

3. Soluție pentru evitarea apelului recursiv - LISP

(R11)

(defun F (e))

(cond

((null e) 0)

(+ ((economă x)

(cond

((> x 2) (+ (car e) (F (cdr e)))) )

(+ x))

)

(F (car e)))

)

)

)

2. Lista permutăriilor având proprietatea că val. absolută a diff. dintre 2 val. consecutive din permutare este  $c = 3$ .

Ideea algoritmului:

1. predicat candidat  $\rightarrow$  va genera un element din lista dată
2. predicat permute  $\rightarrow$  construiește o permutare validă, prin intermediul unei liste auxiliare și a refurmării altoru scind dimensiunea ei este egală cu dimensiunea listei inițiale
3. predicat perm  $\rightarrow$  utilizat pentru inițializarea parametrilor auxiliari din predicatul "permute" + predicat auxiliar pentru det. cum sănii unei liste
4. predicat main  $\rightarrow$  refurmărea liste tuturor permutărilor valide construite

Modelle matematice.

candidat ( $e_1 \dots e_m$ ) =

1.  $e_1 \in m > L$

2. Candidat ( $e_2 \dots e_m$ ),  $m > 1$

permutare ( $e, m, \text{lg}, \text{coe}$ ) =  $\begin{cases} \text{coe}, m = \text{lg} \\ \text{permutare } (e, m, \text{lg}+1, \text{e} \oplus \text{coe}), \\ \text{lg} - \text{coe}_1 \leq 3 (\text{coe} = \text{coe}_1 \dots \text{coe}_m), \\ e \text{ nu se regaseste in coe}, \text{ unde} \\ e = \text{candidat}(e) \end{cases}$

lungime ( $e_1 \dots e_m$ ) =  $\begin{cases} 0, m = 0 \\ 1 + \text{lungime}(e_2 \dots e_m), \text{ altfel} \end{cases}$

perm ( $e$ ) = permutare ( $e, m, \text{lg}(e)$ ), unde  $m = \text{lungime}(e)$   
 $e = \text{candidat}(e)$

main ( $e$ ) =  $\bigcup \text{perm}(e)$

% candidat ( $L$  - lista,  $E$  - Element)

% model de flux:  $(i, e), [i, i] \rightarrow \text{medeterminist}$

%  $(i, e) \rightarrow$  va genera element din lista data

%  $(i, i) \rightarrow$  va verifica apartenența elementului curent din lista data

candidat ( $[H \setminus I], H$ ).

candidat ( $[ - IT], R$ ):

candidat ( $T, R$ ).

% permuteare ( $L$  - lista,  $N$  - Nr,  $lg$  - Nr,  $coe$  - lista,  $R$  - lista)

%  $lg$  - lungimea curenta a permutarii

%  $coe$  - permutarea curenta

%  $N$  - lungimea listei  $L$

% model de flux:  $(i, i, i, i, e) \rightarrow \text{medeterminist}$

permutare ( $L$ ,  $N$ ,  $N$ ,  $C_{\text{LP}}$ ,  $C_{\text{LP}}$ ).

permutare ( $L$ ,  $N$ ,  $Lg$ ,  $[HIT]$ ,  $R$ ): -

% se genereză elemente

candidat ( $L$ ,  $E$ ),

% se verifică condiții

$\text{CLS}(E - \{+\}) = 3$ ,

$\text{metr}(\text{candidat}([HIT], E))$ , % s-a respectat predicatul candidat

% ce model de flux ( $i, i$ )  $\rightarrow$  verifică apartenența ei. Ea e lista

$Lg \in Lg + L$ ,

permutare ( $L$ ,  $N$ ,  $Lg$ ,  $[EITHITSS]$ ,  $R$ ).

% e lungime ( $L$  - Lista,  $R$  - Nr)

% corectarea lungimii unei este

% model de flux:  $(i, a) \rightarrow$  deterministic

lungime ( $[3, 0]$ ).

lungime ( $[IT]$ ,  $R$ ): -

lungime ( $i$ ,  $R$ ),

$R \in R, +L$ .

% perm ( $L$  - lista,  $R$  - lista)

% model de flux:  $(i, a) \rightarrow$  nedeterminist

perm ( $L$ ,  $R$ ): -

% se generază un elem. pt. inițializarea colectoarei

% corectarea nu poate fi vidată, deoarece în predicatul

% "permutare", una din condiții este verificata cu ajutorul

% primului elem. din corectare  $\Rightarrow$  trebuie să existe

candidat ( $L$ ,  $E$ ),

lungime ( $L, N$ ),

permutare ( $L, N, L, [EITSS]$ ,  $R$ ).

• mcm (L - lista, R - lista de liste)

• se returnează lista cu toate permutările valabile

• model de flux:  $(i, \omega) \rightarrow$  deterministic

mcm(L, R) : -

functie  $R \rightarrow \text{perm}(L, R_1, R_2)$ .

