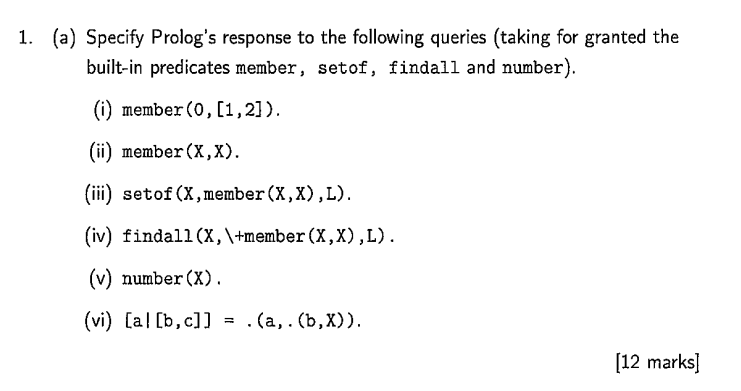
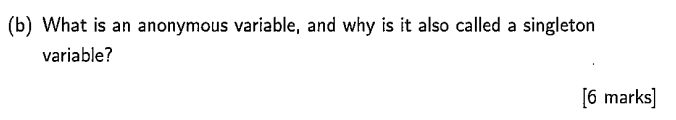
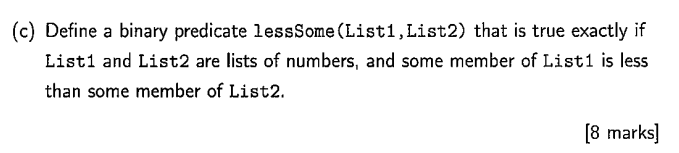
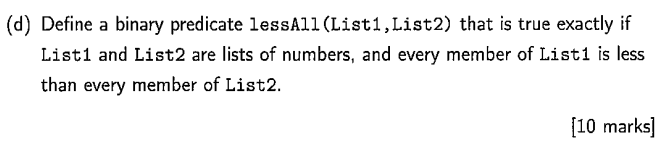
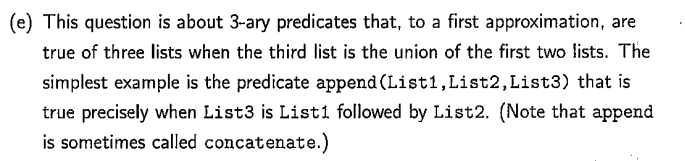
## 2017

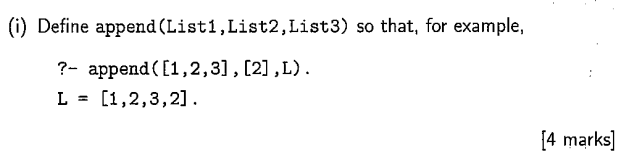
  
%1a  
% False  
% does not terminate  
% does not terminate, goes out of stack  
% L = []  
% false  
% X = [c]  


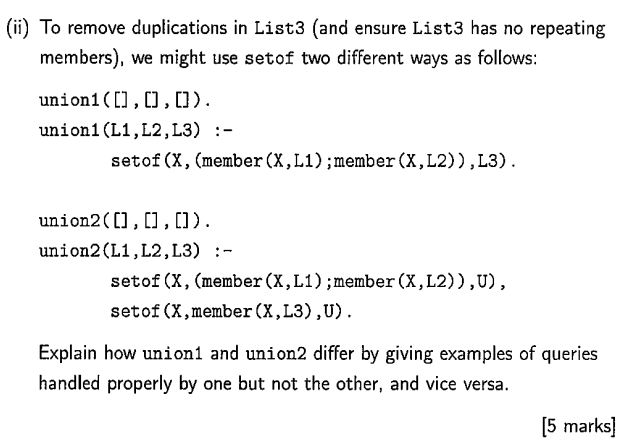
%b An anonymous variable is one that every occurrence of **\_** denotes a distinct variable. It is called a singleton variable as it only ever appears once in a clause



%c  
lessSome(List1, List2):- member(X, List1), member(Y, List2),  
 number(X), number(Y), X<Y, !.

  
%d  
lessAll([], \_ ).  
lessAll([H|T], List2):- lessThanAll(H, List2), lessAll(T, List2).  
lessThanAll(\_, []).  
lessThanAll(X, [H|T]):- X<H, lessThanAll(X,T).  


  
%e (i)  
append([], List, List).  
append([H|T], List2, [H|List3]):- append(T, List2, List3).



%e (ii)

Union 1 will run with L3 uninstantiated

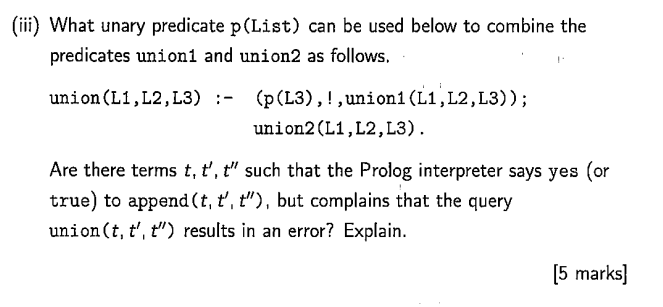
Union 2 needs L3 to be instantiated

union2([1,2,3],[4,5,6],L). Will never terminate.

