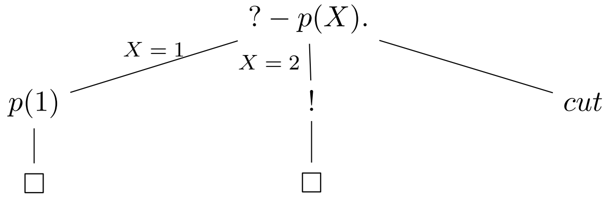
Functional and Logic Programming Fall 2017

S11 : Prolog (DCG)

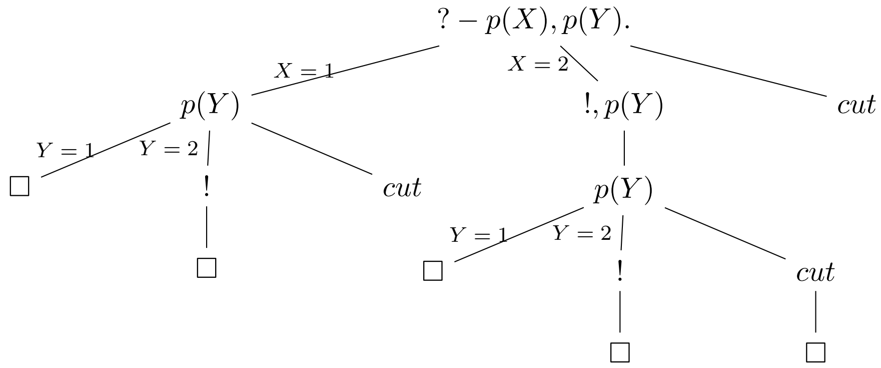
Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

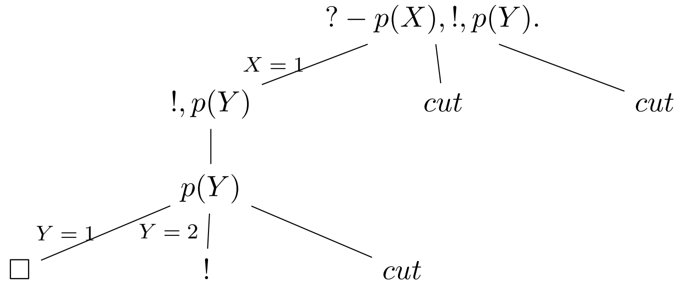
(a)



(b)



(c)



Exercise 2

separate/4

```
% separate(+X,+L:list,?L1,?L2), succeed if
% - L1 is the list of all element from L less or equals than X
% - L2 is the list of all element from L greater than X
separate(_,[],[],[]).
separate(X,[Y|YS],[Y|L1],L2) :-
    Y =< X,
    separate(X,YS,L1,L2).
separate(X,[Y|YS],L1,[Y|L2]) :-
    Y > X,
    separate(X,YS,L1,L2).

% Ex2.1.b
% separateCut(+X,+L:list,?L1,?L2), succeed if
% - L1 is the list of all element from L less or equals than X
% - L2 is the list of all element from L greater than X
separateCut(_,[],[],[]) :- !.
separateCut(X,[Y|YS],[Y|L1],L2) :-
    Y =< X,
    !,
    separateCut(X,YS,L1,L2).
separateCut(X,[Y|YS],L1,[Y|L2]) :-
    separateCut(X,YS,L1,L2).
```

myUnion/4

```
% myUnion(+E1:list,+E2:list,?E3), E1 and E2 are seen as set, their elements are disjoint 2 by 2
% myUnion/3 succeed if and only if E3 is the union of the set E1 and E2
myUnion([],Acc,Acc).
myUnion([X|XS],Acc,Res) :-
    \+ member(X,Acc),
    myUnion(XS,[X|Acc],Res).
myUnion([X|XS],Acc,Res) :-
    member(X,Acc),
    myUnion(XS,Acc,Res).

% Ex2.2.b
% myUnionCut(+E1:set,+E2:set,?E3), E1 and E2 are seen as set, their elements are disjoint 2 by 2
% myUnionCut/3 succeed if and only if E3 is the union of the set E1 and E2
myUnionCut([],Acc,Acc) :- !.
myUnionCut([X|XS],Acc,Res) :-
    member(X,Acc),
    !,
    myUnionCut(XS,Acc,Res).
myUnionCut([X|XS],Acc,Res) :-
    myUnionCut(XS,[X|Acc],Res).
```

