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 funcţională renunţă la instrucţiunea 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 comparaţiei între **goto** şi structurile de control structurate **while**, **repeat** şi **for**).
- Principiile de lucru ale programării funcţionale se potrivesc cu cerinţele programării paralele: absenţa atribuirii, independenţa rezultatelor finale de ordinea de evaluare şi abilitatea de a opera la nivelul unor întregi structuri.
- **Inteligenţa artificială** a stat la baza promovării programării funcţionale. Obiectul acestui domeniu constă din studiul modului în care se pot realiza cu ajutorul calculatorului comportări care în mod obişnuit sunt calificate ca inteligente. Inteligenţa artificială, prin problematica aplicaţiilor specifice (simularea unor procese cognitive umane, traducerea automată, regăsirea informaţiilor) necesită, în primul rând, prelucrări simbolice şi mai puţin 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 execuţiei programului). Informațiile prelucrate sunt de obicei şiruri de caractere, liste sau arbori binari.
- **Pur funcțional** = absența totală a facilităților procedurale - controlul memorării, atribuiri, 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 şi a altor informaţii prin structura de listă;
    - compunerea funcţiilor ca instrument de formare a unor funcţii mai complexe;
    - utilizarea recursivităţii în definiţia funcţiilor;
    - 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 conversaţional (interactiv), interpretorul alternând prelucrarile de date cu intervenţia utilizatorului.

- În Lisp există standardul **CommonLisp** și standardul **CLOS** (Commom Lisp Object System) pentru programare orientată obiect.

- Mecanismul implicit de evaluare – evaluarea întârziată (*lazy evaluation*).

- Verificarea tipurilor se face dinamic (la execuție) – *dynamic type checking*.

- Obiectele de bază în Lisp sunt *atomii* şi *listele*.

- Datele primare (*atomii*) sunt numerele şi simbolurile (simbolul este echivalentul Lisp al conceptului de variabilă din celelalte limbaje). Sintactic, simbolul apare ca un şir de caractere (primul fiind o literă); semantic, el desemnează o S-expresie. Atomii sunt utilizaţi la construirea listelor (majoritatea S-expresiilor sunt liste).
- O *listă* este o secvenţă de atomi şi/sau liste.
  - Liste *liniare*
  - Liste *neliniare*

- In Lisp s-a adoptat notaţia *prefixată* (notaţie ce sugerează şi interpretarea operaţiilor drept funcţii), simbolul de pe prima poziţie a unei liste fiind numele funcţiei ce se aplică.

- *Evaluarea* unei S-expresii înseamnă extragerea (determinarea) valorii acesteia. Evaluarea valorii funcţiei are loc după evaluarea argumentelor sale.

4. Structuri dinamice de date

- Una dintre cele mai cunoscute şi mai simple SDD este *lista liniară simplu înlănţuită* (LLSI).

- Un element al listei este format din două câmpuri: *valoarea* şi *legătura* spre elementul următor. Legăturile ne dau informaţii de natură structurală ce stabilesc relaţii de ordine între elemente):

valoare	legătură
C1	C2

- Variaţiuni ale conţinutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conţine un pointer atunci se generează arbori binari, iar dacă C2 poate fi şi altceva decât pointer atunci apar aşa-numitele *perechi cu punct* (elemente cu două câmpuri-dată).

- **Orice listă are echivalent în notaţia perechilor cu punct**, însă NU orice pereche cu punct are echivalent în notaţia de listă (în general numai notaţiile cu punct în care la stânga parantezelor închise se află NIL pot fi reprezentate în notaţia de listă). În Lisp, ca şi în Pascal, NIL are semnificaţia de pointer nul.

- Definiţia recursivă a echivalenţei între liste şi perechi cu punct în Lisp:

  a). dacă A este *atom*, atunci lista (A) este echivalentă cu perechea cu punct (A **.** NIL)
  b). dacă lista (l_1 l_2…l_n) este echivalentă cu perechea cu punct <**p**> atunci lista (l l_1l_2…l_n) este echivalentă cu perechea cu punct (l **.** <**p**>)

5. Reguli sintactice

1. **atom numeric** (n) - un şir de cifre urmat sau nu de caracterul ‘.’ şi precedat sau nu de semn (‘+’ sau ‘-‘)

2. **atom şir de caractere** (c) - şir de caractere cuprins între ghilimele

3. **simbol** (s) - un şir de caractere, altele decât delimitatorii: spaţiu ( ) ' " . virgula = [ ]
    - delimitatorii pot să apară într-un simbol numai dacă sunt evitaţi (folosind convenţia 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)
    - (e_1 e_2 ... e_n), n>1

    unde e, e_1,...,e_n sunt **S-expresii**

6. **pereche cu punct** (pp) – este o construcție de forma (e_1 . e_2) unde e_1 şi e_2 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 **şir 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 funcţii, caz în care mai întâi se evaluează toate argumentele, după care se aplică funcţia acestor valori.

Observaţie Funcţia 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).