Union 1 requires an ordered L3, if instantiated

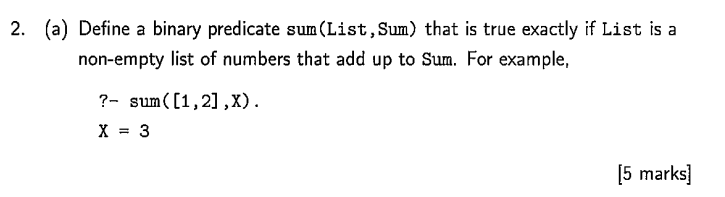
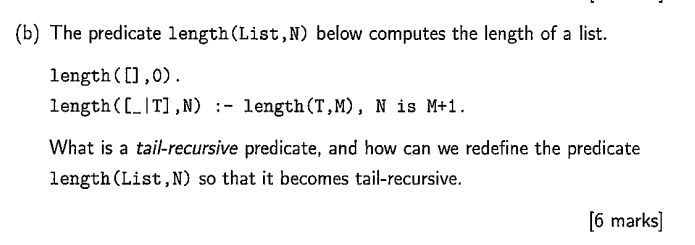
Union 2 can handle an unordered L3

union1([1,2,3],[4,5,6],[1,2,4,3,6,5]) Will fail.

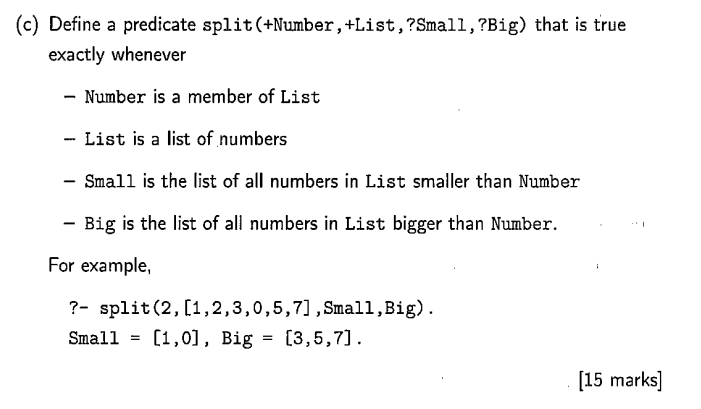


%e (iii)a|  
 p(List):- var(List)

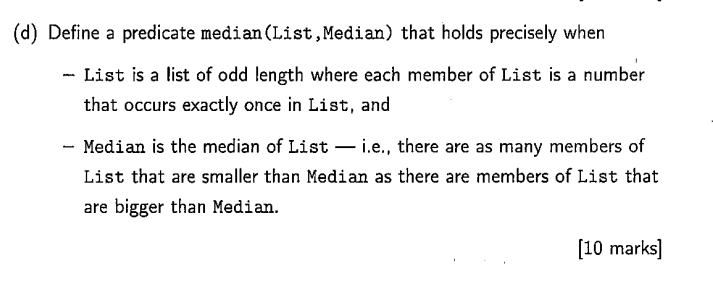
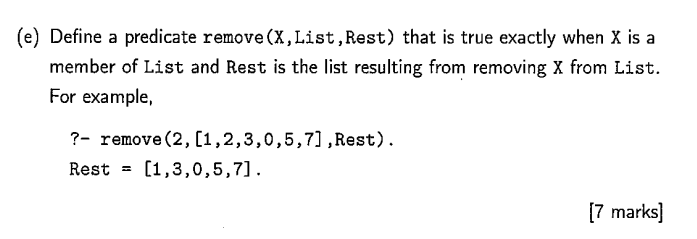
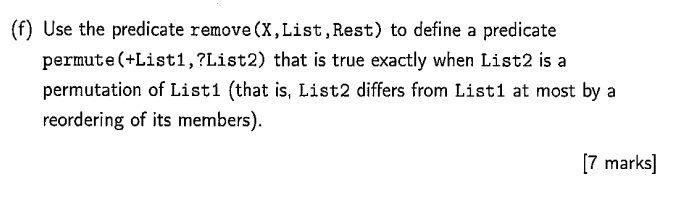
% append(T, TT, [1,2,3,4,5]). Will produce all possible lists T, TT that result in T’’. union(T,TT, [1,2,3,4,5]). Will throw an error because L1, L2 need to be instantiated

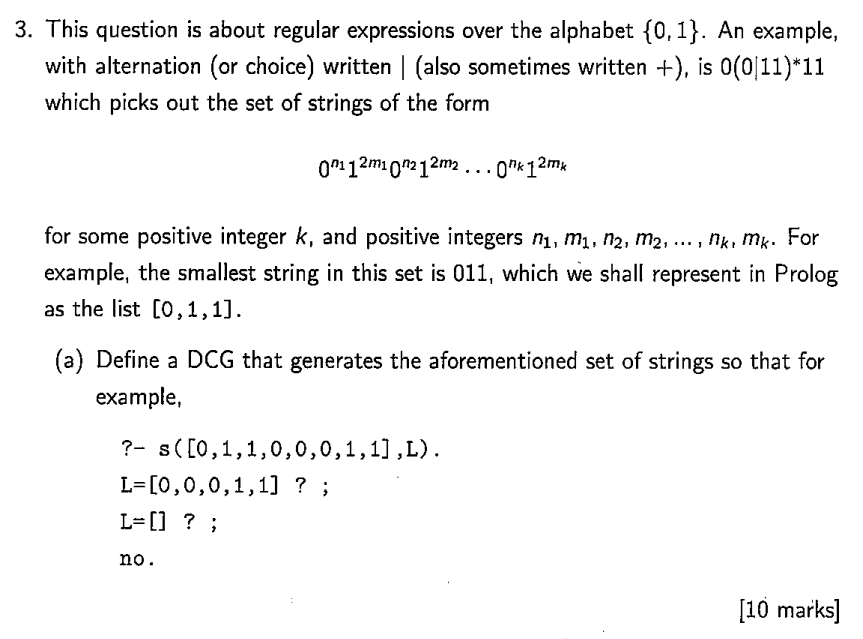
  
%2a  
sum([X], X).  
sum([X,Y|List], Sum):- sum([Y|List], PR), Sum is PR+X.  
  
