Programare logică şi funcţională avansată(PLFAv)

V. Negru
Universitatea de Vest din Timişoara
Departamentul de Informatică
e-mail: vnegru@info.uvt.ro

Conţinut curs - I

- Noţiuni de bază de programare funcţională şi logică (2 ore)
- Tehnici avansate în Lisp (macrouri) (4 ore)
- Închideri lexicale şi abstractrizare date (4 ore)
- Continuare. Aplicaţii (4 ore)
- Noţiuni de bază de programare logică (2 ore)

Conţinut curs - II

• Probleme de satisfacere a constrângerilor (CSP). Limbaje de modelare a problemelor şi de specificare a constrân gerilor. (4 ore)

• Căutarea unei soluţii pe domenii finite (generate&test, standard backtracking, forward checking, lookahead, partial lookahead). Soluţii optimale (forme de căutare branch and bound) (4 ore)

• Programare logică cu constrângeri (4 ore)

Bibliografie

◇ I

• B. Buchberger, G. Collins, R. Loos - Computer algebra and Symbolic Computation, Springer Verlag, 1983 N. C. Heinze, J. Jaffar, C. Michailov, P. J. Stuckey,

• R. H. C. Yap - The CLP($\mathcal{R}$) Programmers Manual, version 1.2, Dept. of Computer Science, Monash University, 1992.

• P. Van Hentenryck - Constraint Satisfaction in Logic Programming, M.I.T. 1989

• T. Muller, K. Popov,C. Schulte, J. Wurtz - Constraint Programming in Oz, DFKI Oz Documentation Series, 1995

• C. Muscalagiu - Introducere in programarea logica si limbajele de programare logica, Ed. Univ. "A.I.Cuza" Iasi, 1996

• I. Bratko - PROLOG. Programming for Artificial Intelligence, Addison Wesley, third edition, 2001

Bibliografie

◇ II

• St. Trausanu-Matu - Programare in LISP. Inteligenta artificiala si web semantic, Ed. POLIROM, 2004

• P. Graham - ANSI Common Lisp, Prentice Hall, 1996

• P. Graham - On Lisp. Advanced techniques for Common Lisp, Prentice Hall, 1994.

• D. Friedman, M. Wand, C. Hayes - Essentials of Programming Languages, second edition, MIT, 2001.

• Grillmeyer - Exploring Computer Science with Scheme, Springer, 1997.

• B. Harvey, M. Wright - Simply Scheme. Introducing Computer Science, second edition, MIT, 1999.

Bibliografie

◇ III

• H. Abelson, G.J. Sussman, J. Sussman - Structure and Interpretation of Computer Programs, Second edition, MIT, 1996.

• C. Queinnec - Les Langages Lisp, InterEditions, 1994.

• http://pauillac.inria.fr/ diaz/gnu-prolog/

• http://www.prologia.fr/

• http://web.info.uvt.ro/ idramnesc/

• http://web.info.uvt.ro/čizbasa/lisp/

• http://web.info.uvt.ro/čizbasa/prolog/

Caracterizare Lisp

◇ Standard: peste 700 de funcţii

- uniformitatea (programe, date - expresii simbolice)
- extensibilitatea (limbaj de programare programabil))
- flexibil (dezvoltare stiluri diferite de programare))
- abstratizare; recursivitate
- programare funcţională
- structuri dinamice de date; alocare dinamica a memoriei
- interpretor / compilator
- programare bottom up (software reutilizabil; prototipare rapida).

Exemple

◇ Funcţie ce întoarce o funcţie

```
(defun adunan (n) #' (lambda (n) (+ x n)))
```

◇ Factorial din n, funcţie recursiva

```
(defun !(n) (if (zerop n) 1 (* n (! (1- n)))))
```

Lambda calcul

◇ Definirea funcţiilor

```
f(x,y)=x^2+y-2 este reprezentată prin:

(f (x, y) x^2+x-2)

unde x şi y sunt parametrii funcţiei,
iar x^2+y-2 este corpul funcţiei
```

◇ Expresii condiţionale

◇ Funcţii recursive

Structuri dinamice de date

◇ Liste

- pointeri (adrese) - reprezentare implicită
- operaţii cu liste
- car, cdr, ...
- gestiune automată a memoriei

Interpretorul Lisp

◇ Interfaţă (front-end) interactivă numită *top-level*

• *prompter Lisp* specific versiunii Common Lisp utilizate (`>`, `:`, `*` etc)

◇ *Ciclu de bază*: `read-eval-print` (top-level loop), format din trei etape sau stări:

• `read`: citeşte o expresie simbolică;

• `eval`: evaluează expresia simbolică introdusă;

• `print`: afişează rezultatul obţinut în urma evaluării expresiei.

Elemente de bază

◇ Atomii

• Numerici (numere)

• Simbolici (simboluri)

◇ Numerele - se evaluează la ele însele

```
>16                    >1.25
16                     1.25
```

◇ Simboluri

• Fie `suma` ⟶ `16`); `locul-nasterii` ⟶ `ARAD`):

```
>suma                  >locul-nasterii
16                     arad
```

Elemente de bază

◇ Simboluri

• Încercarea de evaluare a unui simbol ce nu a fost legat la o valoare produce eroare:

```
>a
error: unbound variable - a
```

◇ Un şir de caractere se evaluează la el însăşi:

```
>"Sir de caractere"
"Sir de caractere"
```

Elemente de bază

◇ Liste

• O listă constă din zero sau mai multe elemente (atomi sau liste), separate prin spaţii şi cuprinse între paranteze rotunde.

• Exemple:

```
(), (a b c), (a (b (c))),
(+ 1 2 3), (aceasta este o lista)
```

• Constantele `t` - adevărat (true) şi `nil` - fals (false); `nil` - reprezinta şi lista vida: ()

```
>t                  >nil                >()
t                   nil                 nil
```

Elemente de bază

◇ Comentariile în Lisp sunt de forma:

```
;<text-oarecare>
```

unde ; este un macrocaracter.