myIntersection/4

```
% myIntersection(+E1:set,+E2:set,?E3), E1 and E2 are seen as set, their elements are disjoint 2 by 2
% myIntersection/3 succeed if and only if E3 is the union of the set E1 and E2
myIntersection([],_,[]).
myIntersection(_,[],[]).
myIntersection([X|XS],E2,[X|E3]) :-
    member(X,E2),
    myIntersection(XS,E2,E3).
myIntersection([X|XS],E2,E3) :-
    \+ member(X,E2),
    myIntersection(XS,E2,E3).

% Ex2.3.b
% myIntersectionCut(+E1:set,+E2:set,?E3), E1 and E2 are seen as set, their elements are disjoint 2 by 2
% myIntersectionCut/3 succeed if and only if E3 is the union of the set E1 and E2
myIntersectionCut([],_,[]) :- !.
myIntersectionCut(_,[],[]) :- !.
myIntersectionCut([X|XS],E2,[X|E3]) :-
    member(X,E2),
    !,
    myIntersectionCut(XS,E2,E3).
myIntersectionCut([_|XS],E2,E3) :-
    myIntersectionCut(XS,E2,E3).
```

maxMin/4

```
% minMaxA(+L:list[numeric],?Max,?Min), succeed if Max is the maximum value of L and Min the minimal
↪  value of L
% L must have at least one element
minMaxA([X],X,X).
minMaxA([X|XS],Max,Min) :-
    minMaxA(XS,Max2,Min2),
    (X > Max2 -> Max = X ; Max = Max2),
    (X < Min2 -> Min = X ; Min = Min2).

% Ex2.4.b
% minMaxB(+L:list[numeric],?Max,?Min), succeed if Max is the maximum value of L and Min the minimal
↪  value of L
% L must have at least one element
minMaxB([X],X,X) :- !.
minMaxB([X|XS],Max,Min) :-
    minMaxB(XS,Max2,Min2),
    (X > Max2 -> Max = X ; Max = Max2),
    (X < Min2 -> Min = X ; Min = Min2).

% Ex2.4.c
% minMaxC(+L:list[numeric],?Max,?Min), succeed if Max is the maximum value of L and Min the minimal
↪  value of L
% L must have at least one element
minMaxC([X|XS],Max,Min) :- minMaxC(XS,X,X,Max,Min).

minMaxC([],AccMax,AccMin,AccMax,AccMin) :- !.
minMaxC([X|XS],AccMax,AccMin,Max,Min) :-
    X > AccMax,
    !,
    minMaxC(XS,X,AccMin,Max,Min).
minMaxC([X|XS],AccMax,AccMin,Max,Min) :-
    X < AccMin,
    !,
    minMaxC(XS,AccMax,X,Max,Min).
minMaxC([_|XS],AccMax,AccMin,Max,Min) :-
    minMaxC(XS,AccMax,AccMin,Max,Min).

%%% utils for minMax

% max(+A:numeric,+B:numeric,?C), succeed of C is the maximum value between A and B
```

```
max(A,B,A) :- A >= B, !.
max(A,B,B) :- A < B.

% min(+A:numeric,+B:numeric,?C), succeed of C is the minimum value between A and B
min(A,B,A) :- A =< B, !.
min(A,B,B) :- A > B.
```

In addition, we use the following *go*/1 in order to simply visualize the search tree of the interpreter and show that each implementation reduce the number of search.