%b

A tail recursive predicate is one whose final recursive call is to itself. A function call is said to be tail recursive if there is nothing to do after the function returns except return its value.

length1(List, N):- tailLength(List, 0, N).  
  
tailLength([], Acc, Acc).  
tailLength([\_|T], Acc, N):- NewAcc is Acc+1, tailLength(T, NewAcc, N).  
%c  
split(Number, List, Small, Big):- member(Number, List), allNumeral(List), spl(Number, List, Small, Big).

allNumeral([]).  
allNumeral([H|T]):- number(H), allNumeral(T).  
  
spl(\_, [], [], []).  
spl(Number, [H|T], [H|Small], Big):- Number>H, spl(Number, T, Small, Big).  
spl(Number, [H|T], Small, Big):- Number =:= H, spl(Number, T, Small, Big).  
spl(Number, [H|T], Small, [H|Big]):- Number<H, spl(Number, T, Small, Big).

  
%d  
  
median(List, Median) :- setof(X, member(X, List), List),  
 split(Median, List, Small, Big), length1(Big, N), length1(Small, N).  
  
%e  
remove(X, List, Rest) :- member(X, List), rm(X, List, Rest).  
  
rm(\_, [], []).  
rm(X, [X|T], Rest):- rm(X, T, Rest).  
rm(X, [H|T], [H|Rest]):- X\=H, rm(X, T, Rest).  
  
%f  
permute([],[]).  
permute([H|T], List2):- remove(H, List2, NewList2), permute(T, NewList2).

%Question 3  


s3 --> a(N), b(M), s33, {M>0, N>0}.

s33 --> [].

s33 --> a(N), b(M), {M>0, N>0}.

a(N) --> [0], a(NN), {N is NN+1}.

a(0) --> [].

b(N) --> [1,1], b(NN), {N is NN+1}.

b(0) --> [].

s4(A,Z):- aa(N,A,B), bb(M, B,C), s44(C, Z), M>0, N>0.

s44(X, X).

s44(A, Z):- aa(N,A,B), bb(M, B,Z), M>0, N>0.

aa(N, [0|X], Z):- aa(M, X, Z), N is M+1.

aa(0, X,X).

bb(N, [1,1|X], Z):- bb(M, X, Z), N is M+1.

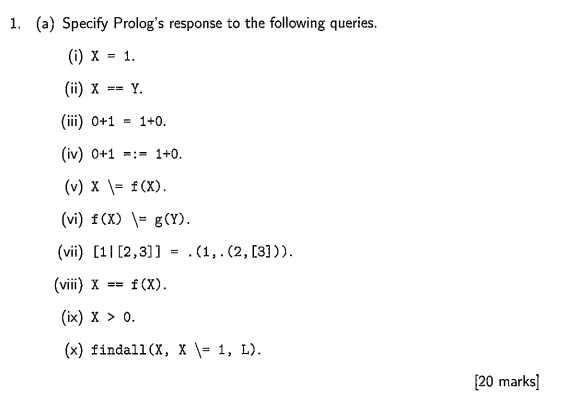
bb(0, X,X).

## 

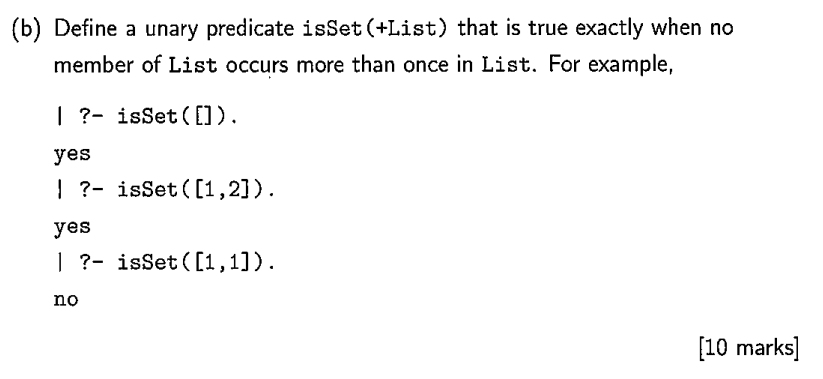
## 

## 

## 2016

7

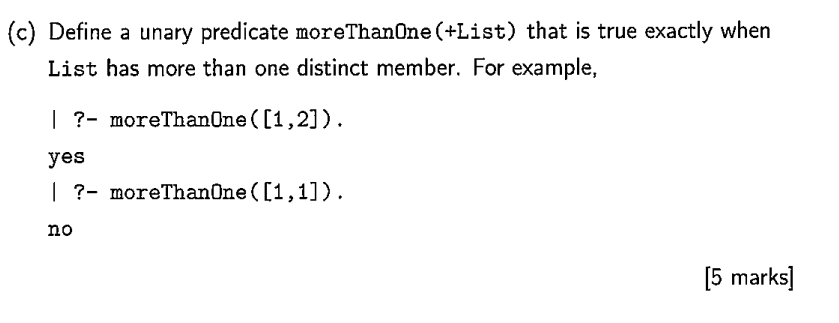
% a  
% i) X = 1  
% ii) False  
% iii) False  
% iv) true  
% v) False  
% vi) true  
% vii) dictionary error, though should be true on older versions  
% viii) false  
% ix) non instantiated for X  
% x) L = [], for some reason  
 (L is an uninstantiated list, findall will find instances where x does not equal 1 in L. There are none. There for List is empty.