• Funcţiile în Lisp se scriu indentate, în general ţinând cont de paranteze:

```
;;; Calculul factorialului
;;;
(defun fact (n)
   ;;conditia de oprire
   (if (zerop n)
        1     ; 1 <-- (fact 0)
        ;;apelul recursiv
        (* n (fact (1- n)))
   )
)
```

Elemente de bază

◇ Expresii simbolice. Listele şi atomii formează expresiile simbolice sau s-expresiile în Lisp.

◇ O definiţie (recursivă) a expresiilor simbolice este următoarea:

• Atomii sunt expresii simbolice;

• O listă este o construcţie de forma () sau $(e_1, e_2, \ldots, e_n)$, unde $n \geq 1$ şi $e_1, e_2, \ldots, e_n$ sunt expresii simbolice;

• O pereche cu punct este o construcţie de forma $(e_1 \ . \ e_2)$, unde e_1 şi e_2 sunt expresii simbolice;

• Listele şi perechile cu punct sunt expresii simbolice.

• Exemple:

```
1, abc (), (a . b), ((a) (b c (d)) e),
"ab1", ("a" 2 b)
```

Elemente de bază

◇ Funcţii - obiecte, entităţi Lisp

• Apelul unei funcţii este reprezentat de o listă în care primul element reprezintă funcţia, iar celelalte elemente argumentele funcţiei.

• Exemplu:

```
>(+ 1 2 3)
6
>(* (+ 2 3) 10)
50
```

```
>(sqrt 4)
2
```

• **Formă** - o expresie simbolică ce poate fi evaluată

• evaluarea unei liste care nu este o formă va produce eroare:

```
>(a 1 2)
error: unbound function - a
```

Elemente de bază

◇ **Evaluarea**. Evaluarea implicită are loc în cadrul ciclului `read-eval-print`, în faza `eval`. Funcţia de evaluare acţionează astfel:

• dacă expresia este atom, întoarce valoarea sa;

• dacă expresia este o listă:

◦ dacă primul element din listă reprezintă o funcţie: 1) regăseşte această funcţie; 2) evaluează restul elementelor (argumentele funcţiei) din listă aplicând aceleaşi reguli la fiecare din ele; 3) aplică funcţia la argumente şi întoarce rezultatul.

◦ daca primul element reprezintă o formă specială, aplică un tratament specific asupra argumentelor sale şi asupra formei speciale;

◦ dacă primul element reprezintă un macro, aplică un tratament specific macrourilor.

Elemente de bază

◇ Stoparea evaluării

• Funcţia `(quote <arg>)`, unde `arg` este o s-expr

• `(quote <arg>)` ≡ `'<arg>`; unde `'` - macrocaracter

```
>(quote a)          >(quote (a b c))      >'a
a                   (a b c)               a
```

◇ Funcţia eval

```
>(eval '(+ 1 2))          >(car '(+ 1 2))
3                         +
```

• Fie `y` ⟶ `x`, `x` ⟶ `10`:

```
>(eval y) ;y (ca argument)
          ;este evaluat la x, iar x la 10
10
```

Elemente de bază

◇ Legarea varibilelor

• Datele cu care operăm sunt expresii simbolice (atomi, liste)

• Datele ocupă locaţii de memorie

• Forma de reprezentare şi conţinutul locaţiei de memorie depind de tipul datelor

• Asocierea unei variabile la o dată se numeşte *legare*

• Locaţia de memorie conţine amprenta tipului datei, tip ce se atribuie variabilei în momentul legării variabilei la valoarea din locaţia de memorie

• Atribuirea tipului in timpul execuţiei (evaluarii)

Elemente de bază

◇ `setq, set, psetq, pset`

• `Setq` este formă specială şi are forma generală:

`(setq` <var$_1$ > <val$_1$ > ... <var$_n$ > <val$_n$ >)

Exemple:

```
>(setq x 1 y '(a b c))
(a b c)
>x
1
>y
(a b c)
```

```
>(setq x 1 y (+ x 2))
 3
>x
1
>y
3
```

• `Set` este funcţie şi are forma generală:

`(set` <var$_1$ > <val$_1$ > ... <var$_n$ > <val$_n$ >)

Elemente de bază

◇ `setq, set, psetq, pset` (cont.)

```
>(set 'x 1 'y 2)
2
>x
1
>y
2
```

```
>(setq y 'x)
  >(setq x 'a)
  >(set y 2)
  >y
  x
  >x
  2
```

• Legare secvenţială, legare în paralel

```
>(setq x 10)
10
>(setq x 1 y (+ x 2))
3
>x
1
>y
3
```

```
>(setq x 10)
10
>(psetq x 1 y (+ x 2))
12
>x
1
>y
12
```

