1 Controlling Backtracking

- Prolog does backtracking by default - this is good - but in certain scenarios it is more efficient to avoid it
- Improves program efficiency and performance
- Ability to control backtracking in situations when it is not really needed
- This is done via a *metalevel predicate* called the *cut*
- Affects procedural behaviour of programs by *pruning* the search tree
- it's use is highly controversial (from software engineering point of view) - as it diminishes the declarative style encouraged by logic programming
- however, widely used - as can greatly improve efficiency

```
member(X, [ X| _]) :- !.

member(X, [ _| Tail) :-
  member( X, Tail).
```

Notice the use of the *cut* in the first clause

2 Example 1

```
cinema_category(Age, children) :-
   Age =< 10.
cinema_category(Age, parental) :-
   Age > 10, Age =< 16.
cinema_category(Age, adult) :-
   Age > 16.
```

Q: cinema_category(14,Category)

A: Category = parental

no more solutions

Same relation, using the cut :

```
cinema_category(Age, children) :-
   Age =< 10, !.
cinema_category(Age, parental) :-
   Age > 10, Age =< 16, !.
cinema_category(Age, adult).
```

Q: cinema_category(14,Category)
A: Category = parental
no more solutions

This time, search tree has only 2 branches

3 Some rules

There are some branches, given the nature of a relation, are going to fail - idea is to *prune* them using a cut.

```
(1)  relation(...) :- ... .
(2)  relation(...) :- ... .
...
(i)  relation(...) :-
       t1,t2,t3,!,t4,t5,t6.

(i+1) relation(...) :- ... .
(n) relation(...) :- ... .
```

- Relation has n clauses (in order indicated)
- Relation (i) examined if relation (1) to (i-1) fail
- t1,t2,t3 fails - the cut doesn't take place so instance (i+1) of relation is chosen
- If t1,t2,t3 succeeds, the cut takes place and computation proceeds to prove t4,t5,t6
- if t4,t5,t6 fails, because of the cut, there is backtracking on t1,t2,t3, instance (i) of relation fails and instances (i+1) to (n) of relation are ignored (not considered as possible candidates to unification for a subgoal)

4 Example 2

Finding a maximum :

```
maximum(X,Y,X) :-
     Y < X.
maximum(X,Y,Y) :-
     Y >= X.
```

with a *cut*

```
maximum(X,Y,X) :-
  Y<X, !.
maximum(X,Y,Y).
```

Q: maximum(10,8,M)
A: M = 10

Draw a search tree to distinguish between them.

5 In general ...

Consider the following clause :

```
H :- B1, B2, ..., Bm, !, Bn.
```

- Assume clause invoked by a goal G that matched H
- When cut encountered in the definition of H - the system has satisfied B1 to Bm
- The current solution to B1 to Bm is *frozen*, and all possible remaining alternatives are discarded
- The goal G becomes committed to this clause
- Any attempts to match G with the head of some other clause is precluded

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
```

- Backtracking is possible within the goal list P,Q,R - BUT - as soon as the cut is reached, all alternate solutions of the goal list P,Q,R are suppressed
- The second C clause is also discarded
- Backtracking is however possible within S, T, U
- Cut is invisible from A - and only affects C

6 Lists and Cuts

Consider another *maximum* procedure, for determining, this time, the maximal number in a list of numbers.

Sometimes, also useful to change *declarative* semantics - to eliminate redundant computations. Consider definition 1 without the cut.

```
max( 0, []).
max( X, [X]).
max( X, [ X| Tail]) :-
  max( M, Tail),
  M =< X.
max( M, [ X| Tail]) :-
  max( M, Tail),
  M > X.
```

Problem : Maximal element will probably be have to be calculated twice if maximal element not in head of list.

Can change order of second and third clause - more efficient

Why is this more efficient ? *More likely for maximal element to be in tail* - however, STILL performing redundant computations.

Alternate definition with cut :

```
max2( 0, []).
max2( X, [X]).
max2 M, [X| Tail]) :-
 max2( M, Tail),
 M >= X, !.
max2( X, [X | _]).
```

- If cut removed, the procedure would provide incorrect solutions through backtracking
- Position of *last* clause is important
- Order of third and fourth clauses in the procedure may now no be changed
- the cut in this case cannot be removed without affecting the procedural correctness - and is called a **red** cut.
- In the *member* definition above, however, the cut could be removed without affecting correctness of procedure (though it would reduce efficiency - that's why we are using it in the first place!) - and is called a **green** cut

Q: max(M,[2,4,5,6,7,23,23,5,4]
A: M = 23
M = 23
no more solutions
Oops - what happens with multiple elements !!

Q: max2(M,[2,4,5,6,7,23,23,5,4]

A: M = 23
no more solutions

7 Order of clauses

- As mentioned previously, the reservations *against* cut - lose correspondence between declarative and procedural meaning of programs
- If no cut, and since program is declarative - possible to change *order* of clauses and goals
- This re-ordering may affect program *efficiency* or *termination* - BUT - has no influence on program correctness or declarative meaning (as the solution to a query is obtained by searching the program in a *top-down* manner, and answering the goals from *left to right*
- However, in programs with cut, a change in the order of the clauses *may* affect the meaning (declarative)
- We can get different results (depending on the query)

```
p :- a,b.
p :- c.
means p = (a and b) or c

p :- a, !, b.
p :- c.
means p = (a and b) or (~a and c)

swapping clauses

p :- c.
p :- a, !, b.
means p = c or (a and b)
```

8 Exchanging NOT with CUT

```
status(P, married) :-
  spouse(P,_).

status(P, single) :-
  \+ spouse(P,_).

spouse(P,S) :-
 wife(P,S).

spouse(P,S) :-
 husband(P,S).

wife(paul, mary).
wife(steve, heather).
wife(rob, donna).

husband(mary, paul).
husband(heather, steve).
husband(donna, rob).
```

Q: status(paul,X)
A: X = married
no more solutions

Q: status(ahmed,X)
A: X = single

- To avoid the repetition of identical computations such as the previous one, the *cut* is introduced and *status* rewritten as :

```
status(P, married) :-
 spouse(P,_),!.

status(P, single).
```

- **This pattern of rewriting clauses eliminates opposite conditions** - but the use of the cut
- Notice that the cut has not only replaced the *not* but also made the resulting program more efficient