**PLF WRITTEN 8 ianuarie2020 ex 1**

A close up of a paper

Description automatically generated

F([],0).

F([H|T],S):-

F(T,S1),

S1<H,

!,

S is H.

F([\_|T],S):-

F(T,S1),

S is S1.

SOLUTION:

F([],0).

F([H |T],S):-

F(T,S1),

F\_aux([H|T],S1,S).

F\_aux([H|\_],S1,S):-

S1<H,

!,

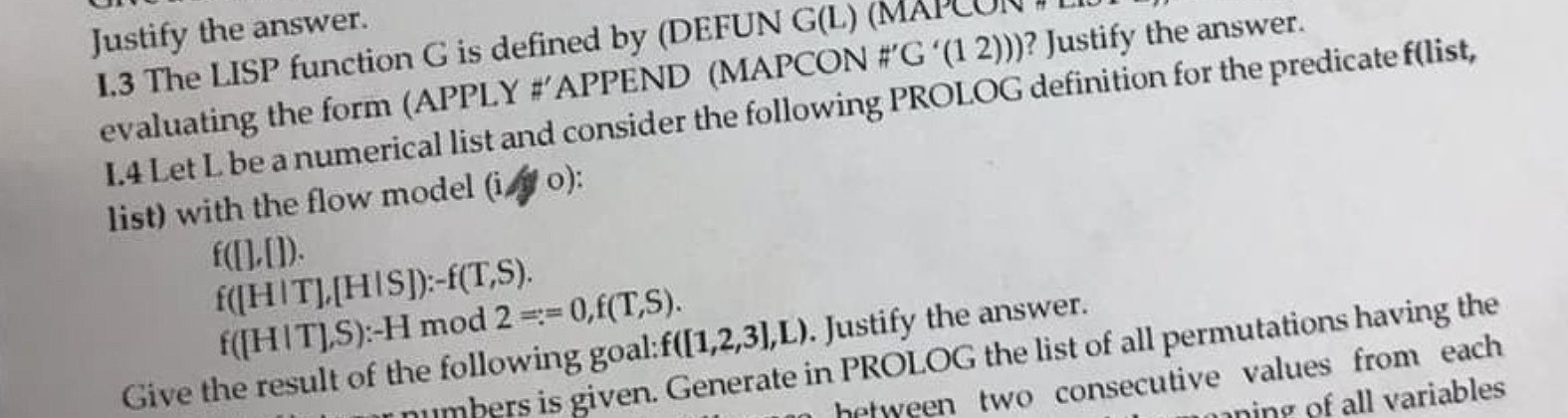
S is H.

F\_aux(\_,S1,S):-

S is S1.

%We are using an auxiliary function to avoid the recursive call in both branches.Instead of this recursive call , I created an auxiliary function which will have 3 parameters , S , S1 and the list.

The auxiliary function will have the role of computing the result using the parameters given.This will separate the decision logic from the base case and recursive call making the code more readable also.



f([],[]).

f([H|T],[H|S]):-

F(T,S).

f([H|T],S):-

H mod 2 =:= 0,

f(T,S) .

The result of the following goal f([1,2,3],L) will first be [1,2,3] because all the elements in the list will match the first branch. No elements will be eliminated in this case. Since there is no !(cut operator), Prolog continues searching for additional solutions. It explores the second branch,

which skips the elements that satisfy H mod 2=:=0.Only element 2 matches this condition , so the second solution will be L=[1,3].

A close up of a paper

Description automatically generated

SOLUTION

%generate\_all\_permutations\_matching\_the\_conditions(L,R)

%L -represents the input list , with the elements we are generating the permutations with

%R – output – a list with all permutations which satisfy the conditions

%flow\_model(i,o)

generate\_all\_permutations\_matching\_the\_conditions(L,R):-

findall(X,(generate\_permutations(L,X),check\_condition(X,1)),R).

%generate\_permutations(L,R)

%L- represents the input list , with the elements we are generating the permutations with

%R -resulted permutation

%mathematical\_model

%generate\_permutations(L) = { [], if L is empty

% H U generate\_permutations(Rest) , where H is a random

%element chosen from the list and Rest is the list without the chosen element

%

generate\_permutations([],[]).

generate\_permutations(L , [H|R]):-

select(H,L,REST),

generate\_permutations(REST,R).

