Functional and Logic Programming Fall 2017

S12 : Prolog

Professor : Le Peutrec Stephane
Assistant : Lauper Jonathan

Exercise 1

myMap

```
% myMap(+Pred, +LS:list, ?LR), succeed if LR is the LS list mapped with the predicate Pred
% Pred is at least of arity 2, because we have to deduce Y from X with Pred
% for example, myMap(power(2),[1,2,3,4],LS) will succeed with LS = [1,4,9,16].
myMap(_,[],[]) :- !.
myMap(Pred,[X|Xs],[Y|Ys]) :-
    call(Pred,X,Y),
    myMap(Pred,Xs,Ys).
```

myPartition

```
% myPartition(+Pred, +L:list, ?Included, ?Excluded), succeed if
% - Included is the list of all element from L that are true with Pred
% - Excluded is the list of all element from L that are false with Pred
myPartition(_,[],[],[]) :- !.
myPartition(Pred,[X|Xs],[X|Included],Excluded) :-
    call(Pred,X),
    myPartition(Pred,Xs,Included,Excluded).
myPartition(Pred,[X|Xs],Included,[X|Excluded]) :-
    \+(call(Pred,X)),
    myPartition(Pred,Xs,Included,Excluded).
```

filter

```
% filter(+Pred, +L:list, ?Filtered)
% succeed if Filtered is the sublist of L where forall x E Filtered, Pred(x)
filter(_,[],[]) :- !.
filter(Pred,[X|Xs],[X|Ys]) :-
    call(Pred,X),
    filter(Pred,Xs,Ys).
filter(Pred,[X|Xs],Ys) :-
    \+(call(Pred,X)),
    filter(Pred,Xs,Ys).
```

filterWithFindall

```
% filterWithFindall(+Pred, +L:list, ?Filtered)
% succeed if Filtered is the sublist of L where forall x E Filtered, Pred(x)
filterWithFindall(Pred,L,Filtered) :-
    findall(X,(member(X,L),call(Pred,X)),Filtered).
```

myIntersection

```
% myIntersection(+E1,+E2,?E3), succeed if E3 is the set resulting from the intersection of E1 and E2
myIntersection(E1,E2,E3) :-
    findall(X,(member(X,E1),member(X,E2)),E3).
```

Some utility predicats

```
%%% Utils

% add(+N:number,+X:number,?Y), succeed if Y is N + X
add(N,X,Y) :- Y is N + X.

% odd(N), succeed if N is odd
% PRE : N >= 0
odd(0) :- !, fail.
odd(1) :- !.
odd(N) :- N2 is N - 2, odd(N2).

% even(N), succeed if N is even
% PRE : N >= 0
even(0) :- !.
even(1) :- !, fail.
even(N) :- N2 is N - 2, even(N2).
```

Exercise 2

```
:- initialization(go).

go :-
    retractall(house(_,_,_,_,_,_)),
    houseComposition(C,A,B,F,N),
    assertHouse(C,A,B,F,N,1).

assertHouse([],[],[],[],[],_) :- !.
assertHouse([C|Cs],[A|As],[B|Bs],[F|Fs],[N|Ns],I) :-
    assertz(house(I,C,N,A,B,F)),
    I1 is I + 1,
    assertHouse(Cs,As,Bs,Fs,Ns,I1).

houseComposition(C,A,B,F,N) :-
    C = [_C1,C2,_C3,_C4,_C5],
    A = [_A1,_A2,_A3,_A4,_A5],
    B = [_B1,_B2,B3,_B4,_B5],
    F = [_F1,_F2,_F3,_F4,_F5],
    N = [N1,_N2,_N3,_N4,_N5],
    % hint1
    N1 = norwegian,
    % hint2
    C2 = blue,
    % hint3
    B3 = milk,
    % hint4
    sameIndex(red,english,C,N),
    % hint5
    sameIndex(green,coffee,C,B),
    % hint6
    sameIndex(yellow,kool,C,F),
    % hint7
    tail(C,Ct),
    sameIndex(green,white,C,Ct),
    % hint8
    sameIndex(spain,dog,N,A),
    % hint9
    sameIndex(ukrainian,tea,N,B),
    % hint10
    sameIndex(japanese,craven,N,F),
    % hint11
    sameIndex(oldGold,snail,F,A),
    % hint12
    sameIndex(gitane,wine,F,B),
    % hint13
    tail(F,Ft),
    tail(A,At),
    (sameIndex(chesterfield,fox,F,At); sameIndex(chesterfield,fox,Ft,A)),
    % hint14
    (sameIndex(kool,horse,F,At); sameIndex(kool,horse,Ft,A)),
    % question 1
    sameIndex(_Nwater,water,N,B),
    % question 2
    sameIndex(_Nzebra,zebra,N,A),
    !.

sameIndex(E1,E2,[E1|_],[E2|_]).
sameIndex(E1,E2,[_|L1],[_|L2]) :-
    sameIndex(E1,E2,L1,L2).

tail([_|Xs],Xs).

drink(N,D) :- house(_,_,N,_,D,_).

hasAnimal(N,A) :- house(_,_,N,A,_,_).
```

