

1. Programare funcțională

- Programare orientată spre valoare, programare aplicativă.
- Se focalizează pe valori ale datelor descrise prin expresii (construite prin definiții de funcții și aplicări de funcții), cu evaluare automată a expresiilor.
- Apare ca o nouă paradigmă de programare.
- Programarea funcțională renunță la instrucțiunea de atribuire prezentă ca element de bază în cadrul limbajelor imperative; mai corect spus această atribuire este prezentă doar la un nivel mai scăzut de abstractizare (analog comparației între **goto** și structurile de control structurate **while**, **repeat** și **for**).
- Principiile de lucru ale programării funcționale se potrivesc cu cerințele programării paralele: absența atribuirii, independența rezultatelor finale de ordinea de evaluare și abilitatea de a opera la nivelul unor întregi structuri.
- **Inteligența artificială** a stat la baza promovării programării funcționale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obișnuit sunt calificate ca inteligente. Inteligența artificială, prin problematica aplicațiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informațiilor) necesită, în primul rând, prelucrări simbolice și mai puțin calcule numerice.
- Caracteristica limbajelor de prelucrare simbolică a datelor constă în posibilitatea manipulării unor structuri de date oricât de complexe, structuri ce se construiesc dinamic (în cursul execuției programului). Informațiile prelucrate sunt de obicei șiruri de caractere, liste sau arbori binari.
- **Pur funcțional** = absența totală a facilităților procedurale - controlul memorării, atribuirii, structuri nerecursive de ciclare de tip FOR
- **CE , NU CUM.**
- Limbaje funcționale – LISP (1958), Hope, ML, Scheme, Miranda, Haskell, Erlang (1995)
- LISP nu e PUR funcțional
- Programare funcțională în limbaje precum Python, Scala, F#

2. Limbajul Lisp

- 1958 John McCarthy elaborează o primă formă a unui limbaj (Lisp - LISt Processing) destinat prelucrărilor de liste, formă ce se baza pe ideea transcrierii în acel limbaj de programare a expresiilor algebrice.
- Câteva caracteristici
 - calcule cu expresii simbolice în loc de numere;
 - reprezentarea expresiilor simbolice și a altor informații prin structura de listă;
 - compunerea funcțiilor ca instrument de formare a unor funcții mai complexe;
 - utilizarea recursivității în definiția funcțiilor;
 - reprezentarea programelor Lisp ca date Lisp.
- Consideră *funcția* ca obiect fundamental – transmis ca parametru, returnat ca rezultat al unei prelucrări, parte a unei structuri de date.
- **Domenii de aplicare** – învățare automată (*machine learning*), bioinformatică, comerț electronic (Paul Graham - <http://www.paulgraham.com/avg.html>), sisteme expert, *data mining*, prelucrarea limbajului natural, agenți, demonstrarea teoremelor, învățare automată, înțelegerea vorbirii, prelucrarea imaginilor, planificarea roboților.
- Editorul GNU Emacs de sub Unix e scris în Lisp.
- **GNU CLisp**, GNU Emacs Lisp,...

3. Elemente de bază ale limbajului Lisp

- Un program Lisp prelucrează expresii simbolice (*S-expresii*). Chiar programul este o astfel de S-expresie.
- Modul uzual de lucru al unui sistem Lisp este cel conversațional (interactiv), interpretorul alternând prelucrările de date cu intervenția utilizatorului.
- În Lisp există standardul **CommonLisp** și standardul **CLOS** (Common Lisp Object System) pentru programare orientată obiect.
- Mecanismul implicit de evaluare în **CommonLisp** – evaluarea strictă (*eager/strict evaluation*).
 - CLAZY – evaluarea întârziată (*lazy evaluation*)
- Verificarea tipurilor se face dinamic (la execuție) – *dynamic type checking*.
- Obiectele de bază în Lisp sunt *atomii* și *listele*.