%check\_condition (L,R)

%L -represents the input list to check the condition

%R – Result(1 if condition is satisfied , 0 otherwise)

%flow model(I,o)

%mathematical model

%check\_condition(l1l2…ln) = { 1 , n =1

% 0 , abs(l1-l2) > 3

% check\_condition(l2l3..ln) , otherwise

check\_condition([\_],1).

check\_condition([H1,H2|\_],0):-

abs(H1-H2) > 3,

!.

check\_condition([H1,H2|T],R):-

abs(H1-H2) =< 3,

!,

check\_condition([H2|T],R).

**PLF WRITTEN 8 ianuarie2020 ex 2**

**A close up of a paper

Description automatically generated**

f([],-1).

f([H|T],S):-

H > 0,

f(T,S1),

S1 < H,

!,

S is H.

f([\_|T],S):-

f(T,S1),

S is S1.

***SOLUTION:***

To avoid the recursive call in both clauses without redefining the predicate, we can use an auxiliary function. This way the recursion will be called only once in the main function, then we call the auxiliary function which handles the decision logic. If H is larger than 0 and more than S1 , the result will be updated to H. Otherwise , it will return the previous answer.

f1([],-1).

f1([H|T],S):-

f1(T,S1),

f\_aux(H,S1,S).

f\_aux(H,S1,S):-

H > 0,

S1 < H,

!,

S is H.

f\_aux(\_,S1,S):-

S is S1.

EXPLANATION:

Now, we avoid repeated recursion. In the main function we call only once the recursion and then we call the auxiliary function to handle the decision logic, which simplifies the structure. The predicate is not redefined, we just use an auxiliary function.

A close up of a paper

Description automatically generated

f([],0).

f([H|T],S):-

f(T,S1),

S1 is S-H.

***SOLUTION***

-The result of the evaluation f([1,2,3,4,5,6,7,8],S) will be a logical error due to the way variables are used. In prolog ,variables are immutable , meaning that once assigned a value , we can’t give them another one. In the clause S1 is S-H , P there is a conflict because Prolog tries to compute S1 using S, but S has no value at that point. This leads to an error because it can’t compute S1 without knowing the value of S.

A close up of a paper

Description automatically generated

%get\_valid\_arrangements(L,K,P,R)

%L- input list used to generate arrangements

%K – number of elements each arrangement should have

%P – expected product of elements from the arrangement

%R – resulted list with all valid arrangements

%flow model(i,i,i,o)

get\_valid\_arrangements(L,K,P,R):-

findall(X,(generate\_arrangements(L,K,X),check\_product(X,P,1)),R).

%generate\_arrangements(L,K,R)

%L – input llist containg the elements we are generating the arrangements with

%K – the number of elements arrangements should have

%R – resulted arrangement

%mathematical model

%generate\_arrangements(l1l2…ln,k) = { [], k = 0

% l1 U generate\_arrangements(l2l3…ln) , k>0

% generate\_arrangements(l12l3..ln) , k>0

generate\_arrangements(\_,0,[]).

generate\_arrangements(L, K, [H|R]) :- %generate all arrangements

H > 0, %check if we have k elements already in the arrangement

K1 is K-1 , %substract 1 because we already added one element to the list

select(H,L,REST), % select one element to add to the arrnangement

generate\_arrangements(REST,K1,R). % recursively generate the rest of the arrangements

%check\_product(L,P,CP,R)

%L – arrangement to be checked if valid

%P – product expected

%CP – current product

%R – result (1 if valid , 0 otherwise)

%mathematical\_model

%check\_product(l1l2…ln,cp,p) = { 0 , n = 0 and cp != p

% 1 , n = 0 and cp = p

% check\_product (l2l3…ln , l1\* cp , p) , otherwise

check\_product([],P,P,1). %if the list is empty and product is valid return 1

check\_product([],CP,P,0):- % if the list is empty and product doesn t match return 0

CP =\= P,

!.

check\_product([H|T],CP,P,R):- % calulate the product with the remaining elements

CP1 is CP \* H, %multiply the current element to the product

check\_product(T,CP1,P,R). % recursively go over the rest of the elements from the list

**PLF WRITTEN 8 ianuarie2020 ex 3**

**A close up of a paper

Description automatically generated**

f([],0).

f([H|T],S):-

f(T,S1),

H < S1,

!,

S is H + S1.

f([\_|T],S):-

f(T,S1),

S s S1+2.