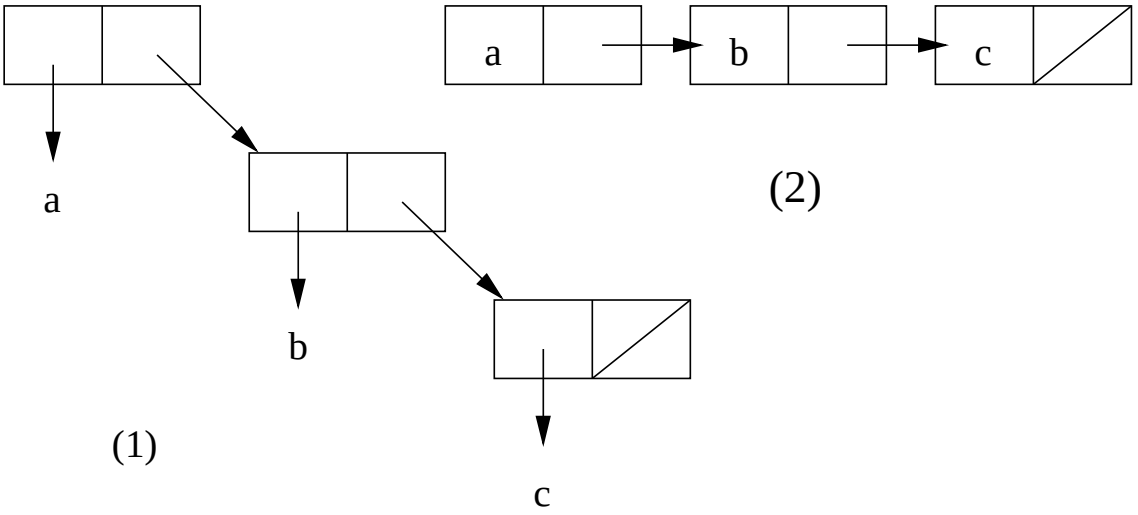
Elemente de bază

◇ Operaţii cu liste

• Elementele listei: atomi, liste

• Prelucrare: nivel superficial / adâncime

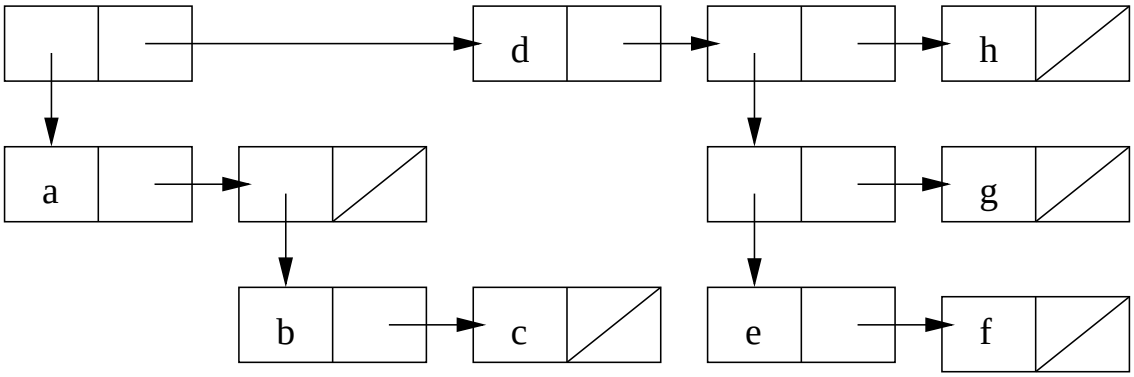
Structura internă a listei (a b c): (1) Descrierea arborescenta; (2) Descrierea simplificată



Elemente de bază

◇ Operaţii cu liste

Structura internă a listei ((a (b c)) d ((e f) g) h), descrierea pe nivele



Elemente de bază

◇ Operaţii cu liste

• Funcţiile `car, cdr` şi `cons`

```
(car <lista>)
```

Exemple de utilizare:

```
>(car '(a b c))
a
>(car '(1 . 2))
1
>(car '((a (b c)) d ((e f) g) h))
(a (b c))
```

```
>(car 'a)
error: bad argument type - a
>(first '(a b c))
 a
```

Elemente de bază

◇ Operaţii cu liste

• Funcţiile `car, cdr` şi `cons`

```
(cdr <lista>)
```

Exemple de utilizare:

```
>(cdr '(a b c))          >(cdr 'a)
(b c)              error: bad argument type - a
>(cdr '(1 (2 . 3)))     >(rest '(a b c))
((2 . 3))                        (b c)
>(dar '((a (b c)) d ((e f) g) h))
(d ((e f) g) h)
```

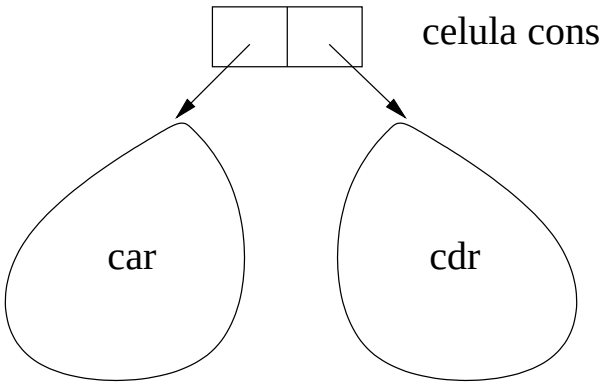
Operaţii cu liste

◇ Operaţii cu liste

• Funcţiile `car, cdr` şi `cons`

```
(cons <el> <lista>)
```

Structura unei celule cons



Elemente de bază

◇ Operaţii cu liste

• Funcţiile `car, cdr` şi `cons`

Exemple de utilizare:

```
>(cons 'a '(b c))
(a b c)
>(cons '(a) '(b c))
((a) b c)
>(cons 1 nil)
(1)
```

```
>(cons nil nil)
  nil
>(cons 1 2)
  (1 . 2)
  >(cons '(a b) 'c)
  ((a b) . c)
```

Elemente de bază

◇ Funcţiile `car, cdr` şi `cons`

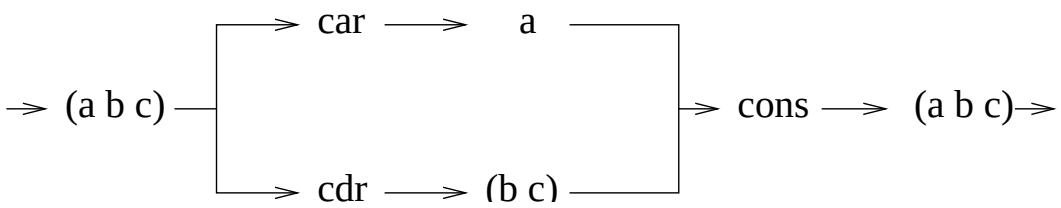
Funcţiile `car` şi `cdr` pot fi compuse:

```
>(car (car (cdr '(a ( b c) d))))
b

>(caadr '(a (b c) d))
b

>(setq l '(a b c))
(a b c)
>(cons (car l) (cdr l))
(a b c)
```