- Datele primare (*atomii*) sunt numerele și simbolurile (simbolul este echivalentul Lisp al conceptului de variabilă din celelalte limbaje). Sintactic, simbolul apare ca un șir de caractere (primul fiind o literă); semantic, el desemnează o S-expresie. Atomii sunt utilizați la construirea listelor (majoritatea S-expresiilor sunt liste).
- O *listă* este o secvență de atomi și/sau liste.
 - Liste *liniare*
 - Liste *neliniare*
- În Lisp s-a adoptat notația *prefixată* (notație ce sugerează și interpretarea operațiilor drept funcții), simbolul de pe prima poziție a unei liste fiind numele funcției ce se aplică.
- *Evaluarea* unei S-expresii înseamnă extragerea (determinarea) valorii acesteia. Evaluarea valorii funcției are loc după evaluarea argumentelor sale.

4. Structuri dinamice de date

- Una dintre cele mai cunoscute și mai simple SDD este *lista liniară simplu înlănțuită* (LLSI).
- Un element al listei este format din două câmpuri: *valoarea* și *legătura* spre elementul următor. Legăturile ne dau informații de natură structurală ce stabilesc relații de ordine între elemente):

| | |
|---------|----------|
| valoare | legătură |
| C1 | C2 |

- Variațiuni ale conținutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conține un pointer atunci se generează arbori binari, iar dacă C2 poate fi și altceva decât pointer atunci apar așa-numitele *perechi cu punct* (elemente cu două câmpuri-dată).
- **Orice listă are echivalent în notația perechilor cu punct**, însă NU orice pereche cu punct are echivalent în notația de listă (în general numai notațiile cu punct în care la stânga parantezelor închise se află NIL pot fi reprezentate în notația de listă). În Lisp, ca și în Pascal, NIL are semnificația de pointer nul.
- Definiția recursivă a echivalenței între liste și perechi cu punct în Lisp:
 - a). dacă A este *atom*, atunci lista (A) este echivalentă cu perechea cu punct (A . NIL)
 - b). dacă lista (l₁ l₂...l_n) este echivalentă cu perechea cu punct <p> atunci lista (l l₁l₂...l_n) este echivalentă cu perechea cu punct (l . <p>)

5. Reguli sintactice

1. **atom numeric** (n) - un șir de cifre urmat sau nu de caracterul '.' și precedat sau nu de semn ('+' sau '-')

2. **atom șir de caractere** (c) - șir de caractere cuprins între ghilimele
3. **simbol** (s) - un șir de caractere, altele decât delimitatorii: spațiu () ' " . virgula = []
 - delimitatorii pot să apară într-un simbol numai dacă sunt evitați (folosind convenția cu "\\")
4. **atom** (a) – poate fi
 - n - atom numeric
 - c – atom șir de caractere
 - s – simbol
5. **listă** (l) - poate fi
 - () lista vidă - NIL
 - (e)
 - (e₁ e₂ ... e_n), n>1
 unde e, e₁,...,e_n sunt **S-expresii**
6. **pereche cu punct** (pp) – este o construcție de forma (e₁ . e₂) unde e₁ și e₂ sunt **S-expresii**
7. **S-expresie** (e) – poate fi
 - a - atom
 - l – listă
 - pp – pereche cu punct
8. **formă** (f) - o S-expresie evaluabilă
9. **program Lisp** este o succesiune de **forme** (S-expresii evaluabile).

6. Reguli de evaluare

- (a) un **atom numeric** se evaluează prin numărul respectiv
- (b) un **șir de caractere** se evaluează chiar prin textul său (inclusiv ghilimelele);
- (c) o **listă** este evaluabilă (adică este **formă**) doar dacă primul ei element este numele unei funcții, caz în care mai întâi se evaluează toate argumentele, după care se aplică funcția acestor valori.

Observație Funcția QUOTE întoarce chiar S-expresia argument, ceea ce este echivalent cu oprirea încercării de a evalua argumentul. În locul lui QUOTE se va putea utiliza caracterul ' (apostrof).

Esența Lisp-ului constă din prelucrarea S-expresiilor. Datele au aceeași formă cu programele, ceea ce permite lansarea în execuție ca programe a unor structuri de date precum și modificarea unor programe ca și cum ele ar fi date obișnuite.