Esenţa Lisp-ului constă din prelucrarea S-expresiilor. Datele au aceeaşi formă cu programele, ceea ce permite lansarea în execuţie ca programe a unor structuri de date precum şi modificarea unor programe ca şi cum ele ar fi date obişnuite.

6. Funcţii 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 e1 e2…): l

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

(APPEND e1 e2...en): 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.
- toţi parametrii atomi, eventual cu excepţia ultimului parametru, sunt ignoraţi.
- Dacă toate S-expresiile argument sunt liste, atunci efectul este concatenarea listelor
- funcţia 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 semnificaţia de fals, şi T, cu semnificaţia de adevărat. Ca şi în alte limbaje, funcţiile care returnează o valoare logică vor returna exclusiv NIL sau T. Totuşi, se acceptă că orice valoare diferită de NIL are semnificaţia de adevărat.

(ATOM e) : T, NIL

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

(LISTP e) : T, NIL

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

(NUMBERP e) : T, NIL

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

Operaţii 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 funcţia 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.

Operaţii aritmetice

(+ n1 n2 …) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1+n2+…

(- n1 n2 …) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1-n2-…

(* n1 n2 …) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1*n2*…

(/ n1 n2 …) : n

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat: n1/n2/…

(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. Funcţia COND

Funcţia 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

In descrierea de mai sus **l1**, **l2**,…**ln** sunt liste nevide de lungime arbitrară **(f_1 f_2 ... f_n)** numite *clauze*. COND admite oricâte clauze ca argumente şi oricâte forme într-o clauză. Iată modul de funcţionare a funcţiei COND:

- se parcurg pe rând clauzele în ordinea apariţiei lor în apel, evaluându-se doar primul element din fiecare clauză până se întâlneşte unul diferit de NIL. Clauza respectivă va fi selectată şi se trece la evaluarea în ordine a formelor f_2, f_3, ... 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

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

Apelul unei funcţii definită prin
(DEFUN **fnume** (p_1 p_2 ... p_n)
...
)

este o formă

(**fnume** arg_1 arg_2 ... arg_n)

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

(a) se evaluează argumentele arg_1, arg_2, ... arg_n; fie v_1, v_2, ... v_n valorile lor;
(b) fiecare parametru formal din definiţia funcţiei este legat la valoarea argumentului corespunzător din apel (p_1 la v_1, p_2 la v_2, ..., 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 funcţiei, valoarea ultimei forme fiind întoarsă ca valoare a apelului funcţiei;
(d) se restaurează valorile parametrilor formali, adică p_1, p_2 ... pn se "dezleagă" de valorile v_1, v_2, ... şi se leagă din nou la valorile corespunzătoare salvate (dacă este cazul).

Observaţie. DEFUN poate redefini si funcţiile sistem standard. Spre exemplu dacă se redefineşte funcţia 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 funcţiile cu efect secundar SET şi SETQ.

Acţiunea prin care o funcţie, pe lângă calculul valorii sale, realizează şi modificări ale structurilor de date din memorie se numeşte *efect secundar*.

(SET s_1 f_1 ... 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 ... s_n f_n): e

- se evalueaza doar formele f_1,...,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 ... f_n f'_n): e

- este un macro, are efect destructiv
- f_1,...,f_n sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp
- f'_1,...,f'_n sunt forme ale căror valori vor fi legate de locaţiile desemnate de parametrii f_1, ..., 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 situaţiile în care

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

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

O expresie lambda este o listă de forma

(LAMBDA l f_1 f_2 ... f_m)

ce defineşte o funcţie anonimă utilizabilă doar local, o funcţie ce are definiţia şi apelul concentrate în acelasi punct al programului ce le utilizează, l fiind lista parametrilor iar f_1...f_m reprezentând corpul funcţiei.

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

O astfel de formă LAMBDA se foloseşte în modul uzual:

((LAMBDA l f_1 f_2 ... f_m) par_1 par_2 ... par_n)

11. Mecanisme definiţionale evoluate

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 funcţiei este

(EVAL f) : e

Efectul constă în evaluarea formei şi 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 funcţiei 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

Observaţie. 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 semnalând eroare în sensul că evaluatorul LISP nu găseste nici o funcţie cu numele Q. Pe de altă parte, nici numai cu ajutorul functiei 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-şi 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 funcţionale. Funcţiile APPLY şi FUNCALL

Funcţiile APPLY şi FUNCALL permit aplicarea unei funcţii asupra unei mulţimi de parametri sintetizată eventual dinamic.

(APPLY ff lp):e

- se evaluează argumentul și apoi se trece la evaluarea funcției
- Funcţia APPLY permite aplicarea unei funcţii asupra unor parametri furnizaţi sub formă de listă. În descrierea de mai sus, **ff** este o formă functională şi **lp** este o formă reductibilă prin evaluare la o listă de parametri efectivi (p_1 p_2 ... p_n).