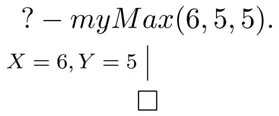
```
go :-
    % separate
    printTrace(separate(3,[1,2,3,4,5],_,_)),
    printTrace(separateCut(3,[1,2,3,4,5],_,_)),

    % myUnion
    printTrace(myUnion([1,2,3,4,5],[3,4,5,6,7],_)),
    printTrace(myUnionCut([1,2,3,4,5],[3,4,5,6,7],_)),

    % myIntersection
    printTrace(myIntersection([1,2,3,4,5],[3,4,5,6,7],_)),
    printTrace(myIntersectionCut([1,2,3,4,5],[3,4,5,6,7],_)),

    % minMax
    printTrace(minMaxA([1,4],4,1)),
    printTrace(minMaxB([1,4],4,1)),
    printTrace(minMaxC([1,4],4,1)),
    nodebug,
    !.

printTrace(Goal) :-
    functor(Goal,F,A),
    writef("+----------------------------+"),nl,
    writef("trace %t/%t",[F,A]), nl,
    leash(-all),
    visible(-all), visible(+call), visible(+redo), visible(+exit),
    trace,
    Goal,
    notrace,
    nodebug,
    writef("+----------------------------+"),nl.
```

Exercise 3

The following implementation of *myMax*/3

```
myMax(X,Y,X) :- X>=Y,!.
myMax(_X,Y,Y).
```

would fail for the query `?- myMax(6,5,5)` :



The interpreter would only explore one branch, because he can't unify $myMax(6,5,5)$ with $myMax(X,Y,X)$, so it would only explore the branch where we unify $myMax(6,5,5)$ with $myMax(Y,X,X)$ and then the maximal between 6 and 5 would be 5, which is wrong.
We should use the following implementation :

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

Exercise 4

```
% DCG for the grammar a*bca*
ex4_1 --> aStar, [b], [c], aStar.

aStar --> [a], aStar.
aStar --> [].

% Ex4.2
% DCG for the grammar (a.|bc)*uu*
ex4_2 --> first, uPlus.

first --> [a], [Any], {char_type(Any,ascii)} , first.
first --> [b], [c], first.
first --> [].

uPlus --> [u], uStar.

uStar --> [u], uStar.
uStar --> [].
```

Exercise 5

```
exp(exp(or(exp(Left),Right))) --> subExp(Left), ['|'], exp(Right), !.
exp(exp(AST)) --> subExp(AST).

subExp([Head|Tail]) --> term(Head), subExp(Tail).
subExp([AST]) --> term(AST).

term(iter(AST)) --> subTerm(AST), ['*'].
term(AST) --> subTerm(AST).

subTerm(point) --> ['.'].
subTerm(AST) --> ['('], exp(AST), [')'].
subTerm(char(C)) --> [C], {char_type(C,ascii), \+ member(C,['*',')','(','.'])}.
```

We use the following *go*/0 predicate in order to “test” our implementation :

```
go :-
    L1 = ['a','b','(','c','c',')','*'],
    exp(Ast,L1,[]),
    writef("%q",[Ast]), nl,
    Ast = exp([char(a),char(b),iter(exp([char(c),char(c)]))]),
    L2 = ['a'],
    exp(Ast2,L2,[]),
    writef("%q",[Ast2]),nl,
    Ast2 = exp([char(a)]),
    L3 = ['a','b','c'],
    exp(Ast3,L3,[]),
    writef("%q",[Ast3]),nl,
    Ast3 = exp([char(a),char(b),char(c)]),
    L4 = ['a','b','*','.','a'],
    exp(Ast4,L4,[]),
    writef("%q",[Ast4]),nl,
    Ast4 = exp([char(a), iter(char(b)), point, char(a)]),
    L5 = ['(','a','b',')','*','c'],
    exp(Ast5,L5,[]),
    writef("%q",[Ast5]),nl,
    Ast5 = exp([iter(exp([char(a), char(b)])), char(c)]),
    L6 = ['a','b','|','c','d'],
    exp(Ast6,L6,[]),
    writef("%q",[Ast6]),nl,
    Ast6 = exp(or(exp([char(a), char(b)]), exp([char(c), char(d)]))),
    L7 = ['a','(','b','c','|','d','e',')','*','g'],
    exp(Ast7,L7,[]),
    writef("%q",[Ast7]),nl,
    Ast7 = exp([char(a), iter(exp(or(exp([char(b), char(c)]), exp([char(d), char(e)])))), char(g)]).
```