Prolog

Lists, arithmetic, data structures

Examples, downloadable Prolog, on-line interpreter and more on SWI Prolog
https://www.swi-prolog.org/

LISTS

In Prolog a list is represented by a term defined as follows:

- [ ] is the empty list
- [t1|t2] is the list whose first element (head) is t1 and the rest of the list (tail) is t2.

Shorthands:

[a, b] means [a | [b|[]]]

[a,b| X] means [a | [b| X]]

Example: "length"

```
% length (L, N):   N is the length of list L

length([], 0).
length([ _ |L],s(N)) :- length(L, N).
```

"_" represents the anonymous variable

```
?- length([X|[Y|[a|[]]]],R).
?- length([a,b,Z],R).
?- length([a|x],R).
```

Example: "member"

```
% member(X, L)   "X is a member of list L"

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

?- member( a, [c|[b|[a|[]]]]).
?- member( a, [a|[b|[a|[]]]]).
?- member( a, [c|[Z|[a|[]]]]).
?- member( a, [c|Z]).
```

Example: "append"

```
% append(L1, L2, L)        L is the concatenation of
%                          L1 and L2

   append([], L, L).
   append([X|L1], L2, [X|L]) :- append(L1,L2,L).

?- append([a,b], [c], Z).
?- append([X,a,Y], [b,c], Z).
?- append([a,b],[c|Y], Z).
?- append([X],[c|Y], [a,c,e,f]).
```

Types

Note Prolog is not typed!.

```
?- member( a, [a|I_am_not_a_list]).
   succeeds!

append([], I_am_not_a_list, L).
   succeeds with c.a.s. {L/I_am_not_a_list}
```

The predicate list

We can control the list type by introducing the list/1 predicate:

```
list([]).
list([ _ |L]) :- list(L).
```

Example: "delete"

```
% delete(X,LB,LS)  list LS is obtained from LB by deleting
% an element which unifies with  X.

delete(X,LB,LS) :-      append(F, [X|R], LB),
                        append(F,R,LS).

delete(X, [X|R], R).
delete(X, [Y|L], [Y|R]) :- delete(X,L,R).
```

Exercise

Describe the behaviour of the Prolog interpreter for each of the following goals.

?- delete(a, [c,d,a,b,a], Z).

?- delete(a, [X|Z], [a,b,a,c]).

?- delete(X, [a,f,a,c], Z).

?- delete(a, [b,X,c], Y).

?- delete(a, [b|X], Y).

Arithmetic

Integers, floats, numbers for a given basis x' are pre-defined constants:

integer: 0, 1, 9977, -79393

float: 4.5e3, 1.0, -0.5e+7

base x': 2'101, 8'174

Several numerical functions (+, _, *, abs, max …) are also Pre-defined

Arithmetic built-in's

There are also some pre-defined predicates (built-in's) which can be used in expressions

Note: every variable which appears in an expression must be ground when evaluated

Z is X	the value of X is unified with Z;
X =:= Y	the value of X and Y are equal;
X=\= Y	the value of X and Y are different;
X > Y	the value of X isgreater than the value of Y.

Predicate " is": examples

p(X) :- X is 3+5.

?- p(X).

X=8

p(X) :- Y is 3+5, X is Y.

?- p(X).

X=8

p(X) :- X is Y, Y is 3+5.

