CAP 5602
Summer, 2011

Lesson 4: Loops

The topics

1. the cut and some of its uses
2. the while loop
3. the do until loop

1. The cut

The cut (!) is a way of controlling backtracking. If a rule contains a cut in its body, and the goal matches the head of the rule, then prolog cannot backtrack beyond the cut to resatisfy the goal.

Let us explain it in detail. Let G be the goal, $H :- A, !, B$ be the rule, and let us assume that it's the rule's turn to match G. Moreover, let's assume that G matches H. The list of goals is now $A, !, B$. If A fails, then prolog tries to match G with the clause that follows the rule. If A is satisfied, the fate of G rests with B. If B is satisfied, so is G. If B fails, G fails, because it cannot backtrack to resatisfy A, or to go to the next clause.

The cut is used for improving the efficiency of the program. The following 2 examples make use of the cut.

Example 1

Let us look at prog7, that computes a grade, based of the following table:

if the score is less than 50, the grade is f

if the score is greater than or equal to 50 but less than 75, the grade is c, and

if the score is greater than or equal to 75, the grade is a.

We use the built-in predicate ¡ to check for inequality.

```
% the use of the cut
% define a grade function that has value f if the score is < 50,
% c if 50 ≤ score < 75, and
% a if score ≥ 75
grade(Score,f):- Score < 50, !.        % clause 1
grade(Score,c):- Score < 75, !.        % clause 2
grade(Score,a).          % clause 3
```

The program has 3 clauses. The first two are rules and the 3rd fact.

If we ask

?- grade(40,G).

we get

G = f.

Prolog unifies the goal with the first clause by the substitution { Score = 40, G = f }. The goal now becomes 40 < 50, !. This goal 40 < 50 is true. The interpreter puts a period after G = f because the goal cannot be resatisfied.

If the query is

?- grade(60,G).

we get

G = c.

Again, the interpreter matches the goal with the first clause the substitution { Score = 60, G = f }, but it fails the subgoal 60 < 50, so it matches the goal grade(60,G) with the head of the second rule. It can backtrack because by failing condition 60 < 50, it DOES NOT PASS the cut!

The second match succeeds because 60 < 75.

Now, let us see what happens if we remove the cuts. We have the program below.

```
grade(Score,f):- Score < 50.        % clause 1
grade(Score,c):- Score < 75.        % clause 2
grade(Score,a).          % clause 3
```

If we ask the query

?- grade(40,G).

we get 3 values,

G = f ; G = c ; G = a.

because the matching continues after each ; return command.[1]

[1]The use of the cut is controversial, just like the use of go to in the imperative programming language. A prolog purist will insist that the clauses of the program could be shuffled like a deck of cards without affecting the program. So, he/she will wite the above program as

```
% a grading scheme
grade(Score,f):- Score < 50.
grade(Score,c):- Score >= 50, Score < 75.
grade(Score,a):- Score > 75.
```

Example 2

The cut is useful in defining the prolog negation. We want to define not(P) is such a way that whenever P is satisfied, not(P) fails, and when P fails, not(P) is satisfied.

The following program does the job.

```
% the negation
not(P):- P, !, fail.          % if P is satisfied, not(P) must fail
not(P).          % if P fails, not(P) must be satisfied
```

The first clause tries to satisfy P. If successful, the body of the clause fails, because the built-in predicate fail is always false. The cut prevents the interpreter from backtracing to P, or to the next clause. On the other hand, if P fails, the interpreter goes on to the next clause and not(P) is satisfied[2].

2. The while Loop

In an imperative language the while loop has the syntax below.

```
while (condition)
    loop_body;
```

We implement this loop with the fail predicate

```
% implementing a loop with fail
loop :- condition, loop_body, fail.
loop.
```

The loop_body must have a cut at the end of its clauses. The loop works as follows. First, prolog tries to satisfy the condition. If it fails, prolog goes to the second clause and the loop terminates. If the condition is true, prolog performs the loop_body and then fails. This means backtrack to the preceding goal, loop_body. But this is not possible because loop_body has a cut at the end. So, loop_body fails. Then, loop tries to resatisfy the condition. If successful, the loop_body is executed again, and again fails.

This is precisely how a while loop works. The following example uses a while loop.