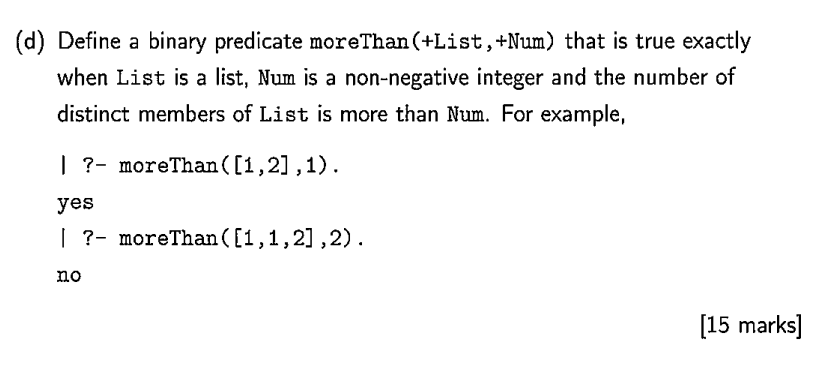
  
%

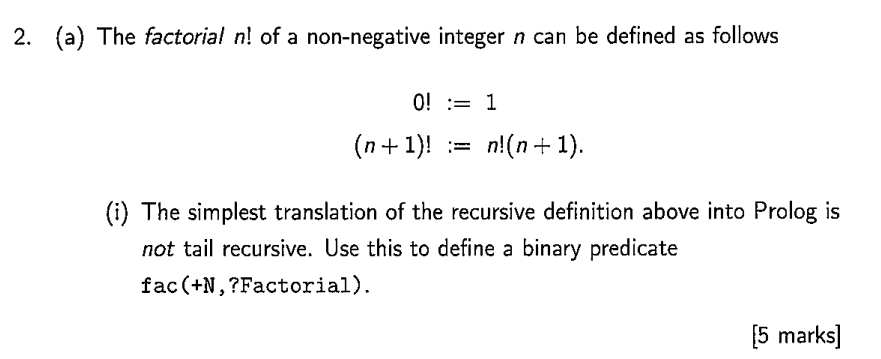
isSet1([]).

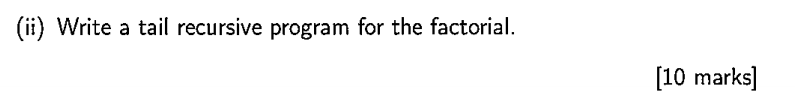
isSet1([H|T]):-

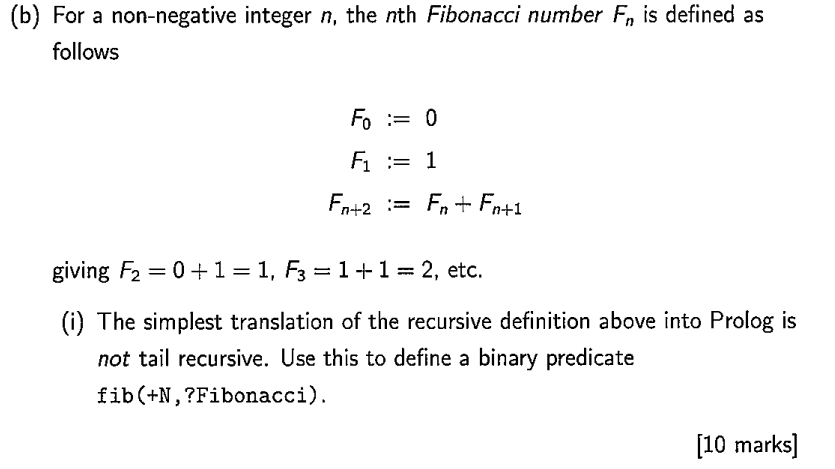
isSet2([]).  
isSet2([H|T]):- nonMember(H, T), isSet2(T).  
  
nonMember2(\_, []).  
nonMember2(X, [H|T]):- X\=H, nonMember2(X, T).  
  
isSetAlt(List) :- setof(X, member(X, List), List).

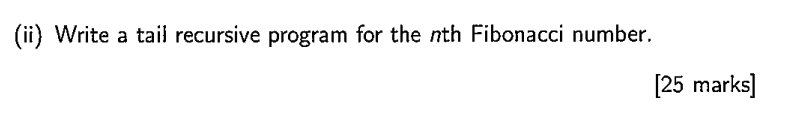
  
%c  
moreThanOne(List):- setof(X, member(X, List), SetList), length(SetList, N), N>1.

setof orders and removes repetitions from List. Therefore, if the length of setof is above one, that means that there is more than 1 number after eliminating repetitions. Meaning there’s at least 2 distinct numbers. %d  
moreThan(List, Number):- setof(X, member(X, List), SetList), length(SetList, N), N>Number.  
  
%2  
%a (i)



fac(0, 1).  
fac(N, Factorial):- N>0, LowerN is N-1, fac(LowerN, LowerFactorial), Factorial is LowerFactorial \* N.  
  
% ii  
  
fac2(N, Factorial):- tailFac(N, 1, Factorial).  
  
tailFac(0, Fac, Fac).  
tailFac(N, Acc, Fac):- N>0, LowerN is N-1, NewAcc is Acc\*N, tailFac(LowerN, NewAcc, Fac).  
  
