Seminar 5 – Programare recursivă în Lisp

- În Lisp listele sunt (implicit) eterogene. Astfel, dacă problema nu specifică altfel vom lucra cu liste neliniare, adică lista are subliste (care pot conține alte subliste și așa mai departe) și conțin atât elemente numerice cât și elemente nenumerice (atomi).
- În Lisp informațiile se reprezintă sub forma de listă. Nu numai datele pentru care vom apela funcțiile noastre, dar și codul scris de noi este de fapt o listă. Orice listă are ca prim element numele funcției de apelat, iar restul elementelor sunt parametri pentru apel.
    - *max* este o funcție care returnează numărul maxim dintre parametri (poate să primească orice număr de parametri). În alte limbaje de programare, funcția max ar fi apelată așa: *max (1,2,3,4,5,6,7)*. În Lisp, funcția max se apelează așa *(max 1 2 3 4 5 6 7)*.
    - Dacă dorim ca dintr-o listă primul element să nu fie considerat numele unei funcții, trebuie să punem apostrof în fața listei. '(1 2 3 4 5 6 7). Dacă scriem doar lista, fără apostrof, Lisp ne va da o eroare: *1 is not a function name* (în Clisp) sau *Illegal argument in functor position: 1 in (1 2 3 4 5 6 7)* (în LispWorks)
- Când definim o funcție nouă, această funcție este tot o listă care începe cu cuvântul *defun,* urmat de numele funcției, lista de parametri și corpul funcției.

```
(defun numeFunctie (param1 param2 ...) corp_functie)
```

- Corpul funcției de cele mai multe ori conține mai multe forme care pot fi condiții și instrucțiunea de executat dacă condiția este adevărată (foarte similar cu modelul matematic). Pentru aceste condiții folosim funcția *cond* (care seamănă cu *switch* din alte limbaje) și care poate să conțină orice număr de condiții.

```
(defun numeFunctie (param1 param2 …)
(cond
      (cond_1 ins_1)
      (cond_2 ins_2)
      (cond_3 ins_3)
      …
      (cond_n ins_n)
)
)
```

1. Definiți o funcție care interclasează fără păstrarea dublurilor două liste liniare sortate.

$$interclasare\ (l_1 l_2 \ldots l_n, k_1 k_2 \ldots k_m) = \begin{cases} l_1 l_2 \ldots l_n, dacă\ m = 0 \\ k_1 k_2 \ldots k_m, dacă\ n = 0 \\ l_1 \cup interclasare\ (l_2 \ldots l_n, k_1 \ldots k_m), dacă\ l1 < k_1 \\ k_1 \cup interclasare\ (l_1 \ldots l_n, k_2 \ldots k_m), dacă\ k_1 < l_1 \\ l_1 \cup interclasare\ (l_2 \ldots l_n, k_2 \ldots k_n), dacă\ l_1 = k_1 \end{cases}$$

```
(defun interclasare (l1 l2)
(cond
       ((null l2) l1)
       ((null l1) l2)
       ((< (car l1) (car l2)) (cons (car l1) (interclasare (cdr l1) l2)))
       ((> (car l1) (car l2)) (cons (car l2) (interclasare l1 (cdr l2))))
       (t (cons (car l1) (interclasare (cdr l1) (cdr l2))))
)
)
```

- În Lisp există 3 funcții pentru a crea liste: *cons*, *list* și *append*. Să vedem cum arată listele returnate în funcție de parametri.

	cons	**list**	**append**
'A 'B	(A . B)	(A B)	Error
'A '(B C D)	(A B C D)	(A (B C D))	Error
'(A B C) '(D E F)	((A B C) D E F)	((A B C) (D E F))	(A B C D E F)
'(A B C) 'D	((A B C) . D)	((A B C) D)	(A B C . D)
'A 'B 'C 'D	Error	(A B C D)	Error
'(A B) '(C D) '(E F)	Error	((A B) (C D) (E F))	(A B C D E F)
'(A B) 'C '(E F) 'D	Error	((A B) C (E F) D)	Error
'(A B) '(E F) 'D	Error	((AB) (E F) D)	(A B E F . D)