***SOLUTION***

The avoid the recursive call in both clauses without redefining the predicate we are going to use an auxiliary function. In this way we are separating the base case and recursion from the deciding logic. Instead of 2 branches, we are going to have only one, which makes therecursive call , then calls the auxiliary function which will compute the final result.

f1([],0).

f1([H|T],S):-

f1(T,S1) % recusing over the tail

f\_aux(H,S1,S).

f\_aux(H,S1,S):-

H < S1,

!,

S is H+S1.

f\_aux(\_,S1,S):-

S is S1+2.

***JUSTIFICATION:***

Now we avoid repeated recursion. In the main function we merged two branches in one, so we will call recursion only once then let the auxiliary function to compute the result, simplifing also the function structure.We managed to avoid the double recursion without redefining the predicate and only using an auxiliary function.

A close up of a paper

Description automatically generated

The result will be:

111

112

121

122

211

212

221

222

This result shows all the combinations because the !(cut operator) is placed at the beginning.This cut only prevents backtracking after findind a solution , but it does not stop Prolog from exploring all combinations of solutions.

A close up of a book

Description automatically generated

%generate\_all\_valid\_subsets(L - list,N,-nr elements ,R-resulted list)

%L-given list , having the elements we will generate subsets with

%N -minimum number of elements a sublist should have

%R – resulted list containing all the valid subsets

%flow model(i,i,o)

generate\_all\_valid\_subsets(L,N,R):-

findall(X,(

generate\_subsets(L,X),

check\_minimum\_elements(X,0,N,1),

check\_sum(X,0,1)),

R).

%generate\_subsets(L,R).

%L – input list with the elements froming subsets with

%R – resulted subset

%mathematical model

%generate\_subsets (l1l2…ln) = { [] , n = 0

% l1 U generate\_subsets(l2l3..ln) , n > 0

% generate\_subsets(l2l3..ln)

% }

generate\_subsets([],[]).

generate\_subsets([H|T],[H|R]):-

generate\_subsets(T,R).

generate\_subsets([\_|T],R):-

generate\_subsets(T,R).

%check\_minium\_elements(L-list,C-number elements,N-minimum nr of elements,R-result)

%L- list of elements to be checked

%C -number of elements in the list

%N - minimum number of elements the list should have

%R – result (1 if has at least N elements , 0 otherwise)

%flow model(I,I,I,o)

%mathematical model

%check\_minimum\_elements(l1l2..ln,c,n) = { 1 , n = 0 and c >=n

% 0 , n = 0 and c < n

% check\_minimum\_elements(l2l3…ln,c+1,n) otherwise

check\_minimum\_elements([],C,N,0):-

C < N,

!.

check\_minimum\_elements([],C,N,1):-

C >= N,

!.

check\_minimum\_elements([\_|T],C,N,R):-

C1 is C + 1,

check\_minimum\_elements(T,C1,N,R).

%check\_sum(L-list,S-sum of elements , R-result)

%L – list of elements to be verified

%S – sum of elements from the list

%R – result (1 if sum is divisible with 3 , 0 otherwise)

%flow model (i,i,o)

%mathematical model

%check\_sum(l1l2l3…ln,s) = { 0 , n = 0 and s mod 3 != 0

% 1 , n = 0 and s mod 3 = 0

% check\_sum(l2l3..ln , c+l1) otherwise

check\_sum([],C,0):-

C mod 3 =\= 0,

!.

check\_sum([],C,1):-

C mod 3 =:= 0,

!.

check\_sum([H|T],C,R):-

C1 is C + H,

check\_sum(T,C1,R).