Relaţia dintre car, cdr şi cons



Elemente de bază

◇ Alte funcţii. `append, list, reverse, last` şi `length`

`(append` <`lista`$_1$ > <`lista`$_2$ > ... <`lista`$_n$ >`)`

Exemple de utilizare:

```
>(append '(a) '(b c))
(a b c)
>(append '((a) b) '(c) '(d (e f)))
((a) b c d (e f))
>(append 'a '(b c))
error: bad argument type - a
>(append '((a)) '(b c) 'd)
((a) b c . d)
>(append)
nil
```

Elemente de bază

◇ Alte funcţii. `append, list, reverse, last` şi `length`

`(list` <`sexpr`$_1$ > <`sexpr`$_2$ > ... <`sexpr`$_n$ >`)`

Exemple de utilizare:

```
>(list 1 2 3)
(1 2 3)
>(list '(a b) 'c '((d e) f))
((a b) c ((d e) f))
>(list 1 '(2 . 3))
(1 (2 . 3))
>(list nil nil)
(nil nil)
```

Elemente de bază

◇ Alte funcţii. `append, list, reverse, last` şi `length`

Comparaţie între `cons, append` şi `list`:

```
>(cons '(a) '(b c))
((a) b c)
>(append '(a) '(b c))
(a b c)
>(list '(a) '(b c))
((a) (b c))
```

Forma generală a funcţiei `last` este:

```
(last <lista>)
```

```
>(last '(a b c d))
(d)
>(last '(a b . c))
(b . c)
>(last '(a))
(a)
```

Elemente de bază

◇ Alte funcţii. `append, list, reverse, last` şi `length`

Forma generală a funcţiei `reverse`:

```
(reverse <lista>)

>(reverse '(1 2 3 4 5)
(5 4 3 2 1)
>(reverse '(a (b c d) e))
(e (b c d) a)
```

Forma generală a funcţiei `length`:

```
(length <lista>)

>(length '(a b c))
3
>(length '((a b (c)) (d e)))
2
>(length ())
0
```

Definirea funcţilor

◇ Funcţii sistem / utilizator

Definirea unei funcţii:

```
(defun <nume-func> <lista-param>
         <expr-1> <expr-2> ... <expr-n>)
```

unde:

• <`nume-func`> este primul argument şi reprezintă numele funcţiei definite de **`defun`**;

• <`lista-param`> este al doilea argument al lui **`defun`**, are forma (<`par-1`> <`par-2`> ... <`par-m`>) şi reprezintă lista cu parametri pentru funcţia definită;

• <`expr-i`>, $i = 1, \ldots, n$ sunt forme ce alcătuiesc corpul funcţiei definite.

Definirea funcţiilor

◇ Exemple de utilizare:

```
>(defun patrat (x)      ; patratul unui număr
   (* x x))
patrat

>(defun calcul (x y z) ; calculează val. unei expr.
   (+ x (* y z))
calcul
```

◇ Apelul unei funcţii

```
(<nume-func> <arg-1> <arg-2> ... <arg-n>)
```

```
>(patrat 2)
4
>(setq y 3)
3
>(patrat y)
9
```

```
>(calcul 2 3 4)
14
>(setq x 3 y 2 z 4)
4
>(calcul x y z)
11
```

Definirea funcţiilor

◇ Evaluarea

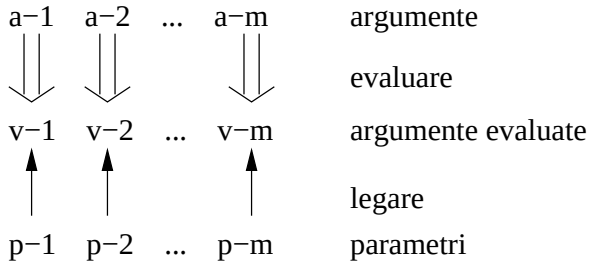
• Se identifică funcţia;

• Se evaluează argumentele;

• Parametrii formali sunt legaţi la argumentele evaluate. Dacă înainte de apel parametrii au fost legaţi, valorile acestora se salvează, urmând a se restaura după revenirea din funcţie; Un parametru nelegat înainte de apelul funcţiei, redevine nelegat după revenirea din funcţie.

• se evaluează corpul funcţiei;

• valoarea întoarsă este dată de valoarea ultimei expresii simbolice din corpul funcţiei.



Definirea funcţiilor

◇ Variabile legate, variabile libere

```
>(setq x 1 y 2)
>(defun f1 (x)
    (+ x y))
>(f1 3)
5
>x
1
>y
2
```

```
>(setq x 1 y 2)
>(defun f2 (x)
    (setq x 10)
    (+ x y)
>(f2 x)
12
>x
1
```

```
>(setq x 1 y 2)
>(defun f3 (x)
    (setq x 10 y 20)
    (+ x y))
>(f3 x)
30
>y
20
```

```
>(setq x 1 y 2)
>(defun f4 (x)
  (setq x 10)
(+ (symbol-value 'x) y)
>(f4 x)
3
>x
1
```

Expresii

◇ Expresii artimetice, relaţionale, logice

◇ Expresii condiţionale

◇ Predicate Lisp

◇ Funcţii de ramificare (`if, cond, case`)

Expresii

- Expresii aritmetice

```
>(setq x 4)
>(* 2 (+ 3 (sqrt x)))
>6
```

- Expresii relaţionale

```
>(setq x 10)
>(>= x 2)
t
>(string= "a" "bcd")
nil
```

- Expresii logice

```
>(setq x 10 y 5)
>(and (> x 5) (< y 10))
t
```

Testul de egalitate

◇ Se cunoaşte tipul obiectelor

=, char=, string=, /=, string/=

◇ Nu se cunoaşte tipul obiectelor

• Identitate structurală

```
>(setq x '(a b c))
>(setq y (cdr x))
>(setq z '(b c))
>(eq y (cdr x))
t
>(eq y z)
nil
```

• Izomorfism structural

```
>(equal y z)
t
```

Expresii condiţionale

◇ Expresii simbolice ce returnează `t` sau `nil`

• Extensie la `nil` sau nonnil (diferit de nil)

• if, cond, case, when, unless, do, do*, ...

