

# Pruning the computation tree: the cut !

- **As you've seen, Prolog will exhaustively search the computation tree for solutions**
  - If a goal fails, OR the user inputs ';' at the prompt, then backtracking reverts to the last place in which a clause was chosen, and tries the next.
- **There are lots of advantages of this scheme, most notably, the ability to get multiple solutions to a query.**
- **However, it can be expensive:**
  - Sometimes there is only one solution, and it is a waste of time searching for others -- they don't exist!
  - Sometimes 'failure' after the first solution can take lots of time to infer
  - Memory resources are used to contain the computation tree for backtracking.
- **Prolog permits the computation tree to be pruned, ie. whole branches of the tree can be 'snipped' away**
  1. **If- Then - Else: (T -> A ; B) or (T -> A)**
  2. **one(P): call goal P, and only first solution will be used; backtracking fails**
    - treats user predicates like many builtin's eg. write,
  3. **"!" - the cut**

# The cut, !

- **The cut takes the form of a goal in a clause.**
  - only one cut per clause.
  - A predicate can have one or more clauses with cuts.
- **Scheme:**
  - 1. all the clauses before the first clause with a cut are executed with normal backtracking.**
  - 2. if the goals before the cut never succeed, the cut does not activate, and the subsequent clause is used, as normal.**
  - 3. if the goals before the cut succeed, the cut activates:**
    - a) backtracking back to goals before the cut cannot occur**
    - b) backtracking to subsequent clauses after the one with the cut cannot occur --> that clause with the activated cut is “committed”**
    - c) the goals after the cut are executed with normal backtracking**

## The cut - example

```
p(1).                %1
p(2).                %2
p(Y) :- q(3, Z), !, r(Z, Y). %3
p(4).                %4
```

```
q(2, 4).             r(5, 6).
q(3, 5).             r(5, 7).
```

- **Clauses 1, 2 are executed as normal**
- **In 3, the goal q(3,Z) executes; if it succeeds, then the ! is activated, and the goal r is executed as normal.**
- **However, in activating this cut...**
  - **clause 4 will not execute for this particular execution call**
  - **clause 3 will not backtrack to q again (in this goal inference)**
- **Note that backtracking in r(Z,Y) occurs as expected, and hence you can still get multiple solutions from clause 3**

# The cut

- **As you can see, the way the cut affects execution is somewhat arbitrary in nature**
- **There are two usages of cuts:**
  - (i) **Green cuts: cuts that prune execution branches that do not lead to solutions.**
  - (ii) **Red cuts: cuts that prune valid solutions**
    - **Green cuts are “good” ; red cuts are not.**
- **Example: ‘test\_if\_bad’**

# cut examples

Example: back to “test\_list”

```
test_list(L) :-  
    test_if_bad(L),  
    !,  
    write('Bad list, '), nl, fail.  
test_list(L) :-  
    write('Good list'), nl.
```

```
test_if_bad([ ]).  
test_if_bad(L) :- length(L, N), N > 100.  
test_if_bad(L) :- member(X, L), integer(X).
```

- a “green cut”, because we know that only one of test\_list clauses is valid for a given list

# Cut examples

- Actually, if-then-else can be emulated via a cut:

```
P :- (T -> Q ; R).      P :- T, !, Q.  
                        P :- R.
```

- We can also optimize 'test\_if\_bad':

```
test_if_bad([ ]) :- !.  
test_if_bad(L) :- length(L) > 100, !.  
test_if_bad(L) :- member(X, L), integer(X), !.
```

- These cuts mean that, if one of the clauses succeeds, backtracking to test\_if\_bad will cause immediate failure.
- This is desirable, because we don't need the interpreter to do any more execution upon that particular call to test\_if\_bad
- In other words, if you traced the old test\_if\_bad, you'd see lots of failures before the whole goal failed; with cuts, tracing will see the failure immediately!

# Cut examples

- **Example: a deterministic member:**
  - **deterministic clause: one that returns one solution per call only**

`member(A, [A|_]).`

`member(A, [_|R]) :- member(A, R).`

`memberd(A, [A|_]) :- !.`

`memberd(A, [_ | R]) :- memberd(A, R).`

- **Here, as soon as memberd clause 1 finds a match, it succeeds**
- **Subsequent backtracking to memberd then fails, due to the cut**
  - **in other words, we prevent memberd clause 2 from finding another match**

## cut examples

- An obvious bad use of the cut:

```
parent(P, C) :- father(P, C), !.
```

```
parent(P, C) :- mother(P, C).
```

- A red cut: we get the first solution from father, and never give another valid solution again, from either father or mother!
- if father fails, then mother is used.

- Variation:

```
parent(P, C) :- !, father(P, C).
```

```
parent(P, C) :- mother(P, C).
```

- This will permit all the father cases to be used; but mother is always ignored! ie. `parent(P, C) :- father(P, C).` (mother clause ignored)

# Cut examples

- **Example: sum the integers between 1 and n**

**sum\_to(1, 1) :- !.**

**sum\_to(N, Sum) :-**

**M is N - 1,**

**sum\_to(M, Tmp),**

**Sum is Tmp + N.**

- **Without the cut, backtracking proceeds to sum\_to(0,\_), sum\_to(-1,...) etc**
- **with the cut, when the case sum\_to(1,1) occurs, backtracking will not commence (via clause 2)**
- **again, a green cut: only one solution desired**

# Cuts

- **example: a meta-predicate that returns one solution to a goal:**

**one(P) :- call(P), !.                    or            one(P) :- P, !.**

- this is usually builtin to Prolog
- A convenient way of avoiding cuts
- **Cuts are extralogical: they may destroy a program’s logical “declarative” reading**
  - eg. 

```
test_list(L) :-  
    test_if_bad(L),  
    !,  
    write('Bad list, '), nl, fail.  
test_list(L) :-  
    write('Good list'), nl.
```
- **Read literally, the second clause says that all lists are good lists!**
- **Hence we must now ascertain the meaning of this predicate by inspecting what the cut is doing.**

# Cuts

- **Cuts are unavoidable in many programs: without them, the program can become too large and inefficient**
- **However, cuts may ruin a logic program's readability**
- **Careless use of cuts can make a Prolog program unintelligible (and hard to fix, understand, ...)**
- **A strategy:**
  - **1. First, try to make a declarative predicate if its feasible (correct, concise, efficient)**
  - **2. If this predicate is too difficult to write purely declaratively, use If-Then-Else or one(P) if they help**
  - **3. Otherwise, use a cut if it is a green cut**
  - **4. red cut: use as rarely as possible; and document their function!**