6. Funcții Lisp

Prelucrările de liste se pot face la:

- nivel superficial
- la orice nivel

(CONS e_1 e_2): l sau pp

- funcția **constructor**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- formează o perechi cu punct având reprezentările celor două SE în cele două câmpuri. Elementul CONS construit în acest fel se numește *celulă CONS*. Valorile argumentelor nu sunt afectate.

(CAR l sau pp): e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- extrage primul element al unei liste sau partea stângă a unei perechi cu punct

(CDR l sau pp): e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- extrage lista fără primul element sau respectiv partea dreaptă a unei perechi cu punct.

(LIST e_1 e_2 ...): l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- întoarce lista valorilor argumentelor la nivel superficial.

(APPEND e_1 e_2 ... e_n): e

- se evaluează argumentele și apoi se trece la evaluarea funcției
- copiază structura fiecărui argument, înlocuind CDR-ul fiecărui ultim CONS cu argumentul din dreapta. Întoarce lista rezultată.
- toți parametrii atomi, eventual cu excepția ultimului parametru, sunt ignorați.
- Dacă toate S-expresiile argument sunt liste, atunci efectul este concatenarea listelor
- funcția APPEND este puternic consumatoare de memorie; prima listă argument este recopiată înainte de a fi legată de următoarea

(LENGTH l): n

- se evaluează argumentul și apoi se trece la evaluarea funcției
- întoarce numărul de elemente ale unei la nivel superficial

Predicate de bază în Lisp

- Limbajul Lisp prezintă atomii speciali NIL, cu semnificația de fals, și T, cu semnificația de adevărat. Ca și în alte limbaje, funcțiile care returnează o valoare logică vor returna exclusiv NIL sau T. Totuși, se acceptă că orice valoare diferită de NIL are semnificația de adevărat.

(ATOM e) : T, NIL

- se evaluează argumentul și apoi se trece la evaluarea funcției
- T dacă argumentul este un atom și NIL în caz contrar.

(LISTP e) : T, NIL

- se evaluează argumentul și apoi se trece la evaluarea funcției
- T dacă argumentul este o listă și NIL în caz contrar.

(EQUAL e1 e2) : T, NIL

- se evaluează argumentele și apoi se trece la evaluarea funcției
- întoarce T dacă valorile argumentelor sunt S-expresii echivalente (adică dacă cele două S-expresii au aceeași structură)

(NULL e) : T, NIL

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Întoarce T dacă argumentul se evaluează la o listă vidă sau atom nul și NIL în caz contrar.

(NUMBERP e) : T, NIL

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Verifică dacă argumentul este număr sau nu.

Operații logice

(NOT e) : T, NIL

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Întoarce T dacă argumentul e este evaluat la NIL; în caz contrar, întoarce NIL. E echivalentă cu funcția NULL.

(AND e1 e2 ...) : e

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluează de la stânga la dreapta până la primul NIL, caz în care se returnează NIL; în caz contrar rezultatul este valoarea ultimului argument.

(OR e1 e2 ...) : e

- Evaluarea argumentelor face parte din procesul de evaluare al funcției
- Se evaluează de la stânga la dreapta până la primul element evaluat la o valoare diferită de NIL, caz în care se returnează acea valoare; în caz contrar rezultatul este NIL.

Operații aritmetice

(+ n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției

- rezultat: $n_1+n_2+\dots$

(- n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n_1-n_2-\dots$

(* n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n_1*n_2*\dots$

(/ n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: $n_1/n_2/\dots$

(MAX n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: maximul valorilor argumentelor

(MIN n1 n2 ...) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: minimul valorilor argumentelor

Pentru numere întregi se pot folosi MOD și DIV.

Operatori relaționali

Sunt cei uzuali: = (doar pentru numere), <, <=, >, >=

7. Ramificarea prelucrărilor. Funcția COND

Funcția COND este asemănătoare selectorilor CASE sau SWITCH din Pascal, respectiv C.