?- p(X).

error

"length" with arithmetic

```
%  length (L, N)   "N is the length of list L"

length([], 0).
length([ _ |L], N1) :- length (L, N), N1 is N+1.
```

Example : merge

```
% merge(L1,L2,L)        L is the merge of the ordered
%                       lists L1 and L2.

merge([],Y,Y).
merge(Y,[],Y).
merge([X|L1],[Y|L2],[X|L]) :- X ≤ Y, merge(L1,[Y|L2],L).
merge([X|L1],[Y|L2],[Y|L]) :- X >Y,  merge([X|L1],L2,L).
```

Example: split

```
% split(L,L1,L2)  L= L1.L2 and  lungh(L1)=lungh(L2)

split([X|L],[X|L1],L2) :- split(L, L2, L1).
split([],[],[]).
```

split([a,b,c],L1,L2)
= {L1/[a|L1']} => split([b,c],L2,L1')
= {L2/[b|L2']} => split([c],L1',L2')
= {L1'/[c|L1"]} => split([],L2',L1")
= {L2'/[],L1"/[]} =>'

Example: mergesort

```
% merge_sort(L,LSort) LSort is the ordered version of L

merge_sort([],[]).
merge_sort([X],[X]).
merge_sort([X,Y|R],Lsort) :- split([X,Y|R], L1, L2),
                             merge_sort(L1,L1Sort),
                             merge_sort(L2,L2Sort),
                             merge(L1Sort, L2Sort, LSort).
```

n.b. if lungh(L) ≥2 then, after split, lungh(L1)<lungh(L) and lungh(L2)<lungh(L).

Data structures

In Prolog data structures are represented by using terms.

We have to decide the representation, the contructors and the predicates.

Example: binary tree

Empty tree is represented by the constant "void"

We use the function tree/3 for constructing a non empty tree:

tree(Root,Left,Right).

Binary tree: examples

```
% empty(T)  test for empty tree
empty(void).

% build_tree(X,T1,T2, T): is the tree built by using the
% Root X, the left sub-tree T1 and the right sub-tree T2
build_tree(X, T1, T2, tree(X,T1,T2)).

% left_tree(T,L): L is the left-sub-tree of T
left_tree(tree(Root,Left,Right), Left).
```

Pre-order tree traversal

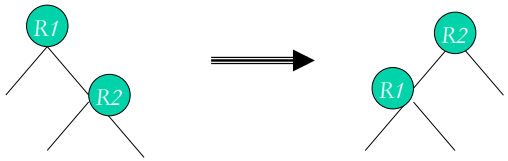
```
% pre_visit(T,L)        L is the list of the (keys of)
%  nodes of T obtained by a pre-order traversal

pre_visit(void,[]).
pre_visit(tree(Root, Left, Right), [Root|L]) :-
                        pre_visit(Left, L1),
                        pre_visit(Right,L2),
                        append(L1,L2,L).
```

Example: rotate_left

```
% rotate_left(T,Tr)      Tr is obtained from T
    by a left rotation (see picture)

rotate_left ( tree(R1,Alfa,tree(R2,Beta,Delta)),
              tree(R2,tree(R1,Alfa,Beta),Delta)).
```



The predicate ! (cut)

Utility of cut (!)

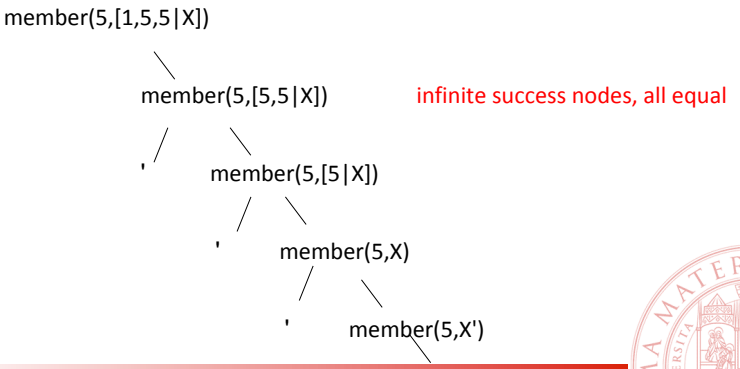
The SLD tree which is searched by the Prolog interpreter for a given goal G and program P can contain several repeated paths and failed paths

This is source of inefficiency

The ! allows us to prune part of the useless paths.