Predicate lisp

◇ Funcţii ce returnează `t` sau `nil`

• numberp, symbolp, atom, consp, listp

Funcţii de ramificare

- if

```
>(setq x 10)
>(if (> x 15) 1 2)
2
>(if (
```

- cond

```
(cond <clauza-1>
      <clauza-2>
      ...
      <clauza-n>)
unde <clauza-i>::=(<ec> <e> <e> ... <e>)
```

Cond

- Exemple

```
>(defun det-tip (e)
   (cond ((numberp e) (cons e '(este numar)))
         ((symbolp e) (cons e '(este simbol)))
         ((lisp e) (cons e '(este lista)))
         (t (cons e '(tip necunoscut)))))
```

```
>;;; (and <ob1> <ob2>)
>(cond (<ob1> <ob2>))
>;;; (or <ob1> <ob2>)
>(cond (<ob1>)
       (<ob2>))
```

Conclusions

- ◇ ●
- ●

Recursivitate

◇ Un obiect este recursiv daca este definit funcţie de el însuşi.

• definim un număr infinit de obiecte printr-o declaraţie finită

• functii recursive, proceduri recursive, definiţii recursive de date, calcul recursiv.

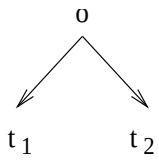
Recursivitate

◇ Exemple

• GNU:
GNU = Gnu is Not Unix

• numerele naturale:
0 este număr natural;
succesorul unui număr natural este un număr natural.

• arborii binari:
o este un arbore binar (arborele vid);
dacă t_1 şi t_2 sunt arbori binari atunci şi



este un arbore binar.

Recursivitate

◇ Definiţie bazată pe inducţie structurală (structura programului trebuie să reflecte structura datelor) - definim tipul de date `listă-de-numere`

lista vidă este o `listă-de-numere`;
dacă l este o `listă-de-numere` şi n este un număr, atunci perechea `(n . l)` este o `listă-de-numere`.

În notatţia BNF avem următoarele reguli:

```
<listă-de-numere>::=()
<listă-de-numere>::=(<număr> . <listă-de-numere>)
```

sau utilizând simbolul bară verticală din BNF:

```
<listă-de-numere>::=() | (<număr> . <listă-de-numere>)
```

sau utilizând asteriscul (Kleen star):

```
<listă-de-numere>::=({<număr>}*)
```

Recursivitate

◇ Exemple

• factorialul unui număr:

$$n! = \begin{cases} n * (n-1)! & \text{dacă } n > 0 \\ 1 & \text{dacă } n = 0 \end{cases}$$

• numarul de elemente (pe nivel superficial) dintr-o listă:

$$\text{nr-elem}(l) = \begin{cases} 1 + \text{nr-elem}(l \setminus \text{primul-elem}) & \text{dacă } l \neq nil \\ 0 & \text{dacă } l = nil \end{cases}$$

Funcţii recursive

◇ Caracterizare

- abstractizare, corectitudine
- pointeri → date recursive
- reducere efecte laterale
- lizibilitate
- iterativ ↔ recursiv

Funcţii recursive

$\diamond$ O funcţie ce se autoapelează este o funcţie recursivă

• funcţii *direct recursive* (f apelează f)

• funcţii *indirect recursive* (f apelează g_1, g_1 apelează g_2, ..., g_k apelează f)

• funcţii *mutual recursive* ($k = 1$)

Funcţii recursive

◇ Calculul factorialului

• factorialul unui număr:

$$\text{fact}(n) = \begin{cases} 1 & \text{dacă } n = 0 \\ n * \text{fact}(n-1) & \text{dacă } n > 0 \end{cases}$$

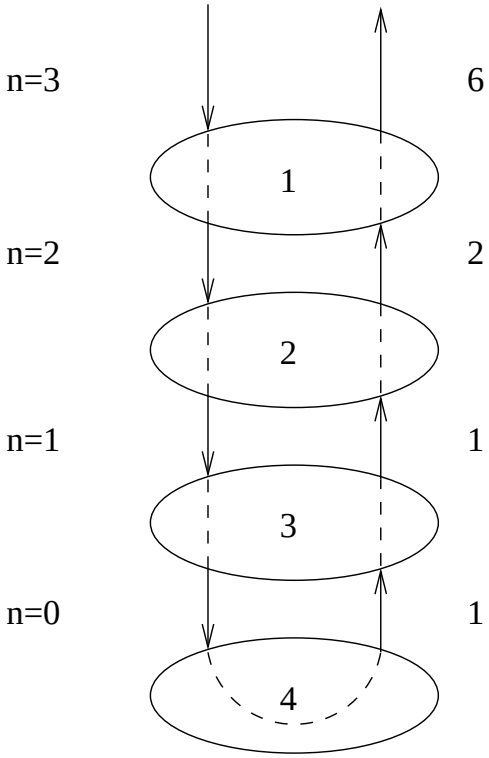
```
(defun fact (n)
 (cond ((zerop n) 1) ; condiţia de terminare
       (t (* n (fact (1- n)))) )) ; apelul recursiv
```

sau

```
(defun fact (n)
  (if (zerop n)
     1 ; condiţia de terminare
     (* n (fact (1- n))) )  ; apelul recursiv
```

Calculul factorialului

◇ Arborele inversat (`>(fact 3)`):




```
(* 3 (fact 2))                ;produs suspendat
     (2 * (fact 1))           ;produs suspendat
          (* 1 (fact 0))      ;produs suspendat
                  1
```

Trasarea funcţiilor (recursive)

◇ Funcţii de depanare (trasare, evaluare pas cu pas, definire puncte de intrerupere etc)