5. Sa se dă nr. de subliste de la care mcm pt. care permutare atom numerică (la care mcm) este impar.

Ideea algoritmului:

1. Funcție "permută"  $\rightarrow$  va returna permută atom numerică, de la care mcm, pentru că există
2. Funcție "subliste"  $\rightarrow$  numără subliste care îndeplinesc condiția din cerință, respectându-se ale funcției mcm; mapări, pe care se va aplica o sumă

Modele matematice:

$$\text{permută } (e_1 \dots e_n) = \begin{cases} \emptyset, & n=0 \\ e_1, & e_1 \text{ e atom numeric} \\ \text{permută } (e_1), & e_1 \text{ e lista și} \\ & \text{permută } (e_1) \neq \emptyset \\ \text{permută } (e_2 \dots e_n), & \text{altele} \end{cases}$$

$$\text{subliste } (P) = \begin{cases} \emptyset, & \text{dacă } P \text{ e atom} \\ & n \\ 1 + \sum_{i=1}^n \text{subliste } (e_i), & \text{dacă } P \text{ e lista și} \\ & \text{permută } (P) \text{ e impar} \\ \sum_{i=1}^n \text{subliste } (e_i), & \text{altele} \\ & \text{și permută } (P) \neq \emptyset \end{cases}$$

; primul ( $\exists \rightarrow$  există de elemente)  
 ; returnează cînd nu sunt nici unul  
 ; de la primul primul ( $\exists$ )  
 (cond  
 ((null e) nil) ; cînd în care nu există atam numeric  
 ((numberp (car e)) (car e))  
 ((and (listp (cdr e)) (primul (cdr e))) (primul (cdr e)))  
 ; se cauteră primul atam numeric în prima listă  
 (+ (primul (cdr e)))  
 )

; succinte ( $\exists \rightarrow$  există de elemente)  
 ; returnează nr. de succinte care au primul atam numeric  
 ; impar  
 (defun succinte (e)  
 (cond  
 ((atam e) 0) ; met (succ (primul e)))  
 ((and (listp e) (oddp (primul e)))  
 (+ 1 (apply #'+ (mapcar #'succinte (cdr e)))))  
 ; dacă am găsit o succintă validă, atunci  
 ; incrementăm nr. de succinte ce și sunt succinte  
 ; mai primări elementele succintei actuale  
 (+ (apply #'+ (mapcar #'succinte (cdr e))))  
 ; caut succinte mai valide în succinta curentă  
 ; (care nu e validă)  
 )

4. Se se determine calea de la radacina catre un nod dat.

$$\text{exist}(\ell_1 \dots \ell_n, e) = \begin{cases} \emptyset, n=0 \\ \text{true}, \ell_1 = e \\ \text{exist}(\ell_1) \vee \text{exist}(\ell_2 \dots \ell_n, e), \ell_1 \text{ exist} \\ \text{exist}(\ell_2 \dots \ell_n, e), \ell_1 \text{ este} \end{cases}$$

$$\text{obrum}(\ell, x) = \begin{cases} \emptyset, \ell \text{ este atom si } \ell \neq x \\ (\ell), \ell \text{ este atom si } \ell = x \\ \ell_1 \oplus \cup \text{obrum}(\ell_i, x), \text{ exist}(\ell, x) = \text{true} \\ \emptyset, \text{ altfel} \end{cases}$$

ia parinte

; exist ( $\ell$  - lista,  $e$  - Element)

; returnez o true daca gaseste elem in lista

(defin exista ( $f$   $e$ ))

(cond

( (null  $e$ ) true )

( (equal (car  $e$ )  $e$ ) true )

( (existp (car  $e$ )) (or (exista (car  $f$ )  $e$ ) (exista (cdr  $f$ )  $e$ )))

; daca parintele prim. e lista, se continua cu el in lista elementului,

; sau si in restul listei de la urmator

( true (exista (cdr  $f$ )  $e$ )) )

)

)

; obrum ( $\ell$  - lista,  $x$  - Element)

; returnez o lista ce continde numerele care sunt dat, se reprezinta

; obrumul de la radacina la mediea  $x$

(defin obrum ( $e$   $x$ ))

(cond

( (and (atom  $e$ ) (equal  $e$   $x$ )) (list  $e$ ))

; inseamna ca am gasit nodul cu valoare incepe

; construirea drumului

( (atam e) mie )

; dacă nu gasim modul care ne interesează, returnăm

; există ușor a.i. să nu fie implementat rezpectiv

; final (mapam elementul mi-e-ului)

( (exists e x) (append (exist (car e))

(mapam #' (car e) (y))

(drum y x)

)

(cdr e)

) )

)

; dacă gasim modul în succesiunele curente, adăugăm

; pozitionează, adică primul element, ea drum și

; restul modurilor

; s-a utilizat funcția car, decarece

; apelul funcției "drum" îl continuă și pe x

(+ mie)

; dacă nu gasim modul în succesiunele curente,

; trebuie să se

)

)

Ideea algoritmului:

1. Funcție "exists" → rolul principal este acela de a determina dacă un mod anume se poate întâlni succesiunea curent

2. Funcție "drum" → construiește drumul propriu-zis

rezultat:

- dacă ești curosant este subiectivă, se verifică dacă succintările conțin modulele cunoscute
- dacă conține ⇒ se adaugă patru mărturii la obiect și se continuă căutarea în succesiune săi
- nu îl conține ⇒ se returnează lista video-  
(nu se merge mai departe)
- dacă ești curosant e afara
- este module cunoscute ⇒ se adaugă la obiect
- nu este module cunoscute ⇒ se returnează lista video-