Example 1

```
member(X, [X| _ ]).
member(X, [ _ |L]) :- member(X,L).
?- member(5,[1,5,5|X]).
```



infinite success nodes, all equal

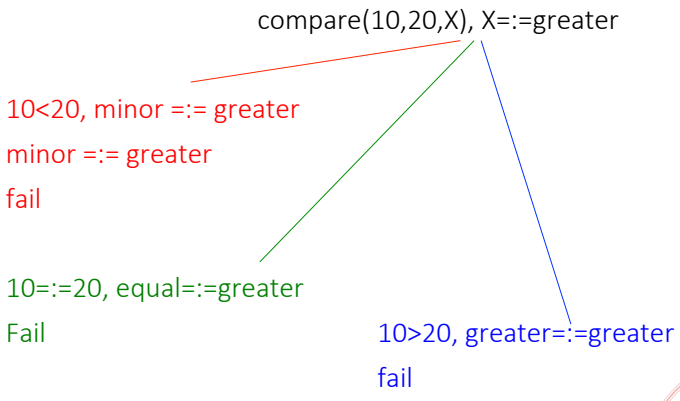
esempio 2

```
compare(X,Y,minor) :- X<Y.
compare (X,Y,equal) :- X=:=Y.
compare(X,Y,greater) :- X>Y.
?- compare(10,20,X), X=greater.
```

We note that:

- The goal fails
- Note that the three rules are mutually exclusive: one can then avoid to continue the search after the first attempt (i.e. the first failure).

Example



Il cut

The problem with the previous examples comes from the search strategy of the prolog interpreter who always tries to inspect the entire SLD tree.

To modify this strategy the programmer can use the predicate cut (!) which can be inserted into the body of any clause.

The execution of the predicate cut has the effect of pruning the SLD tree, cutting redundant or dead branches according to the following rules.

Operational meaning of !

Let

! , G1

be a goal that has cut (!) as leftmost atom and let

c: H :- A, ! , B.

the the clause which introduced that cut occurrence.

The execution of the predicaste ! makes definitive any choice which has been made in the computation done to solve the atom that has been unified with H, that is, makes definitive the choice of clause c and the choices made to solve A.

Example

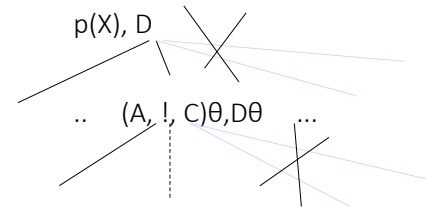
p(s_1) :- B1.

...

p(s_i):- A, !, C.

...

p(s_n) :- Bn



G = p(X), D

!, Cδ, Dδ

|

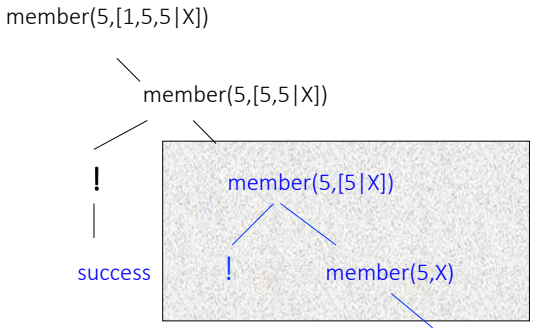
Cδ, Dδ

The part of the tree deleted will not be anymore analysed (also in case the goal Cδ, Dδ fails)

Example

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

member(X,[-|L]) :- member(X,L).



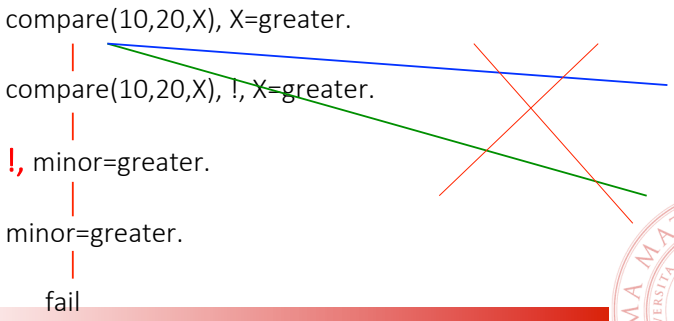
The grey part is cut (no searched) . . .