Rezultatul trasării funcţiei `fact`:

```
>(trace fact)          ; activare trasare
(FACT)
>(fact 3)
 0: (FACT 3)           ; -->
   1: (FACT 2)         ; -->
     2: (FACT 1)       ; -->
       3: (FACT 0)     ; -->
       3: returned 1   ; <--
     2: returned 1     ; <--
   1: returned 2       ; <--
 0: returned 6         ; <--
6
>(untrace fact)        ; dezactivare trasare
nil
```

Recursivitate

◇ Corectitudinea unui algoritm

• funcţionează corect în toate cazurile

• demonstraţie prin inducţie

◇ Conceperea unui algoritm recursiv

• definirea problemei prin descompunerea într-un numar finit de probeme tot mai mici (clauză / clauze recursive)

• gasirea modului de rezolvare a celei mai mici versiuni a problemei printr-un numar finit de operaţii (condiţie / condiţii de terminare)

Funcţii recursive

◇ Reguli de scriere a funcţiilor recursive în Lisp

• `if`, `cond`

• clauzele recursive vor fi precedate de clauzele / condiţiile de terminare

• folosirea greşită a condiţiilor de terminare conduce la ciclări la infinit / depăşirea memoriei disponibile

Exemplu:

```
(defun our-member (el l)
  (cond ((equal (carl l)) el)
        (t (our-member (cdr l))) ))
```

Obs.: trebuie inserată la început în `cond` clauza `((endp l) nil)`.

Recursivitate în Lisp

◇ Recursivitate simpla,

• la fiecare apel se crează o copie a variabilelor locale

• parcurgere pe nivel superficial în cazul listelor (parcurgere doar a subarborelui drept)

◇ Recursivitate arborescentă,

• la fiecare apel se crează mai multe copii ale variabilelor locale

◇ Recursivitate dublă

• se crează două copii ale variabilelor locale

• parcurgere în profunzime în cazul listelor (parcurgere atât a subarborelui stâng, cât şi a subarborelui drept))

Recursivitate simplă

◇ lungimea unei liste (pe nivelul superficial)

$$\text{our\_length}(l) = \begin{cases} 0 & \text{dacă } l = nil \\ 1 + \text{our\_length}((cdr\, l)) & \text{altfel} \end{cases}$$

```
(defun our_length (l)
  (if (endp l) 0
              (+ 1 (our_lenght (cdr l)))) )
```

Recursivitate dublă

◇ Determinarea numerelor lui Fibonacci

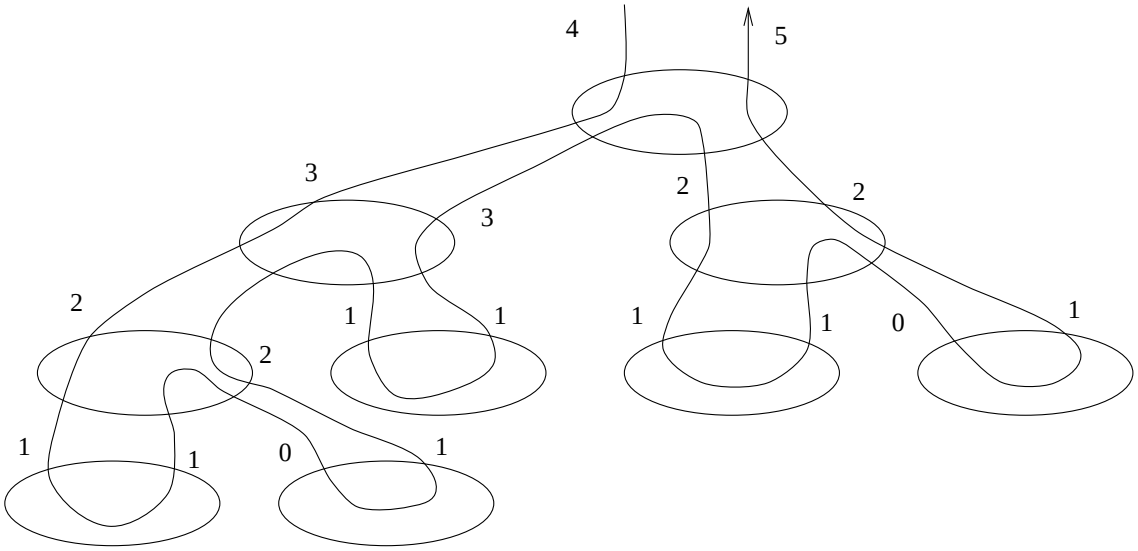
$$fib(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ 1 & \text{dacă } n = 1 \\ fib(n-1) * fib(n-2) & \text{dacă } n > 1 \end{cases}$$

Funcţia Lisp corespunzătoare este o funcţie dublu recursivă:

```
(defun fib (n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (t (+ (fib (- n 1)) (fib (- n 2)))) ))
```

Recursivitate dublă

◇ În urma apelului `>(fib 4)` rezultatul este `5`, arborele inversat fiind:



Trasarea funcţiei fib

```
>(trace fib)
(FIB)
>(fib 4)
 0: (FIB 4)
   1: (FIB 3)
     2: (FIB 2)
       3: (FIB 1)
       3: returned 1
       3: (FIB 0)
       3: returned 1
     2: returned 2
     2: (FIB 1)
     2: returned 1
   1: returned 3
   1: (FIB 2)
     2: (FIB 1)
     2: returned 1
     2: (FIB 0)
     2: returned 1
   1: returned 2
 0: returned 5
5
```