%b i

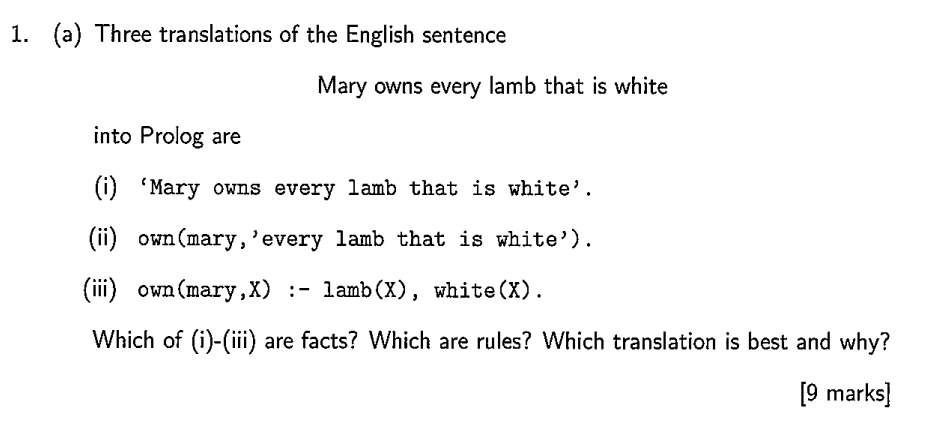
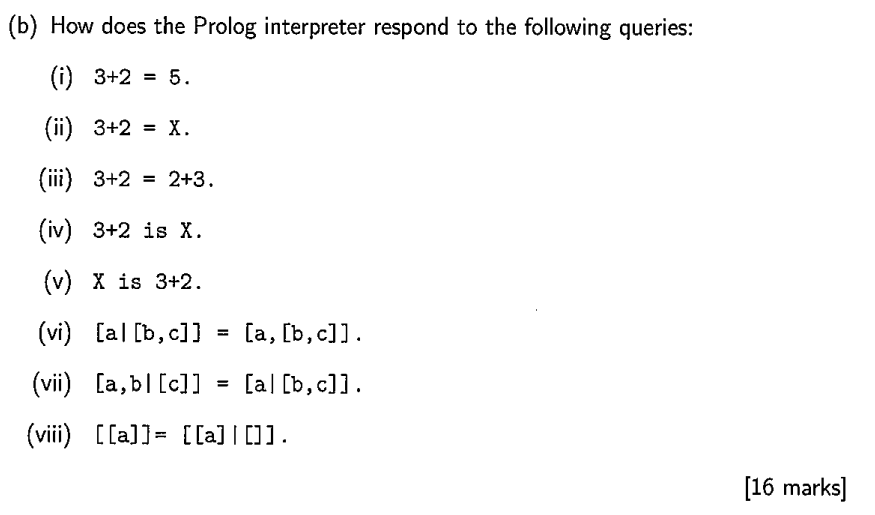
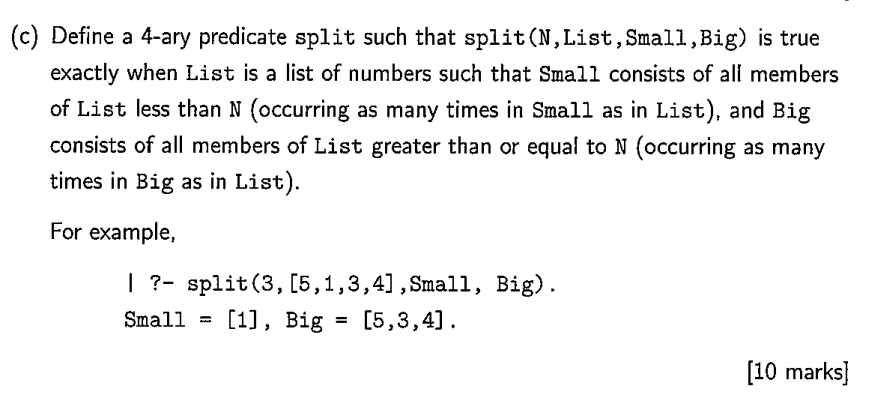
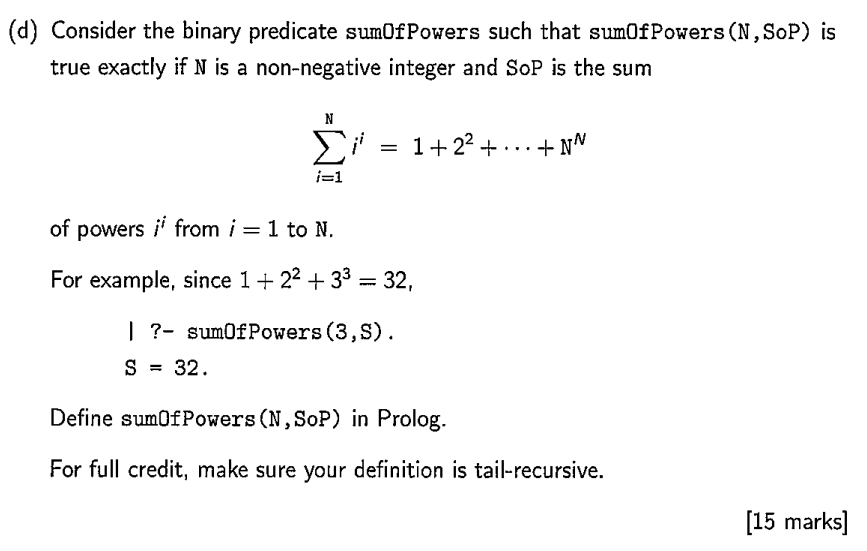
  
fib(0, 0).  
fib(1, 1).  
fib(N, Fibonnacci):- N>1,  
 LowerN is N-1, LowerN2 is N-2,  
 fib(LowerN, PR1), fib(LowerN2, PR2),  
 Fibonnacci is PR1+ PR2.  
  