(COND (l1 l2...ln)): e

- evaluarea argumentelor face parte din procesul de evaluare al funcției

În descrierea de mai sus **l1, l2,...ln** sunt liste nevide de lungime arbitrară (**f1 f2 ... fn**) numite *clauze*. COND admite oricâte clauze ca argumente și oricâte forme într-o clauză. Iată modul de funcționare a funcției COND:

- se parcurg pe rând clauzele în ordinea apariției lor în apel, evaluându-se doar primul element din fiecare clauză până se întâlnește unul diferit de NIL. Clauza respectivă va fi selectată și se trece la evaluarea în ordine a formelor f_2, f_3, \dots, f_n . Se întoarce valoarea ultimei forme evaluate din clauza selectată;
- dacă nu se selectează nici o clauză, COND întoarce NIL.

8. Definirea funcțiilor utilizator. Funcția DEFUN

(DEFUN s l f1 f2...): s

Funcția DEFUN crează o nouă funcție având ca nume primul argument (simbolul **s**), iar ca parametri formali elementele simboluri ale listei ce constituie al doilea argument; corpul funcției create este alcătuit din una sau mai multe forme aflate, ca argumente, pe pozițiile a treia și eventual următoarele (evaluarea acestor forme la execuție reprezintă efectul secundar). Se întoarce numele funcției create. Funcția DEFUN nu-și evaluează nici un argument.

Apelul unei funcții definite prin

(DEFUN **fnume** (p1 p2 ... pn)

...

)

este o formă

(**fnume** arg1 arg2 ... argn)

unde **fnume** este un simbol, iar **arg_i** sunt forme; evaluarea apelului decurge astfel:

- (a) se evaluează argumentele arg₁, arg₂, ... arg_n; fie v₁, v₂, ... v_n valorile lor;
- (b) fiecare parametru formal din definiția funcției este legat la valoarea argumentului corespunzător din apel (p₁ la v₁, p₂ la v₂, ..., p_n la v_n); dacă la momentul apelului simbolurile reprezentând parametri formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare;
- (c) se evaluează în ordine fiecare formă aflată în corpul funcției, valoarea ultimei forme fiind întoarsă ca valoare a apelului funcției;
- (d) se restaurează valorile parametrilor formali, adică p₁, p₂ ... p_n se “dezleagă” de valorile v₁, v₂, ... și se leagă din nou la valorile corespunzătoare salvate (dacă este cazul).

Observație. DEFUN poate redefini și funcțiile sistem standard. Spre exemplu dacă se redefinește funcția CAR astfel: (DEFUN CAR(L) (CDR L)), atunci (CAR '(1 2 3)) se va evalua la (2 3).

9. SET, SETQ, SETF

Pentru a da valori simbolurilor în Lisp se folosesc funcțiile cu efect secundar SET și SETQ.

Acțiunea prin care o funcție, pe lângă calculul valorii sale, realizează și modificări ale structurilor de date din memorie se numește *efect secundar*.

(SET $s_1 f_1 \dots s_n f_n$): e

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect:
 - valorile argumentelor de rang par (f_i) devin valorile argumentelor de rang impar corespunzătoare evaluate în prealabil la simboluri (s_i)
- rezultatul întors valoarea ultimei forme evaluate (f_n)

(SETQ $s_1 f_1 \dots s_n f_n$): e

- se evaluează doar formele f_1, \dots, f_n
- efect:
 - valorile argumentelor de rang par (f_i) devin valorile argumentelor de rang impar corespunzătoare neevaluate (s_i)
- rezultatul întors valoarea ultimei forme evaluate (f_n)

(SETF $f_1 f'_1 \dots f_n f'_n$): e

- este un macro, are efect distructiv
- f_1, \dots, f_n sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp
- f'_1, \dots, f'_n sunt forme ale căror valori vor fi legate de locațiile desemnate de parametrii f_1, \dots, f_n corespunzători.
- efect:
 - se evaluează formele de rang par (f'_i) și valorile acestora se leagă de locațiile desemnate de formele f_i corespunzătoare
- rezultatul întors valoarea ultimei forme evaluate (f'_n)

10. Definirea funcțiilor anonime. Expresii LAMBDA

În situațiile în care

- funcție folosită o singură dată este mult prea simplă ca să merite a fi definită
- funcția de aplicat trebuie sintetizată dinamic (nu este, deci, posibil să fie definită static prin DEFUN)

se poate utiliza o formă funcțională particulară numită **expresie lambda**.