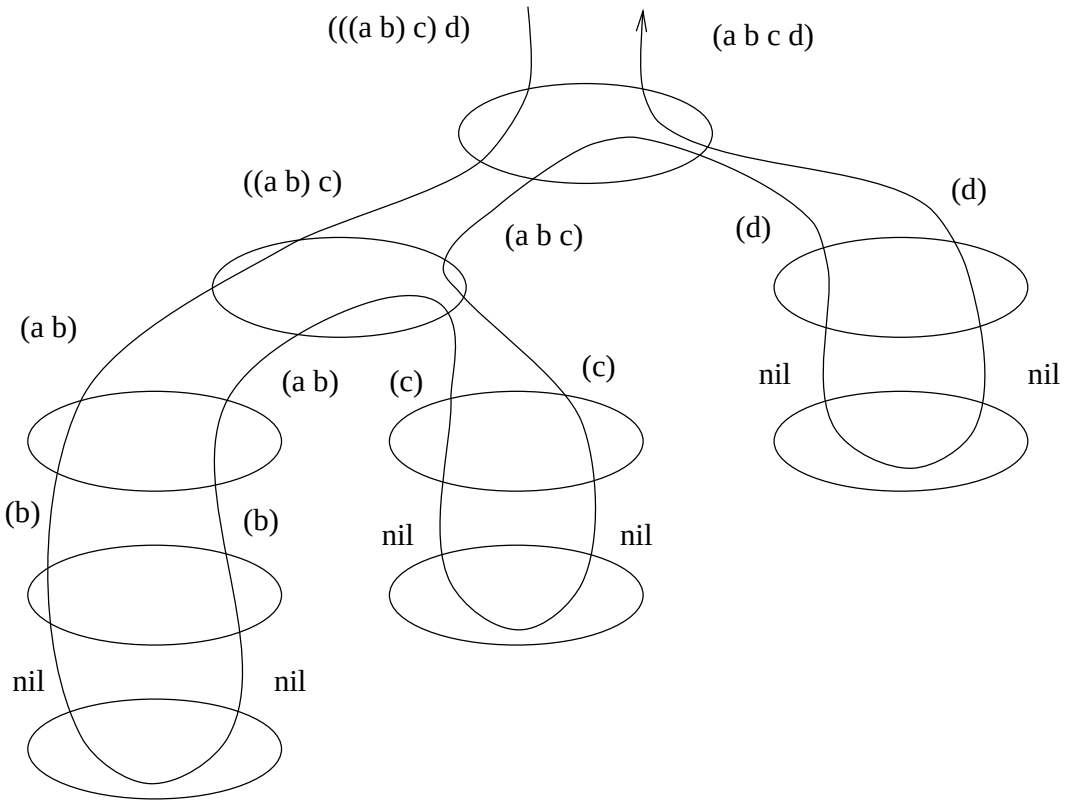
Recursivitate simplă şi dublă

◇ Atomizarea unei liste

```
(defun atomizare (l)
  (cond ((endp l) nil)
        ; recursivitate simplă
        ((atom (car l)) (cons (car l)
                              (atomizare (cdr l))))
        ; recursivitate dublă
        (t (append (atomizare (car l))
                   (atomizare (cdr l)))) ))
```

Recursivitate simplă şi dublă

◇ În urma apelului `>(atomizare ((a b) c) d))` rezultatul întors este `(a b c d)`. Arborele inversat corespunzător este:



Funcţii final recursive

◇ Caz particular de recursivitate (tail recursion / tail-end recursion / recursivitate la capăt / recursivitate finală)

• O funcţie este final-recursivă dacă valoarea obţinută pe ultimul nivel de recursivitate rămâne neschimbată până la revenirea pe nivelul de sus.

• apelurile recursive nu sunt argumente pentru alte funcţii şi nu sunt utilizate ca şi teste (apelul recursiv este ultima operaţie ce apare într-o funcţie).

• la o funcţie ce nu este final recursivă se poate observa că apelul recursiv este conţinut într-un apel de funcţie (+, −, cons, append etc).

• O funcţie final-recursivă se bucură de proprietatea că poate fi tradusă automat într-o funcţie iterativă.

• nefinal recursiv → final recursiv: tehnica variabilelor colectoare

Funcţii final recursive

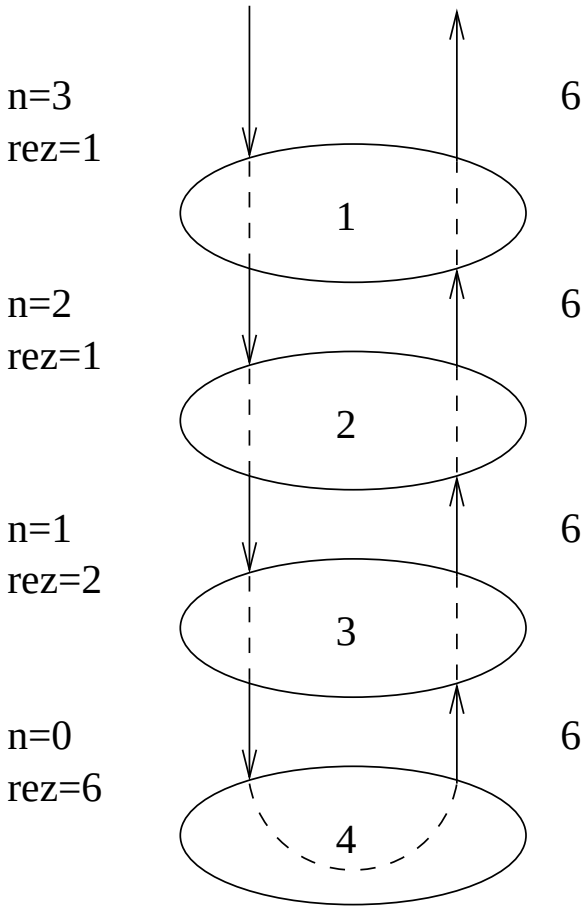
◇ Funcţia `fact`:

```
(defun fact (n)
  (fact-aux n 1))          ; rez = 1
(defun fact-aux (n rez)    ; funcţie auxiliară
  (if (zerop n) rez        ; rezultatul final = rez
        (fact-aux (1- n)
           (* n rez))) ))  ; rez = n*rez
```

```
> (trace fact fact-aux)
(fact-aux fact)
> (fact 3)
Entering: FACT, Argument list: (3)
 Entering: FACT-AUX, Argument list: (3 1)
  Entering: FACT-AUX, Argument list: (2 3)
   Entering: FACT-AUX, Argument list: (1 6)
    Entering: FACT-AUX, Argument list: (0 6)
    Exiting: FACT-AUX, Value: 6
   Exiting: FACT-AUX, Value: 6
  Exiting: FACT-AUX, Value: 6
 Exiting: FACT-AUX, Value: 6
Exiting: FACT, Value: 6 6
```