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

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

12. Funcții MAP

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

(MAPCAR f l_1 ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcţia n-ară **f** este aplicată pe rând asupra:
    - CAR-ului listelor => e_1
    - CADR-ului listelor => e_2
    - ….până când una din liste ajunge vidă
  - Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

(NCONC l1 l2 ... ln) : l

Relativ la modificarea sau nu a structurii listelor implicate, concatenarea de liste se poate efectua în două maniere: cu modificarea listelor (folosind funcţia NCONC) şi fară (folosind funcţia 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 şi întoarce primul argument, care le va îngloba la ieşire pe toate celelalte.

(MAPCAN f l_1 ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcţia n-ară **f** este aplicată pe rând asupra:
    - CAR-ului listelor => e_1
    - CADR-ului listelor => e_2
    - CADDR-ului listelor => e_3
    - ….până când una din liste ajunge vidă
  - Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

(MAPLIST f l_1 ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcţia n-ară **f** este aplicată pe rând asupra:
    - listelor => e_1
    - CDR-ului listelor => e_2
    - CDDR-ului listelor => e_3
    - ….până când una din liste ajunge vidă
  - Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

(MAPCON f l_1 ... l_n) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcţia n-ară **f** este aplicată pe rând asupra:
    - listelor => e_1
    - CDR-ului listelor => e_2
    - CDDR-ului listelor => e_3
    - ….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 funcţii 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 rămaşi neatribuiţi, respectiv NIL dacă nu mai există parametri actuali neatribuiţi.

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 şi ALIST.

Un obiect Lisp este o structură alcătuită din:

- numele simbolului;
- valoarea sa;
- lista de proprietăţi 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. Apariţia unui nou simbol determină adăugarea lui la OBLIST. La fiecare întâlnire a unui atom, el este căutat in OBLIST. Dacă se găseşte, se întoarce adresa lui.

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

In general, o funcţie definită cu *n* parametri P_1, P_2, ..., P_n şi apelată prin (F A_1 A_2 ... A_n) va adăuga la ALIST n perechi de forma (P_i . A_i) daca funcţia îşi evaluează argumentele sau (P_i . A_i'), unde A_i' este valoarea argumentului A_i, daca mediul Lisp evaluează argumentele. De exemplu, pentru funcţia

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

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

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

Simbolurile sunt reprezentate în structură prin adresa lor din OBLIST, iar atomii numerici şi cei şir de caractere prin valoarea lor direct în celula care îi conţine.

Iniţial, la începutul programului, ALIST este vidă. Pe măsura evaluării de noi funcţii, se adaugă perechi la ALIST, iar la terminarea evaluării funcţiei, perechile create la apel se şterg. In corpul funcţiei în curs de evaluare valoarea unui simbol s este căutată întâi în perechile din ALIST începând dinspre vârf spre bază (această acţiune este cea care stabileşte dinamic domeniul de vizibilitate). Dacă valoarea nu este găsită în ALIST se continuă căutarea în OBLIST. Aşadar, este clar că **ALIST şi OBLIST formează contextul curent al execuţiei unui program.**

Exemplu

- (SETQ X 1) iniţializează simbolul X cu valoarea 1;
- (SETQ Y 10) iniţializează simbolul Y cu valoarea 10;

- (DEFUN DEC (X) (SETQ X (- X 1))) defineşte o funcţie 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 ieşirea din corpul funcţiei şi revenirea în contextul apelator.

16. Macrodefiniții

Din punct de vedere sintactic, macrodefiniţiile se construiesc în acelaşi mod ca funcţiile, cu diferenţa că în loc să se folosească funcţia DEFUN se va folosi funcţia DEFMACRO:

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

Funcţia DEFMACRO crează o macrodefiniţie având ca nume primul argument (simbolul s), iar ca parametri formali elementele simboluri ale listei ce constituie al doilea argument; corpul macrodefiniţiei create este alcătuit din una sau mai multe forme aflate, ca argumente, pe a treia poziţie şi eventual pe următoarele. Valoarea întoarsă ca rezultat este numele macrocomenzii create. Funcţia DEFMACRO nu-şi evalueaza niciun argument.

- evaluarea argumentelor face parte din procesul de evaluare al macrodefiniției

Modul de lucru al macrodefiniţiilor create cu DEFMACRO este diferit de cel al funcţiilor create cu DEFUN:

- parametrii macrodefiniţiei nu sunt evaluaţi;
- se evaluează corpul macrodefiniţiei şi se va produce o S-expresie intermediară;
- această S-expresie va fi evaluată, acesta fiind momentul în care sunt evaluaţi parametrii.

17. Apostroful invers (backquote)

Apostroful invers (`) uşurează foarte mult scrierea macrocomenzilor. Oferă o modalitate de a crea expresii în care cea mai mare parte este fixă, şi 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 excepţia 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 construcţia **,@**. Această combinaţie produce şi ea evaluarea expresiei care urmează, dar cu diferenţa că valoarea rezultată trebuie să fie o

listă. Elemente acestei liste sunt dizolvate în lista în care apare combinaţia **,@**. 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)