% ii

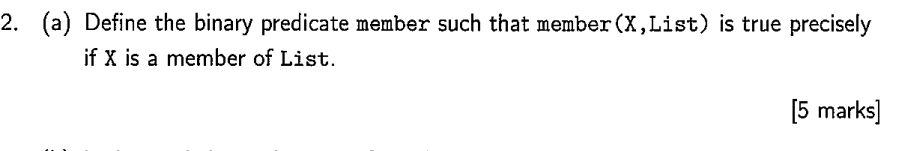
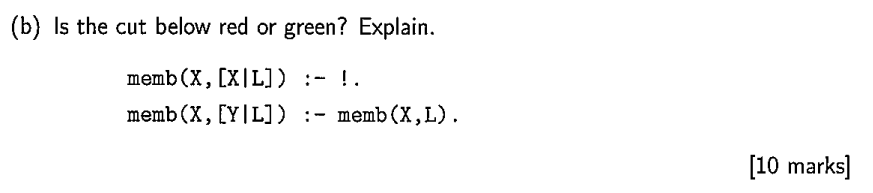
  
tailFib(N, Fibonnacci):- fibAcc(N, 0, 1, Fibonnacci).

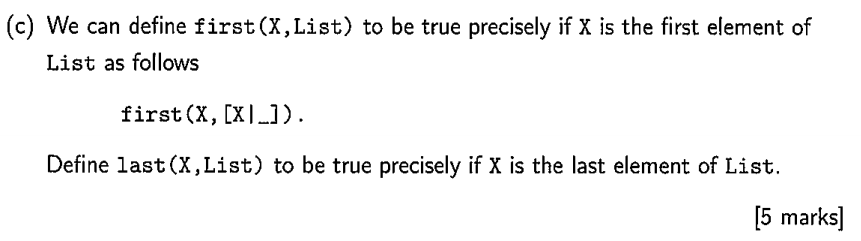
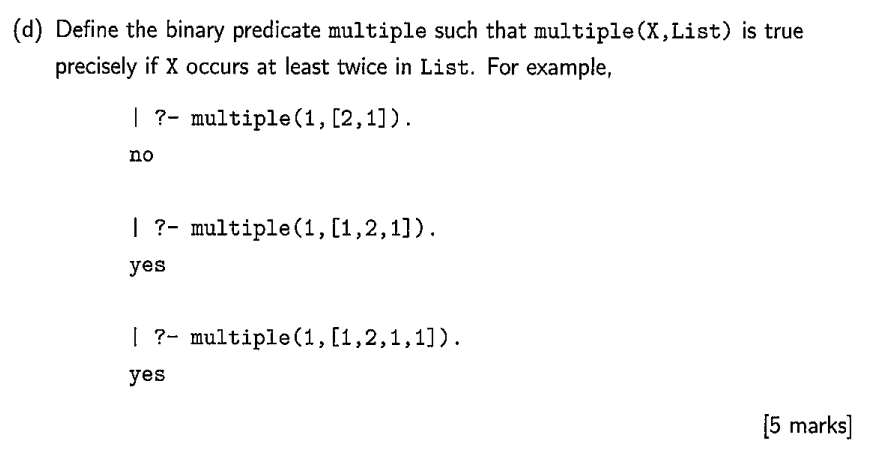
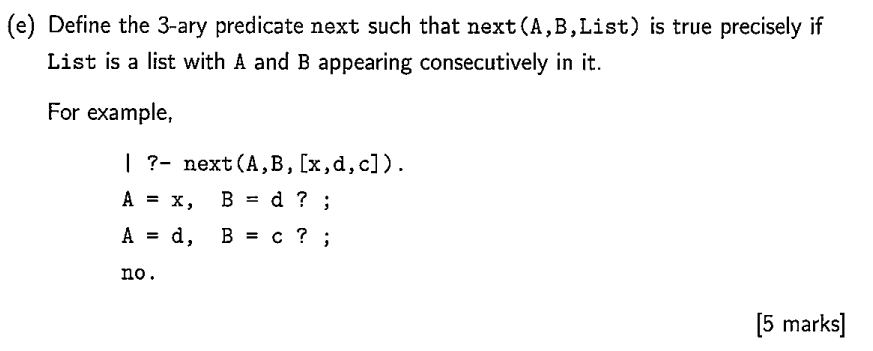
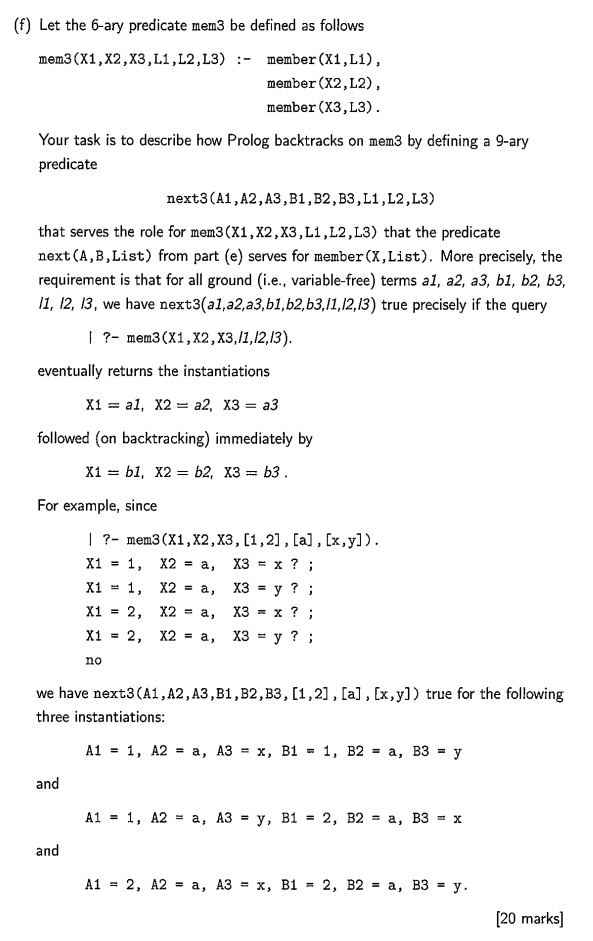
fibAcc(0, Fib, \_ik, Fib).  
fibAcc(N, A, B, Fib):- N>0, NewB is B+A, LowerN is N-1,  
 fibAcc(LowerN, B, NewB, Fib).