Example

compare(X,Y,minor) :- X<Y, !.

compare(X,Y,uguale) :- X=:=Y, !.

compare(X,Y,greater) :- X>Y.



Semantics of cut

In the previous examples the introduction of the cut did not alter the correspondence between operational semantics and declarative semantics: we obtain the same set of solutions (c.a.s) with and without the execution of the cut.

This is not always true: introducing the cut can alter the meaning of the program by changing the set of solutions.

The predicate fail

The fail predicate is another predefined predicate that, when executed, generates a failure.

Example

```
not_member(X, []).
not_member(X, [X|_]):- !,fail.
not_member(X,[-|L]) :- not_member(X,L).
```

When X unifies with the head of the list [X|-] first the cut is evaluated and this cuts all subsequent choices, then the fail predicate that aborts the computation (desired effect).

It's a way to obtain negation!

Example

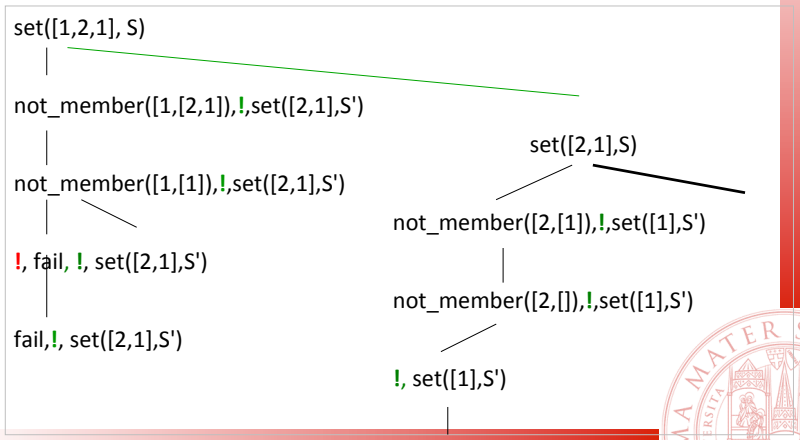
```
% set(L,S)     S is obtained from the set L by
%              removing duplicates

set([],[]).
set([X|L],[X|S]) :- not_member(X,L),!,set(L,S).
set([X|L], S) :- set(L,S).
```

In a SLD tree more cuts may be needed. Each cut, when evaluated, can generate a pruning.

It is important a carefull analysis of the effects of different cuts

Example



Esercizio

```
% Define flatten(L,L')
% where  L is a list of lists and L' is the flatten
% version of L
% e.g. L=[[a,b],[c],[[d,e],f]]  L=[a,b,c,d,e]
```

```
flatten([X|L],F) :-  flatten(X,F1), flatten(L,F2),
                     append(F1,F2,F).
flatten(X,[X]) :- constant(X), X =!=[].
flatten([],[]).
```

flatten_dl

```
% Let us replace each result list by a difference list
flatten_dl([X|L],F-Z) :-  flatten_dl(X,F1-Y),
        flatten_dl(L,F2-W), append_dl(F1-Y,F2-W,F-Z).
flatten_dl(X,[X|Z]-Z) :- constant(X), X =!=[].
flatten_dl([],Z-Z).
```

Risolvendo append nella prima clausola Y=F2, W=Z e F1=F:

```
flatten_dl([X|L],F-Z) :- flatten_dl(X,F-F2), flatten_dl(L,F2-Z).
flatten_dl(X,[X|Z]-Z) :- constant(X), X =!=[].
flatten_dl([],Z-Z).
```

Exercise

Define the data type queue with the opreations in_queue and de_queue.