O expresie lambda este o listă de forma

(LAMBDA l $f_1 f_2 \dots f_m$)

ce definește o funcție anonimă utilizabilă doar local, o funcție ce are definiția și apelul concentrate în același punct al programului ce le utilizează, l fiind lista parametrilor iar $f_1 \dots f_m$ reprezentând corpul funcției.

Argumentele unei expresii lambda sunt evaluate la apel. Dacă se dorește ca argumentele să nu fie evaluate la apel trebuie folosită forma **QLAMBDA**.

O astfel de formă LAMBDA se folosește în modul uzual:

((LAMBDA l f₁ f₂ ... f_m) par₁ par₂ ... par_n)

11. Mecanisme definiționale evaluate

Argumente funcționale

| Argument | | |
|------------------|------------|----------------------|
| Funcție f | # f | (function f) |
| Simbol x | ' x | (quote x) |

| | Funcție f |
|--------------------------------------|------------------|
| Standard | #'f |
| CLisp | 'f |
| GCLisp, Emacs Lisp, alte dialecte | 'f |

| | Expresie Lambda |
|--------------------------------------|-----------------|
| Standard | #'(lambda) |
| CLisp | (lambda ...) |
| GCLisp, Emacs Lisp, alte dialecte | '(lambda) |

Forma EVAL

Aplicarea formei EVAL este echivalentă cu apelul evaluatorului Lisp. Sintaxa funcției este

(EVAL f) : e

Efectul constă în evaluarea formei și returnarea rezultatului evaluării. Forma este evaluată de fapt în două etape: mai întâi este evaluată ca argument al lui EVAL iar apoi rezultatul acestei evaluări este din nou evaluat ca efect al aplicării funcției EVAL. De exemplu:

- (SETQ X '((CAR Y) (CDR Y) (CADR Y)))
- (SETQ Y '(A B C))
- (CAR X) se evaluează la (CAR Y)
- (EVAL (CAR X)) va produce A

Mecanismul este asemănător cu ceea ce înseamnă indirectarea prin intermediul pointerilor din cadrul limbajelor imperative.

- (SETQ L '(1 2 3))
- (SETQ P '(CAR L))
- P se evaluează la (CAR L)
- (EVAL P) va produce 1
- (SETQ B 'X)
- (SETQ A 'B)
- (EVAL A) se evaluează la X
- (SETQ L '(+ 1 2 3))
- L se evaluează la (+ 1 2 3)
- (EVAL L) se evaluează la 6

Observație. Lisp nu evaluează primul element dintr-o formă, ci numai îl aplică.

Ex: Asocierea (SETQ Q 'CAR) nu permite apelul sub forma (Q '(A B C)), un astfel de apel semnaland eroare în sensul că evaluatorul LISP nu găsește nici o funcție cu numele Q. Pe de altă parte, nici numai cu ajutorul funcției EVAL nu putem rezolva problema:

- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (EVAL P) va produce CAR
- ((EVAL P) '(A B C)) va produce mesaj de eroare: “Bad function when ...”

Mesajul de eroare de mai sus apare deoarece Lisp nu-și evaluează primul argument dintr-o formă.

!!! O listă este totdeauna evaluată dacă acest lucru nu este oprit explicit (prin QUOTE), în schimb primul argument al oricărei liste nu este niciodată evaluat!

Forme funcționale. Funcțiile APPLY și FUNCALL

Funcțiile APPLY și FUNCALL permit aplicarea unei funcții asupra unei mulțimi de parametri sintetizată eventual dinamic.