## 2015

%1a[

  
% first two are facts, third is a rule, third is best as will verify an item is a lamb and is white  
  
%b  
% False  
% X = 3+2.  
% False  
% args not instantiated  
% X = 5.  
% false  
% true.  
% true  
  
%c  
% see 2 c 2017  
%d  
sumOfPowers(N,SoP):- sOP(N, 0, SoP).  
  
sOP(0, SoP, SoP).  
sOP(N, Acc, SoP):- N>0, NewAcc is Acc + N\*\*N, LowerN is N-1, sOP(LowerN, NewAcc, SoP).  
  
%2

  
%a  
member1(X, [X|\_]):- !.  
member1(X, [H|T]):- H\=X, member1(X, T).  
%b  
% cut is green. It has no impact on the results of the code and is only used for efficiency - To stop searching once we’ve found member.

  
%c  
last(X, [X]) :- !.  
last(X, [\_|T]) :- last(X, T).  
  
%d  
multiple(X, List):- findall(C, (member(C, List), X=C), XList), length(XList, N), N>1.  
%e  
next(A, B, [A, B|\_]).  
next(A, B, [\_|T]):- next(A,B,T).%f

% David's good attempt - "Values" is like a stack of possible outcomes.

% getnext(+DefaultValues, +Values, ?NextValues)

% Get the next set of possible outcomes.

% e.g. [[x,y],[a],[1,2]] -> [[y],[a],[1,2]] or [[y],[a],[1,2]] -> [[x,y],[a],[2]]

getnext(\_, [[\_,H2|T]|Values], [[H2|T]|Values]).

getnext([DH|DT], [[\_]|Values], [DH|NextValues]) :-

getnext(DT, Values, NextValues).

next3(A1,A2,A3,B1,B2,B3,L1,L2,L3) :-

helper(A1,A2,A3,B1,B2,B3,[L3,L2,L1],[L3,L2,L1]).

helper(A1,A2,A3,B1,B2,B3,Values,DefaultValues) :-

Values = [[A3|\_],[A2|\_],[A1|\_]],

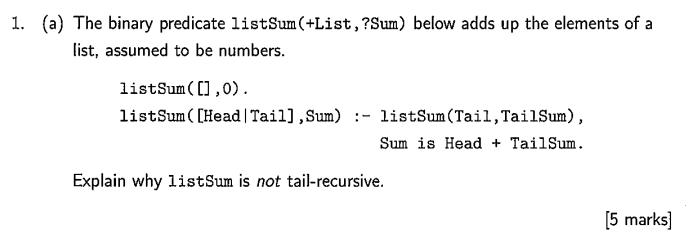
getnext(DefaultValues, Values, [[B3|\_],[B2|\_],[B1|\_]]).

helper(A1,A2,A3,B1,B2,B3,Values,DefaultValues) :-

getnext(DefaultValues, Values, NextValues),

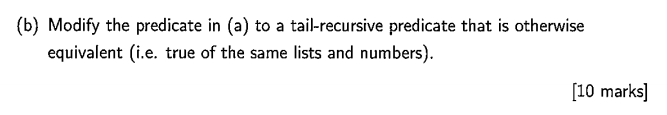
helper(A1,A2,A3,B1,B2,B3,NextValues,DefaultValues).

## 2014



% 1 (a)

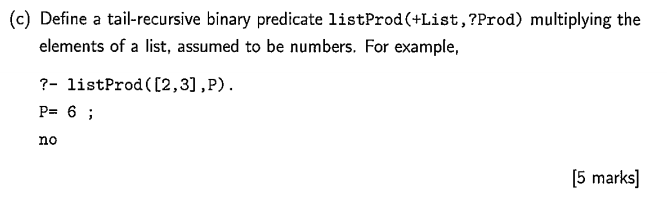
% list sum is not tail recursive because it does not return what its recursive call to itself returns. Also a function is said to be **tail recursive** when the recursive call is the last function invoked in the evaluation of the body of the function - which is not the case here



listSum(List, Sum):-listSum2(List,0,Sum).

listSum2([],Sum,Sum).

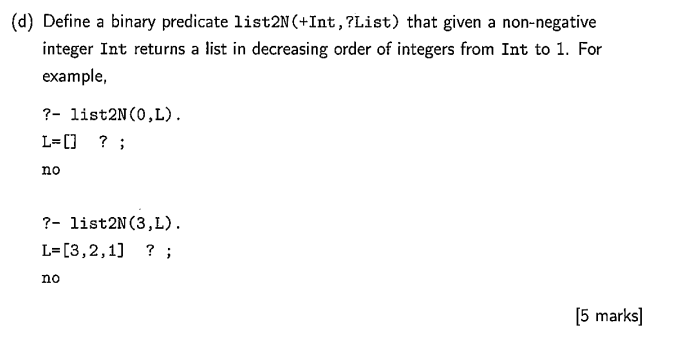
listSum2([H|T],X,Sum):-NewSum is H+X, listSum2(T,NewSum,Sum).



listProd(List,Prod) :- listProd2(List,1,Prod).

