):-animal(X).
```

In the above, `fail` always fails.

Similarly, we can express the `different(X,Y)` as follows:

```
different(X,X):-!,fail.  
different(X,Y).
```

In general, we have the following relation, `not(P)`, defined as follows:

```
not(P):-P,!,fail.
```

```
not(P):-true.
```

Thus, we can rewrite the above as

```
like(mike,X):-animal(X), not snake(X).
```

and

```
different(X,X):-not X=Y
```

Homework: Exercises 5.5 and 5.6.

Nothing is free

The cut operator may increase the efficiency, but we will lose the correspondence between declarative and procedural meaning. For example, in the following program

```
p:-a,b.  
p:-c.
```

It means that $p \leftrightarrow (a \wedge b) \vee c$.

On the other hand, if we put in a cut,

```
p:-a,! ,b.  
p:-c.
```

The meaning changes to $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.

Finally, if we switch the two clauses, the meaning further changes to $p \leftrightarrow c \vee (a \wedge b)$.

We have a similar problem with the `not` operator. If we ask Prolog `?-not human(john)`, the answer will be 'yes'. This merely states that Prolog does not have enough information to answer the question, thus, it plays the safe side.

This is quite useful. For example, when we look at the scheduling for Concord Coach, if we could not find a bus at 7:15 p.m. on Monday, we would assume that there is no such a bus. Otherwise, the bus company would have to print quite a large schedule. This assumption is called *closed world assumption*, in the sense that everything that exists is stated within the program.

Borrowing this idea, anything that is not true according to the program is assumed to be false. The same idea happens to any database.