Funcţii final recursive

◇ Arborele inversat



Funcţii final recursive

◇ Varianta final-recursivă pentru determinarea celui de-al n-lea număr din şirul lui Fibonacci este:

```
(defun fib1 (n)
  (cond ((< n 2) 1)
        (t (fib1-aux
                1 ; f1 - penultimul număr calculat
                1 ; f2 - ultimul număr calculat
                2 ; i  - indexul pentru numărul
                  ;    curent de calculat
                n)) ))

(defun fib1-aux (f1 f2 i n)
  (cond ((> i n) f2)
        (t (fib1-aux f2        ; f2 -> f1
                     (+ f1 f2) ; (+ f1 f2) -> f2
                     (1+ i)    ; (1+ i) -> i
                     n)) ))
```

Recursivitate

◇ Comparaţie dublu ↔ simplu recursiv

• Cazul nefinal recursiv: complexitatea este exponenţială (apelul `>(fib 100)` necesită mai mult de 10^{20} apeluri de funcţie;

• cazul final recursiv: complexitatea este polinomială (`>(fib1 100)` întoarce valoarea $\approx 5.73 * 10^{20}$, valoare ce se obţine după circa 100 de apeluri de funcţie).

Recursivitate

◇ •

◇ Recursivitate compusă

• În cazul în care în cadrul argumentelor unei funcţii recursive există apeluri recursive spunem că avem *recursivitate compusă*.

• Exemplu: funcţia lui Ackermann

```
A<I, 0>=I+1
A<0,J>=A<1,J-1>
A<i,j>=A<A<I-1,J>,J-1>
```

Valorile si numărul operaţiilor cresc foarte repede: `A<0,1>=2`, `A<1,2>=5`, `A<2,3>=29`, `A<3,4>=`2^{65536}, ...

Recursivitate

◇ Recursivitate monotona / nemomotonă

• *Recursivitatea monotonă* (recursivitate structurală) este recursivitatea în care modificarările asupra argumentelor din apelurile recursive se fac tot timpul în aceeaşi direcţie.

• *Recursivitatea nemonotonă* este cea în care modificările asupra argumentelor sunt nemonotone (nu tot timpul în aceeaşi direcţie).

• De exemplu, funcţia recursivă ce implementează metoda lui Newton pentru găsirea zerourilor unei funcţii este nenonotonă.

Recursivitate nemomotonă

• Metoda lui Newton pentru găsirea unei soluţii pentru $f(x) = 0$ se bazează pe formula iterativă urmatoare:

$$x_{k+1} = x_k - \frac{f(x_k)}{Df(x_k)}$$

În continuare este prezentat programul Lisp pentru $f(x) = x^3 - 1$:

```
(defun f (x)
     (- (* x x x) 1) )
(defun df (x)
     (* 3 x x) )
(defun newx (x)
     (- x (/ (f x) (df x))) )
(defun newton (x)
     (cond ((< (abs (f x)) 0.00001) x)
           (t (newton (newx x))) ))
```

Recursivitate nemonotonă

Trasarea funcţiei **newton** (cu argumentul funcţiei ce nu se modifică în acelaşi sens):

```
>(newton -1.0)
 0: (NEWTON -1.0)
   1: (NEWTON -0.3333333)
     2: (NEWTON 2.7777781)
       3: (NEWTON 1.895052)
         4: (NEWTON 1.3561869)
           5: (NEWTON 1.0853586)
             6: (NEWTON 1.0065371)
               7: (NEWTON 1.0000424)
                 8: (NEWTON 1.0)
                 8: returned 1.0
               7: returned 1.0
             6: returned 1.0
           5: returned 1.0
         4: returned 1.0
       3: returned 1.0
     2: returned 1.0
   1: returned 1.0
 0: returned 1.0
1.0
```

Recursivitate

◇ Parcurgerea şi construire liste

• Parcurgerea pe nivel superficial: se crează o copie a unei liste (copierea doar a nivelului superficial)

```
(defun our-copy-list (l)
  (if (atom l) l
              (cons (car l)
                    (our-copy-list (cdr l)))))
```

• Parcurgerea în adâncime: se crează o copie a arborelui binar (copiere a listei pe toate nivelurile).

```
(defun our-copy-tree (l)
  (if (atom l) l
              (cons (our-copy-tree (car l))
                    (our-copy-tree (cdr l)))))
```

Recursivitate

◇ Parcurgere şi calcul

• Sabloane de funcţii de calcul şi / sau parcurgere

```
(defun <calcul> (n)
  (if <cond-oprire>    ; test-zero
      <număr-neutru>  ; în raport cu oper
      (<oper> n (calcul (1- n)))))
```

Exemple: suma primelor n numere naturale/ factorialul / m la puterea n

```
(defun <calcul> (lista)
  (if <cond-oprire>    ; test-sfarşit listă
      <număr-neutru>  ; în raport cu oper
      (<oper> <element-curent> (calcul <rest-listă>))))
```

Exemple: Suma / produsul elementelor unei liste; construirea unei liste

Recursivitate

◇ Probleme

- Operatii cu liste (pe nivel superficial şi în adâncime)
- Operaţii cu mulţimi
- Operaţii cu vectori (rari)
- Operaţii cu matrice (rare)
- Operaţii cu polinoame
- Operaţii cu expresii generalizate
- Operaţii asupra arborilor binari
- etc.