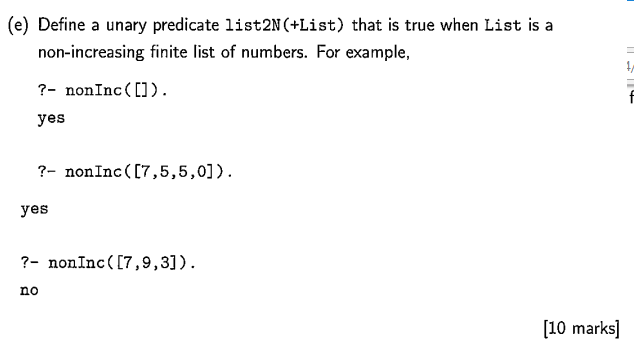
listProd2([], Prod, Prod).

listProd2([H|T], X, Prod):- NewProd is H\*X, listProd2(T, NewProd, Prod).



list2N(0, []).

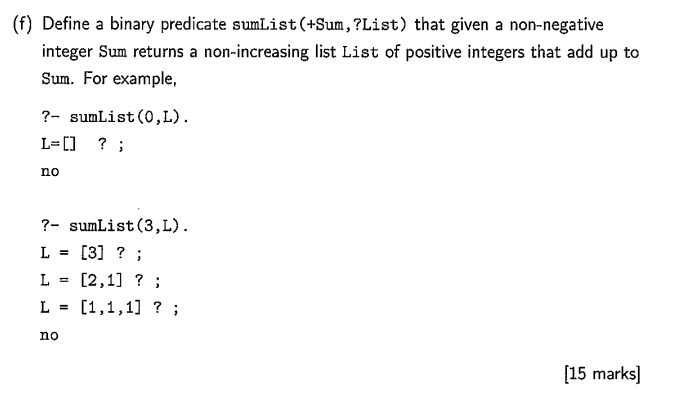
list2N(N, [N|T]):- integer(N), N>=0, NewN is N-1, list2N(NewN, T).7

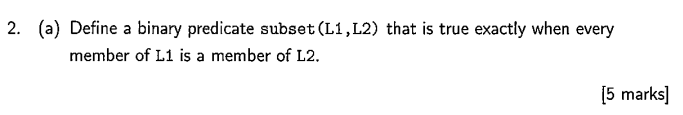


nonInc1([]).

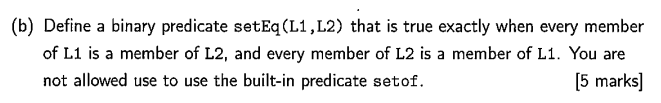
nonInc1([\_]).

nonInc1([A,B|T]) :- A >= B, nonInc1([B|T]).





%2a)  
subset([],\_).  
subset([H|T],L):- member(H,L), subset(T,L).



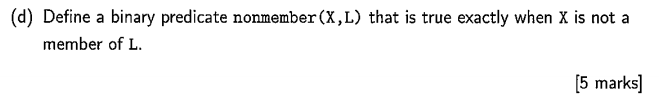
%2b)

setEq(L1,L2):- subset(L1,L2), subset(L2,L1).



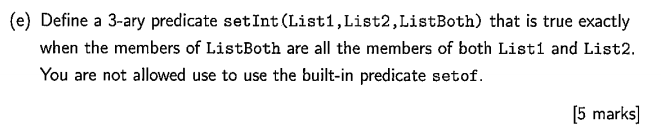
%2c)

setEq2(L1,L2):- setof(X,member(X,L1),Z), setof(Y,member(Y,L2),Z).



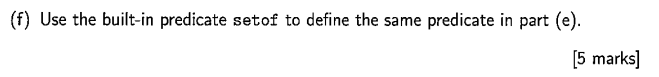
%2d)

nonmember(X,L):- \+ member(X,L).



%2e)

setInt(L1,L2,L3):- append(L1,L2,LU), setEq(LU,L3).

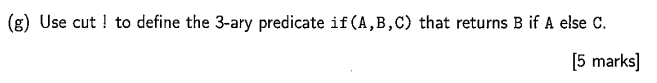


%2f)

setInt2(L1,L2,L3):- setof(X,member(X,L1),A1), setof(Y,member(Y,L2),A2),

append(A1,A2,L3).

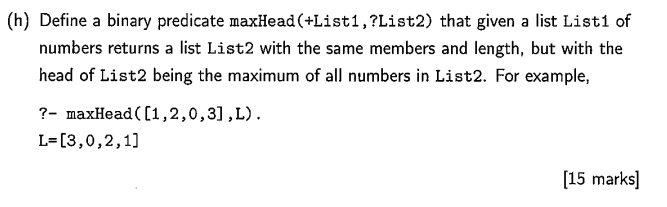
memberOfBoth(X,L1,L2):- member(X,L1), member(X,L2).



%2g)

%(Tim Fernando verified !)

if(A,B,C):- A, !, B; C.



maxHead(L1,L2):- maxHead([],L1,L2).

maxHead(L1,[H|T],L3):- isMax(H,T), append([H|L1],T,L3).

maxHead(L1,[H|T],L3):- append(L1,[H],X), maxHead(X,T,L3).

isMax(\_,[]).

isMax(X,[H|T]):- X>=H, isMax(X,T).