Example

The loop processes all likes clauses and prints, on a new line, the message X likes Y.

for each pair likes(X,Y) is the database. The built-in predicate nl feeds a new line and write prints its argument.

[2]In many prolog compilers, not is a built-in predicate. We can define it SWI prolog.

```
% the likes predicate
likes(bob,mary).
likes(mary,bill).
likes(mary,peter).
likes(peter,mary).
% a loop that prints all likes of the form
% X likes Y.
loop:- likes(X,Y), printout(X,Y), fail.
loop.
printout(X,Y):- nl, write(X), write(' likes '), write(Y), write('.'),!.
```

When we enter the query
?- loop.
we get the output below.

```
bob likes mary.
mary likes bill.
mary likes peter.
peter likes mary.
true.
```

The program works as follows: it gets the first likes clause and processes it. Then fails forces printout to be resatisfied. The cut prevents the interpreter from doing it, so printout also fails. Then the interpreter tries to resatisfy likes with the second likes clause, and so, on until there are no more likes clauses. Then prolog executes the second clause of loop and prints true followed by a period. Had we ommitted the second clause of loop, the output would have been

```
bob likes mary.
mary likes bill.
mary likes peter.
peter likes mary.
false.
```

If we put the cut inside the rule,

```
loop:- likes(X,Y), printout(X,Y),!, fail.
```

the loop is executed at most onec because as soon as the likes goal is satisfied and the fact printed, fail makes the loop goal false. We will also mention that in many prolog compilers, the built-in predicates nl, and write

are not resatisfiable, so, the cut at the end of printout is not necessary. However, SWI yields a syntax error when we ommit it.

3. The do until loop

The do until loop has the syntax

```
do
    loop_body
until exit_condition
```

The mening is clear; repeat the loop_body until the exit condition is met.

We can implement this loop with the clauses below.

```
loop:- repeat, loop_body, exit_condition.
```

As with the while loop, loop_body has a cut at the end of its clauses. The built-in predicate repeat is always true. If the compiler does not come with the repeat predicate, we can write it as

```
repeat.
repeat:- repeat.
```

Example

The loop predicate below is a do until loop. The loop body prompts the user to enter a number or any non-integer to finish. The built-in predicate read get the input. When we use the predicate read we must put a period after the input. The loop body then processes the input using the predicate process. If the input is a number then process creates a new fact, data(X), where X is the input, and places it at the end of the database. The syntax of the assertz statement is assertz(the clause to be created). If we want to insert the new clause at the beginning of the database we use asserta. However, asserta would list the numbers in the reverse order, since the second number would be put on top of the first, the third on top of the second and so on.

The predicate print_numbers prints a header and then calls the while loop print_all_numbers. The tab(10) built-in predicate prints 10 blanks.

```
% the do loop
loop:- repeat, loop_body(X), not(integer(X)).
% the loop body
loop_body(X):- write('Enter an integer, a non-integer to stop.'),
        nl, write('Remember to put a period after the number > '),
        read(X),
        process(X),!.
```

```
% process(X) creates a fact number(X) if X is an integer
process(X):- integer(X), assertz(data(X)).
process(X).          % X is not an integer
% this printes the integers in the database
print_numbers :- tab(10), write('The numbers are'), nl, % the header
        tab(10), write('==============='), nl, nl,
        print_all_numbers.

% a while loop
print_all_numbers:- data(X), print_number(X), fail.
print_all_numbers.

print_number(X) :- nl, write('The next number is '), write(X), write('.'),
!
```

In the dialog below, the user entered 4 numbers, 1,2,3,4, in this order.

```
?- loop.
Enter an integer, a non-integer to stop.
Remember to put a period after the number > 1.
Enter an integer, a non-integer to stop.
Remember to put a period after the number > 2.
Enter an integer, a non-integer to stop.
Remember to put a period after the number > 3.
Enter an integer, a non-integer to stop.
Remember to put a period after the number > 4.
Enter an integer, a non-integer to stop.
Remember to put a period after the number > a.
true .
```

When the user enters the query ?- print_numbers followed by period and return, prolog prints the lines below.

```
    The numbers are
    ===============


The next number is 1.
The next number is 2.
The next number is 3.
```

The next number is 4.
true.