(APPLY ff lp):e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Funcția APPLY permite aplicarea unei funcții asupra unor parametri furnizați sub formă de listă. În descrierea de mai sus, **ff** este o formă funcțională și **lp** este o formă reductibilă prin evaluare la o listă de parametri efectivi ($p_1 p_2 \dots p_n$).

(FUNCALL ff l1 l2 ...ln):e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- FUNCALL este o variantă a funcției APPLY care permite aplicarea unei funcții (sau expresii) rezultate prin evaluarea unei forme funcționale **ff** asupra unui număr fix de parametri.

12. Funcții MAP

Rolul funcțiilor MAP este de a aplica o funcție în mod repetat asupra elementelor (sau sublistelor succesive) listelor date ca argumente.

(MAPCAR f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - CAR-ului listelor => e₁
 - CADR-ului listelor => e₂
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

(NCONC l₁ l₂ ... l_n) : l

Relativ la modificarea sau nu a structurii listelor implicate, concatenarea de liste se poate efectua în două maniere: cu modificarea listelor (folosind funcția NCONC) și fără (folosind funcția APPEND)

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: **NCONC** realizează concatenarea efectivă (fizică) prin modificarea ultimului pointer (cu valoarea NIL) al primelor n-1 argumente și întoarce primul argument, care le va îngloba la ieșire pe toate celelalte.

(MAPCAN f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - CAR-ului listelor => e₁
 - CADR-ului listelor => e₂
 - CADDR-ului listelor => e₃
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

(MAPLIST f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - listelor => e₁
 - CDR-ului listelor => e₂
 - CDDR-ului listelor => e₃
 -până când una din liste ajunge vidă

- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

(MAPCON f l₁ ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
 - listelor => e₁
 - CDR-ului listelor => e₂
 - CDDR-ului listelor => e₃
 -până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

14. Argumente opționale

În lista parametrilor formali ai unei funcții putem folosi următoarele variabile: **&OPTIONAL** și **&REST**.

- **&OPTIONAL** - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare parametrul corespunzător din lista parametrilor actuali, respectiv NIL dacă parametrul actual nu există;
- **&REST** - dacă apare în lista parametrilor formali atunci următorul parametru formal va lua ca valoare lista parametrilor actuali rămași neatribuiți, respectiv NIL dacă nu mai există parametri actuali neatribuiți.

Exemple

```
(defun f (x &optional y)
  (cond
    ((null y) x)
    (t (+ x y))
  )
)
```

- (f 1 2) → 3
- (f 3) → 3

```
(defun g (x &rest y)
  (+ x (apply #' + y))
)
```

- (g 1 2 3) → 6
- (g 4 5) → 9
- (g 3) → 3

15. Gestiunea obiectelor în LISP. OBLIST și ALIST.

Un obiect Lisp este o structură alcătuită din:

- numele simbolului;
- valoarea sa;
- lista de proprietăți asociată simbolului.

Gestiunea simbolurilor folosite într-un program Lisp este realizată de sistem cu ajutorul unei tabele speciale numită lista obiectelor (**OBLIST**). Orice simbol este un obiect unic în sistem. Apariția unui nou simbol determină adăugarea lui la OBLIST. La fiecare întâlnire a unui atom, el este căutat în OBLIST. Dacă se găsește, se întoarce adresa lui.

Pentru implementarea mecanismului de apel, sistemul Lisp folosește o altă listă, numită lista argumentelor (**ALIST**). Fiecare element din **ALIST** este o pereche cu punct formată dintr-un parametru formal și argumentul asociat sau valoarea acestuia.

În general, o funcție definită cu n parametri P_1, P_2, \dots, P_n și apelată prin $(F A_1 A_2 \dots A_n)$ va adăuga la ALIST n perechi de forma $(P_i . A_i)$ dacă funcția își evaluează argumentele sau $(P_i . A_i')$, unde A_i' este valoarea argumentului A_i , dacă mediul Lisp evaluează argumentele. De exemplu, pentru funcția

```
(DEFUN F (A B)
  (COND
    ((ATOM B) A)
    (T (CONS A B)))
)
```

apelată cu $(F 'X '(1.1))$, vom avea pentru ALIST structura $(\dots (A . X) (B . (1 . 1)))$.