Exercise 3

We would use the following ebnf grammar for our “grammar” syntax :

```
grammar ::= rule grammar | rule
rule ::= rule_name "->" rule_body
rule_body ::= rule_body_part_rec "|" rule_body | rule_body_part_rec
rule_body_part_rec ::= rule_body_part rule_body_part_rec | rule_body_part
rule_body_part ::= rule_name | atomName

rule_name ::= "r_" anyChar*
atomName ::= (lowerCase - "r") (anyChar - "_") lowerCase* | lowerCase anyChar*

lowerCase ::= "a" | "b" | "c" | "d" | "e" | "f" | "g"
            | "h" | "i" | "j" | "k" | "l" | "m" | "n"
            | "o" | "p" | "q" | "r" | "s" | "t" | "u"
            | "v" | "w" | "x" | "y" | "z"

anyChar ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

Note :

- The `anyChar` rule correspond any Unicode character, excluding the surrogate blocks, FFFE, and FFFF. It as been taken from `https://www.w3.org/TR/xml/#NT-Char`.
- We use the Extended Backus-Naur Form (EBNF) to specify the syntax.
- The syntacic diagram are available from figure 1 to 5.
- We don’t show the diagram for the `rule_name`, `atomName` or `lowerCase`.

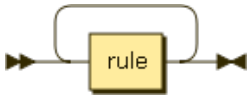

Figure 1: ENBF diagram for the rule “grammar”

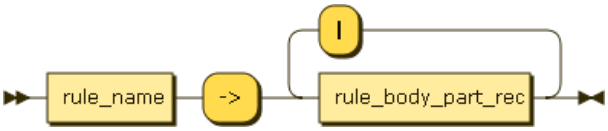

Figure 2: ENBF diagram for the rule “rule”

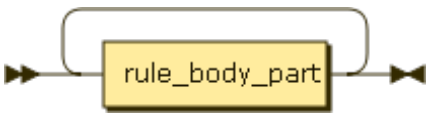

Figure 3: ENBF diagram for the rule “rule_body_part_rec”

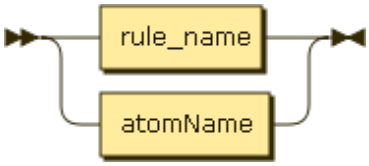

Figure 4: ENBF diagram for the rule “rule_body_part”

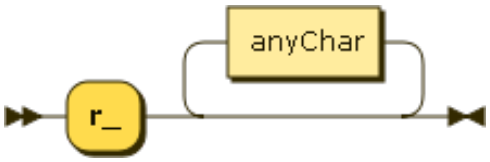

Figure 5: ENBF diagram for the rule “rule_name”

Implementation

```
grammar(Rules) --> rule(Rule), grammar(Rs), {append(Rule,Rs,Rules)}.
grammar(Rules) --> rule(Rules).

rule(Rules) --> ruleName(Identifier), ['->'], ruleBody(Is),
               {processBody(Identifier,Is,Rules)}.

ruleBody([Identifier|Is]) --> ruleBodyPartRec(Identifier),['|'] , ruleBody(Is).
ruleBody([Identifier]) --> ruleBodyPartRec(Identifier).

ruleBodyPartRec([Id|Ids]) --> ruleBodyPart(Id), ruleBodyPartRec(Ids).
ruleBodyPartRec([Id]) --> ruleBodyPart(Id).

ruleBodyPart(Identifier) --> ruleName(Identifier).
ruleBodyPart([Identifier]) --> atomName(Identifier).

ruleName(Identifier) --> [Identifier], {atom_chars(Identifier,L), L = [r,'_'|_]}.

atomName(Identifier) --> [Identifier], {atom_chars(Identifier,L) , \+(L = [r,'_'|_]), \+(L = [-,>])}.

processBody(_,[],[]).
processBody(Id,[RB|RBs],[Rule|Rs]) :-
    transformListInTermWithoutFunctor(RB,Body),
    Rule = -->(Id,Body),
    processBody(Id,RBs,Rs).

generateRules(L) :-
    grammar(Rs,L,[]),
    processRules(Rs),
    !.

processRules([]) :- !.
processRules([R|Rs]) :-
    expand_term(R,T),
    assertz(T),
    processRules(Rs).

transformListInTermWithoutFunctor(L,P) :-
    addCommaFunctor(L,R),
    transformListWithCommas(R,P).

addCommaFunctor([X],[X]):-!.
addCommaFunctor([X,Y],[',',X,Y]):-!.
addCommaFunctor([X|R],[',',X|[T]]) :- addCommaFunctor(R,T).

transformListWithCommas([',',X,[','|Y]],P) :-
    !,
    transformListWithCommas([','|Y],F),
```

```
    P =..[',',X,F].
transformListWithCommas([',',X,Y],P) :-
    !,
    P =..[',',X,Y].
transformListWithCommas([X],X).
```