2. Definiți o funcție care elimină toate aparițiile unui element dintr-o listă neliniară.

$$elimAll(l_1 \dots l_n, e) = \begin{cases} \emptyset, n = 0 \\ elimAll(l_1) \cup elimAll(l_2 \dots l_n), dacă\ l_1 este\ listă \\ elimAll(l_2 \dots l_n), dacă\ l_1 = e \\ l_1 \cup elimAll(l_2 \dots l_n), altfel \end{cases}$$

```
(defun elimAll (l e)
(cond
       ((null l) nil)
       ((listp (car l)) (cons (elimAll (car l) e) (elimAll (cdr l) e)) )
       ((equal (car l) e) (elimAll(cdr l) e))
       (t (cons (car l) (elimAll (cdr l) e)))
)
)
```

3. Să se construiască o listă cu pozițiile pe care se găsește elementul numeric minim într-o listă liniară.

- Avem nevoie de 3 funcții pentru a rezolva problema:
    - O funcție care să calculeze minimul unei liste liniare (dar care poate să conțină elemente nenumerice)
    - E important de observat că o variantă de soluție care tot verifică dacă primul element din listă este egal cu minimul listei (calculat pe loc) nu este corect, pentru că în acest caz minimul listei se tot modifică. De exemplu, în lista (1 2 3 4 5) pe parcursul apelurilor recursive, primul element din listă ar fi tot timpul egal cu minimul listei (1 e minimul listei (1 2 3 4 5), 2 e minimul listei (2 3 4 5), 3 e minimul listei (3 4 5), etc.). De aceea ori

trebuie să transmitem ca parametru și lista originală, și să calculăm minimul acestei liste, ori transmitem ca parametru elementul minim pentru care calculăm pozițiile.

- Funcție care să combine primele 2 funcții.

$$minim(l_1 \dots l_n) = \begin{cases} 10000, n = 0 \\ \min(l_1, minim(l_2 \dots l_n), dacă\ l_1 este\ număr \\ minim(l_2 \dots l_n), altfel \end{cases}$$

```
(defun minim (l)
(cond
      ((null l) 10000)
      ((numberp (car l)) (min (car l) (minim (cdr l))))
      ((atom (car l)) (minim (cdr l)))
)
)
```

$$pozitii(l_1 \dots l_n, e, pc) = \begin{cases} \emptyset, n = 0 \\ pc \cup pozitii(l_2 \dots l_n, e, pc + 1), dacă\ l_1 = e \\ pozitii(l_2 \dots l_n, e, pc + 1), altfel \end{cases}$$

```
(defun pozitii (l e p)
(cond
      ((null l) nil)
      ((equal (car l) e) (cons p (pozitii (cdr l) e (+ 1 p))))
      (t (pozitii (cdr l) e (+ 1 p)))
)
)
```

$$pozMain(l_1 \dots l_n) = pozitii(l_1 \dots l_n, minim(l_1 \dots l_n), 1)$$

```
(defun pozMain (l) (pozitii l (minim l) 1))
```

Există o altă variantă de a rezolva problema când într-o singură parcurgere a listei căutăm minimul și în același timp construim și lista cu pozițiile minimului. Vom considera un *minim curent* (o variabilă colectoare), o poziție curentă și o listă cu pozițiile unde am găsit minimul curent (încă o variabilă colectoare). La fiecare pas, primul element din lista de parcurs poate fi:

- Atom nenumeric – mergem mai departe, crescând pozitia curentă
- Atom numeric egal cu minimul curent – am găsit o nouă poziție pe care se găsește minimul curent, adăugăm poziția curentă în lista colectoare cu poziții.
- Atom numeric mai mic decât minimul curent – am găsit un nou minim curent, vom merge mai departe cu acest element, iar lista colectoare devine o nouă listă care conține doar poziția curenta (renunțăm la pozițiile adunate până acum, pentru că sunt pozițiile unui element care de fapt nu este minimul final).
- Atom numeric mai mare decât minimul curent – mergem mai departe, crescând poziția curentă.

