BACKTRACKING PLF

% compute subsets

% n - nr of elements => total nr of subsets: 2^n

% subsets(l1,l2,...,ln) = - l1 reun subsets(l2,...,ln), n>0

% - subsets(l2,...,ln), n>0

% compute permutations

% n - nr of elements => total nr of permutations: n!

% permutations(l1,l2,...,ln) = - insert(l1,permutations(l2,...,ln), n>0

% - empty list if n==0

% insert element e in all possible locations in L

% insert(e, l1,...,ln) = - e reun l1,l2,...,ln, n>=0

% - l1 reun insert(e,l2,...,ln), n>0

%

% compute combinations

% combinations(l1,l2,...,ln,k) = - l1, if k==1

% - combinations(l2,...,ln,k-1), if k>=1

% - l1 reun combinations(l2,...,ln,k-1), if k>1

%

% compute arrangement

% arr(l1,l2,...,ln,k) = - l1, if k==1

% - arr(l2,...,ln,k), if k>=1

% - insert(l1, arr(l2,...,ln,k-1)), if k>1

%

% onebyonesol(L,R)

% findall( R, onebyonesol(L,R), All)

% subsets(L,R) (i,o)

subset([],[]).

subset([H|T],[H|R]):-

subset(T,R).

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

subset(T,R).

main\_subset(L,X):-

findall(R, subset(L,R), X).

% insert(E-element, L-list, R-res) (i,i,o)

insert(E,L,[E|L]).

insert(E,[H|T],[H|R]):-

insert(E,T,R).

% perm(L-list, R-res) (i,o)

perm([],[]).

perm([H|T],R):-

perm(T,TR),

insert(H,TR,R).

main\_perm(L,X):-

findall(R, perm(L,R),X).

% comb(L-list, K-nr, R-res) (i,i,o)

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

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

K>=1,

comb(T,K,R).

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

K>1,

NewK is K-1,

comb(T,NewK,R).

main\_comb(L,E,X):-

findall(R,comb(L,E,R),X).

% arr(L-list, K-nr, R-res) (i,i,o)

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

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

K>=1,

arr(T,K,R).

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

K>1,

NewK is K-1,

arr(T,NewK,A),

insert(H,A,R).

main\_arr(L,K,X):-

findall(R,arr(L,K,R),X).

VALLEY SEQUENCE

subset([],[]).

subset([H|T],[H|R]):-

subset(T,R).

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

subset(T,R).

main\_subset(L,X):-

findall(R, subset(L,R), X).

% perm(L-list, R-res) (i,o)

perm([],[]).

perm([H|T],R):-

perm(T,TR),

insert(H,TR,R).

main\_perm(L,X):-

findall(R, perm(L,R),X).

% comb(L-list, K-nr, R-res) (i,i,o)

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

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

K>=1,

comb(T,K,R).

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

K>1,

NewK is K-1,

comb(T,NewK,R).

main\_comb(L,E,X):-

findall(R,comb(L,E,R),X).

% insert(E-element, L-list, R-res) (i,i,o)

insert(E,L,[E|L]).

insert(E,[H|T],[H|R]):-

insert(E,T,R).

% arr(L-list, K-nr, R-res) (i,i,o)

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

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

K>=1,

arr(T,K,R).

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

K>1,

NewK is K-1,

arr(T,NewK,A),

insert(H,A,R).

main\_arr(L,K,X):-

findall(R,arr(L,K,R),X).

% find all arr of len k with sum S and product P

suma([],0).

suma([H|T],R):-

suma(T,RT),

R is RT+H.

prod([],1).

prod([H|T],R):-

prod(T,RT),

R is RT\*H.

onesol(L,K,S,P,A):-

arr(L,K,A),

suma(A,S),

prod(A,P).

main\_sol(L,K,S,P,A):-

findall(R, onesol(L,K,S,P,R),A).

% display all the subsequences of a sequence which have a valley aspect

% check if valley using falg F=1 is decreasing, F=0 if increasing

% valley(l1,l2,...,ln,F) = - true, if n=1 and F=0

% - valley(l2,...,ln,1), if n>1, F=1, l1>l2

% - valley(l2,...,ln,0), if n>1, F=1 or F=0, l1<l2

% - false, otherwise

% valley\_main(l1,l2,...,ln) = - false, if l1<l2

% - valley(l1,l2,...,ln,1), otherwise

% valley\_sol(l1,l2,...,ln) = a1,...,ak, if perm(subs(l1,l2,...,ln))=a1,...,ak and

% valley\_main(a1,...,ak)

% all\_sol(l1,l2,...,ln) = findall(valley\_sol(l1,l2,...,ln))

% valley(L-list, F-flag) (i,i)

valley([],0).

valley([H1,H2|T],1):-

H1>H2,

valley([H2|T],1).

valley([H1,H2|T],\_):-

H1<H2,

valley([H2|T],0).

valley\_main([H1,H2|T]):-

H1>H2,

valley([H1,H2|T],1).