**PLF WRITTEN 8 ianuarie2020 ex 4**

A close up of a paper

Description automatically generated

f(1,1).

f(K,X):-

K1 is K-1,

f(K1,Y),

Y > 1,

!,

K2 is K1-1,

X is K2.

f(K,X):-

K1 is K-1,

f(K1,Y),

Y > 0.5,

!,

X is Y.

f(K,X):-

K1 is K-1,

f(K1,Y),

X is Y-1.

***SOLUTION***

For avoiding the recursive call in al clauses we can use an auxiliary function .In this way we can merge all three branches in one and call the recursive call only once. After getting the result from the recursion , we call the auxiliary function to do the decision logic.

f1(1,1).

f1(K,X):-

K1 is K-1,

f(K1 , Y),

f\_aux(K1,Y,X):-

Y > 1,

!,

K2 is K1 -1,

X is K2.

f\_aux(K1,Y,X):-

Y > 0.5,

!,

X is Y.

f\_aux(\_,Y,X):-

X is Y-1.

This method avoid recursing in all branches while sustaining the same predicate logic and simplifying the code.

A close up of a paper

Description automatically generated

p(1).

p(2).

q(1).

q(2).

r(1).

r(2).

s:-p(X), !, q(Y), r(Z) , write(X,Y,Z), nl.

The result of the goal s is:

111

112

121

122

The reason of that result is the way the ! is placed. The cut operator(!) is placed after p(X) .This stops Prolog from backtracking to find other solutions for p(X).Once a value is found p(X) is fixed .However the cut doesn’t affect the exploration of solution q(Y) and r(Z).Prolog keeps looking recursively for all the combinations for q(Y) and r(Z) and this results in the 4 solutions I displayed earlier.

A text on a piece of paper

Description automatically generated

%produs – the produce of all the elements from a list using a collector variable

%produs(L – list , C -collector variable , P-produs rezultat)

%L – input list

%C – collector variable which will contain the product of elements

%P – the final product after traversing the entire list

%produs(i,i,o)

%mathematical model

%produs(l1l2…ln , c) = { c , n = 0

% produs(l2l3…ln , l1\*c) , otherwise

% }

produs([],C,C). % if the list is null , we return the collector variable containg the product

produs([H|T],C,P):-

C1 is H \* C, %multiply the current element to the collector variable

produs(T,C1,P). % recursively go through the list of the list

%a insert – inserting an element into a list

%insert(L – list , E -element , R – resulted list)

%L – the list in which we are inserting the element

%E – element to be inserted

%R – resulted list

%insert(i,I,o)

&mathematical model

%insert(l1l2…ln , e) = { e U l1l2…ln

% l1 U insert(e,l2l3…ln)

% }

insert(L,E,[E|L]). %adding the element at the beginning of the list

insert([H|T],E,[H|R]):- %adding the head

insert(T,E,R). %recursively go through all the list

%arrangements of k elements of a linear list

%arr(L – list , K – number of elements ,R-resulted list)

%L – input list containing the elements we are creating the arrangements with

%K – number of elements the arrangements should contain

%R – resulted permutation

%flow model - arr(i,i,o)

%mathematical model

%arr(l1l2…ln, k) = { l1 , k = 1

% arr(l2…ln , k) , k>=1

% insert (l1 , arr(l2l3…ln,k-1)) otherwise

arr([H|\_],1,[H]).

arr([\_|T], K,R):-

arr(T,K,R).

arr([H|T],K,R):-

K > 1,

K1 is K-1,

arr(T,K,R1),

insert(H,R1,R).

%onesol – a function to help us determine the solutions for our problem

%onesol(L-list , K – nr elements , V – maximum expected product , R- resulted list)

%flow model – onesol(I,I,I,o)

onesol(L,K,V,R):-

arr(L,K,X).

produs(X,1,P),

P =< V.

%allsols – a function used for determining all the solutions and add them into a result list with the utilization of the

%predefined predicate findall

%allsols(L-list , K – number elements , V – maximum product , R – resulted list)

%flow model(I,I,I,o)

allsols(L,K,V,R):-

findall(X,onesol(L,K,V,X),R).