Când lista se termină, în lista colectoare avem pozițiile minimului listei.

Cum inițializăm minimul curent?

- O variantă este să punem primul element din lista inițială ca minim curent. Dar acest element poate fi atom numeric sau atom nenumeric. De aceea, în algoritmul descris mai sus (care pornea de la ideea că minimul curent e un număr) mai intervine o ramură care verifică dacă elementul curent e atom numeric și minimul curent este atom nenumeric, pur și simplu luăm atomul numeric ca minim curent și în lista colectoare punem poziția pe care l-am găsit.

- O variantă foarte similară este să pornim cu minimul curent cu valoarea nil (tot avem nevoie de aceea ramură extra care să seteze la prima valoarea numerică, descrisă mai sus).
- O altă variantă este să facem o funcție care caută primul număr în lista inițială și să punem minim curent această valoare.

În continuare vom face implementarea în care minimul curent va fi primul element din listă.

$$pozMin\ (l_1 l_2 ... l_n, minC, pozC, listPoz) = \begin{cases} listPoz, n = 0 \\ pozMin(l_2 ... l_n, minC, pozC + 1, listPoz), l_1 este\ atom\ nenumeric \\ pozMin(l_2 ... l_n, l_1, pozC + 1, (pozC)), minC\ nu\ este\ număr \\ pozMin(l_2 ... l_n, minC, pozC + 1, listPoz\ U\ pozC), minC = l_1 \\ pozMin(l_2 ... l_n, l_1, pozC + 1, (pozC)),\ l_1 < minC \\ pozMin(l_2 ... l_n,\ minC, pozC + 1, listPoz), altfel \end{cases}$$

```
(defun pozMin (l minC pozC listPoz)
(cond
	((null l) listPoz)
	((not (numberp (car l))) (pozMin (cdr l) minC (+ pozC 1) listPoz))
	((not (numberp minC)) (pozMin (cdr l) (car l) (+ pozC 1) (list pozC)))
	((= minC (car l)) (pozMin (cdr l) minC (+ pozC 1) (append listPoz (list pozC))))
	((< (car l) minC) (pozMin (cdr l) (car l) (+ pozC 1) (list pozC)))
	(t (pozMin (cdr l) minC (+ pozC 1) listPoz))
)
)
```

Ne mai trebuie și funcția principală care să facă primul apel la pozMin:

$$pozMinMain(l_1 ... l_n) = pozMin(l_1 ... l_n, l_1, 1, \emptyset)$$

```
(defun pozMinMain (l)
	(pozMin l (car l) 1 nil)
)
```

4. Dându-se o listă liniară, să se adauge în listă un element din N în N.

$$addN(l_1 ... l_n, e, pc, N) = \begin{cases} \emptyset, n = 0 \\ e \cup addN(l_1 ... l_n, e, pc + 1, N), dacă\ pc\ mod\ N = 0 \\ l_1 \cup addN(l_2 ... l_n, e, pc + 1, N), altfel \end{cases}$$

```
(defun addN (l e pc n)
(cond
	((null l) nil)
	((equal 0 (mod pc n)) (cons e (addN l e (+ 1 pc) n)))
	(t (cons (car l) (addN (cdr l) e (+ 1 pc) n)))
)
)
```

- Și funcția principală care să facă primul apel

$$addNMain(l_1 ... l_n, e, N) = addN(l_1 ... l_n, e, 1, N)$$

```
(defun addNMain (l e n)
	(addN l e 1 n)
)
```