% valley\_sol(L-list, R-res) (i,o)

valley\_sol(L,R):-

subset(L,S),

perm(S,P),

valley\_main(P),

R is P.

all\_sol(L,X):-

findall(R,valley\_sol(L,R),X).

L1-LISP

;1.

;a) Write a function to return the n-th element of a list, or NIL if such an element does not exist.

(defun return-n (l n)

(cond

((null l) NIL)

((= n 1) (car l))

(t (return-n (cdr l) (- n 1)))

)

)

(print (return-n '(1 2 3 4 5 6) 5))

;b) Write a function to check whether an atom E is a member of a list which is not necessarily linear.

(defun check-e (l e)

(cond

((null l) NIL)

((and (atom (car l)) (eq e (car l))) t)

((atom (car l)) (check-e (cdr l) e))

(t (or (check-e (car l) e) (check-e (cdr l) e)))

)

)

(print (check-e '(1 7 (5 8 (4)) 9 10) 4))

;c) Write a function to determine the list of all sublists of a given list, on any level.

;A sublist is either the list itself, or any element that is a list, at any level.

;Example: (1 2 (3 (4 5) (6 7)) 8 (9 10)) => 5 sublists :

;( (1 2 (3 (4 5) (6 7)) 8 (9 10)) (3 (4 5) (6 7)) (4 5) (6 7) (9 10) )

(defun display (l)

(print l)

)

(defun determine (l)

(cond

((null l) nil)

((listp (car l))

(display (car l)) ; Display the sublist

(determine (car l)) ; Recursively process the sublist

(determine (cdr l))) ; Continue with the rest of the list

(t (determine (cdr l)))

)

)

;; Example call

(determine '(1 2 (3 (4 5) (6 7)) 8 (9 10)))

;d) Write a function to transform a linear list into a set.

(defun nr-occ (e n l)

(cond

((null l) n)

((eq e (car l)) (nr-occ e (+ n 1) (cdr l)))

(t (nr-occ e n (cdr l)))

)

)

(print (nr-occ 1 0 '(4 5 1 2 1)))

(defun turn-set (l l1)

(cond

((null l) l1)

((> (nr-occ (car l) 0 l1) 0) (turn-set (cdr l) l1))

(t (turn-set (cdr l) (cons (car l) l1)))

)

)

(print (turn-set '(1 7 9 0 1 2 6 7 1 1 2) NIL))

; Write a function to check if an atom is member of a list (the list is non-liniar)

(defun check-el (e l)

(cond

((and (atom l) (equal l e)) t)

((atom l) nil)

(t (some #'identity (mapcar (lambda(a) (check-el e a)) l)))

)

)

;(print (check-el 7 '(3 2 (5 6 (1 2)))))

;Write a function that returns the sum of numeric atoms in a list, at any level.

(defun summ(l)

(cond

((numberp l) l)

((atom l) 0)

(t (apply #'+ (mapcar #'summ l)))

)

)

;(print (summ '(1 2 3 (4 5 (6 7) (0 9 a)))))

;Define a function to tests the membership of a node in a n-tree represented as (root list\_of\_nodes\_subtree1 ... list\_of\_nodes\_subtreen) Eg. tree is (a (b (c)) (d) (E (f))) and the node is "b" => true

(defun check-tr(e tree)

(cond

((and (atom tree) (equal tree e)) t)

((atom tree) nil)

(t (some #'identity (mapcar (lambda(a) (check-tr e a)) tree)))

)

)

;(print (check-tr 'g '(a (b (c)) (d) (e (f)))))

;Write a function that returns the product of numeric atoms in a list, at any level.

(defun prod(l)

(cond

((numberp l) l)

((atom l) 1)

(t (apply #'\* (mapcar #'prod l)))

)

)

;(print (prod '(1 4 (a 2 (8)))))

;Write a function that computes the sum of even numbers and the decrease the sum of odd numbers, at any level of a list.

(defun summ1(l)

(cond

((and (numberp l) (equal (mod l 2) 0)) l)

((and (numberp l) (equal (mod l 2) 1)) (- 0 l))

(t (apply #'+ (mapcar #'summ1 l)))

)

)

;(print (summ1 '(1 2 3 4 (8 (8) 9))))

;Write a function that returns the maximum of numeric atoms in a list, at any level.

(defun maxxx(l)

(cond

((numberp l) l)

((atom l) -100000)

(t (apply #'max (mapcar #'maxxx l)))

)

)

;(print (maxxx '(1 52 3 4 (8 (8 a) 12))))

;Write a function that substitutes an element E with all elements of a list L1 at all levels of a given list L.

(defun repl(e l1 l)

(cond

((and (atom l) (equal l e)) l1)

((atom l) l)

(t (mapcar (lambda(a) (repl e l1 a)) l))

)

)

(print (repl 1 '(10 10) '(2 1 (3 2 (1 3)))))