Dacă la momentul apelului simbolurile reprezentând parametrii formali aveau deja valori, acestea sunt salvate în vederea restaurării ulterioare.

Simbolurile sunt reprezentate în structură prin adresa lor din OBLIST, iar atomii numerici și cei șir de caractere prin valoarea lor direct în celula care îi conține.

Inițial, la începutul programului, ALIST este vidă. Pe măsura evaluării de noi funcții, se adaugă perechi la ALIST, iar la terminarea evaluării funcției, perechile create la apel se șterg. În corpul funcției în curs de evaluare valoarea unui simbol s este căutată întâi în perechile din ALIST începând dinspre vârf spre bază (această acțiune este cea care stabilește dinamic domeniul de vizibilitate). Dacă valoarea nu este găsită în ALIST se continuă căutarea în OBLIST. Așadar, este clar că **ALIST și OBLIST formează contextul curent al execuției unui program.**

Exemplu

- $(SETQ X 1)$ inițializează simbolul X cu valoarea 1;

- (SETQ Y 10) inițializează simbolul Y cu valoarea 10;
- (DEFUN DEC (X) (SETQ X (- X 1))) definește o funcție de decrementare a valorii parametrului;
- (DEC X) se evaluează la 0;
- X se evaluează tot la 1;
- (DEC Y) se evaluează la 9;
- Y se evaluează tot la 10.

Să observăm deci că modificările operate asupra valorilor simbolurilor reprezentând argumentele formale se vor pierde după ieșirea din corpul funcției și revenirea în contextul apelator.

16. Macrodefiniții

Din punct de vedere sintactic, macrodefinițiile se construiesc în același mod ca funcțiile, cu diferența că în loc să se folosească funcția DEFUN se va folosi funcția DEFMACRO:

(DEFMACRO s l f1 ... f n): s

Funcția DEFMACRO creează o macrodefiniție având ca nume primul argument (simbolul s), iar ca parametri formali elementele simboluri ale listei ce constituie al doilea argument; corpul macrodefiniției create este alcătuit din una sau mai multe forme aflate, ca argumente, pe a treia poziție și eventual pe următoarele. Valoarea întoarsă ca rezultat este numele macrocomenzii create. Funcția DEFMACRO nu-și evaluează niciun argument.

- evaluarea argumentelor face parte din procesul de evaluare al macrodefiniției

Modul de lucru al macrodefinițiilor create cu DEFMACRO este diferit de cel al funcțiilor create cu DEFUN:

- parametri macrodefiniției nu sunt evaluați;
- se evaluează corpul macrodefiniției și se va produce o S-expresie intermediară;
- această S-expresie va fi evaluată, acesta fiind momentul în care sunt evaluați parametrii.

17. Apostroful invers (backquote)

Apostroful invers (``) ușurează foarte mult scrierea macrocomenzilor. Oferă o modalitate de a crea expresii în care cea mai mare parte este fixă, și în care doar câteva detalii variabile trebuie completate. Efectul apostrofului invers este asemănător cu al apostrofului normal ('), în sensul că blochează evaluarea S-expresiei care urmează, cu excepția că orice **virgulă** (,) care apare va produce *evaluarea* expresiei care urmează. Iată un exemplu:

- (SETQ V 'EXEMPLU)
- ``(ACESTA ESTE UN ,V) se evaluează la (ACESTA ESTE UN EXEMPLU)

De asemenea, apostroful invers acceptă construcția ,@. Această combinație produce și ea evaluarea expresiei care urmează, dar cu diferența că valoarea rezultată trebuie să fie o

listă. Elemente acestei liste sunt dizolvate în lista în care apare combinația ,@. Iată un exemplu:

- (SETQ V '(ALT EXEMPLU))
- `(ACESTA ESTE UN ,V) se evaluează la
(ACESTA ESTE UN (ALT EXEMPLU))
- `(ACESTA ESTE UN ,@V) se evaluează la
(ACESTA ESTE UN ALT EXEMPLU)