

# CURS 1

## Introducere în programarea declarativă. Recursivitate

### Cuprins

Bibliografie .....	1
1. Programare și limbaje de programare .....	1
2. Recursivitate .....	2
2.1 Exemple recursivitate.....	3

### Bibliografie

Capitolul 1, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

## 1. Programare și limbaje de programare

### ➤ Limbaje

- **Procedurale (imperative) – limbaje de nivel înalt**

- Fortran, Cobol, Algol, Pascal, C,...
  - program – secvență de instrucțiuni
  - instrucțiunea de atribuire, structuri de control – pentru controlul execuției secvențiale, ramificării și ciclării.
  - rolul programatorului – “ce” și “cum”
    - 1. să descrie **CE** e de calculat
    - 2. să organizeze calculul
    - 3. să organizeze gestionarea memoriei
- CUM
- !!! se susține că instrucțiunea de atribuire este periculoasă în limbajele de nivel înalt, așa cum instrucțiunea GO TO a fost considerată periculoasă pentru programarea structurată în anii '68.

- **Declarative (descriptive, aplicative) – limbaje de nivel foarte înalt**

- se bazează pe expresii
- expresive, ușor de înțeles (au o bază simplă), extensibile

- programele pot fi văzute ca descrieri care declară informații despre valori, mai degrabă decât instrucțiuni pentru determinarea valorilor sau efectelor.
  - renunță la instrucțiuni
    1. protejează utilizatorii de la a face prea multe erori
    2. sunt generate din principii matematice - analiza, proiectarea, specificarea, implementarea, abstractizarea și raționarea (deducții ale consecințelor și proprietăților) devin activități din ce în ce mai formale.
  - rolul programatorului - “ce” (nu “cum”)
  - două clase de limbaje declarative
    1. **limbajele funcționale** (de exemplu Lisp, ML, Scheme, Haskell, Erlang)
      - se focalizează pe valori ale datelor descrise prin expresii (construite prin aplicări ale funcțiilor și definiții de funcții), cu evaluare automată a expresiilor
    2. **limbaje logice** (de exemplu Prolog, Datalog, Parlog), care se focalizează pe aserțiuni logice care descriu relațiile dintre valorile datelor și derivări automate de răspunsuri la întrebări, plecând de la aceste aserțiuni.
  - aplicații în Inteligența Artificială – demonstrarea automată, procesarea limbajului natural și înțelegerea vorbirii, sisteme expert, învățare automată, agenți, etc.
- 
- Limbaje multiparadigmă: **F#, Python, Scala** (imperativ, funcțional, orientat obiect)
  - Interacțiuni între limbajele declarative și cele imperative – limbaje declarative care oferă interfețe cu limbaje imperative (ex C, Java): SWI-Prolog, GNUProlog, etc.
  - **Logtalk** – integrează logica și OOP.
  - Programare logică în Python
    - **Karen**
    - **SymPy** – bibliotecă pentru calcul simbolic

## 2. Recursivitate

- mecanism general de elaborare a programelor.
- recursivitatea a apărut din necesități practice (transcrierea directă a formulelor matematice recursive; vezi funcția lui Ackermann)
- recursivitatea este acel mecanism prin care un subprogram (funcție, procedură) se autoapelează.
  - două tipuri de recursivitate: **directă** sau **indirectă**.
- **!!! Rezultat**

- orice funcție calculabilă poate fi exprimată (deci și programată) în termeni de funcții recursive
- două lucruri de considerat în descrierea unui algoritm recursiv: **regula recursivă** și **condiția de ieșire din recursivitate**.
- **avantaj** al recursivității: text sursă extrem de scurt și foarte clar.
- **dezavantaj** al recursivității: umplerea segmentului de stivă în cazul în care numărul apelurilor recursive, respectiv al parametrilor formali și locali ai subprogramelor recursive este mare.
  - în limbajele declarative există mecanisme specifice de optimizare a recursivității (vezi mecanismul recursivității de coadă în Prolog).

## 2.1 Exemple recursivitate

### Notatii

- o listă este o secvență de elemente ( $l_1 l_2 \dots l_n$ )
- lista vidă (cu 0 elemente) o notăm cu  $\emptyset$
- prin  $\oplus$  notăm operația care adaugă un element în listă

#### 1. Să se creeze lista (1,2,3,...n)

##### a) direct recursiv

$$createLista(n) = \begin{cases} \emptyset & \text{daca } n = 0 \\ createLista(n-1) \oplus n & \text{altfel} \end{cases}$$

##### b) folosind o funcție auxiliară recursivă pentru crearea sublistei ( $i, i+1, \dots, n$ )

// crearea listei formată din elementele  $i, i+1, \dots, n$

Model matematic recursiv

$$create(i, n) = \begin{cases} \emptyset & \text{daca } i > n \\ i \oplus create(i+1, n) & \text{altfel} \end{cases}$$

// crearea listei formată din elementele  $1, 2, \dots, n$

$$createLista(n) = create(1, n)$$

### Pseudocod

Alegerea reprezentării: reprezentare simplu înlănțuită, cu alocare dinamică a nodurilor.

### NodLSI

e: TElement //informația utilă a nodului

urm:  $\uparrow$ NodLSI //adresa la care e memorat următorul nod

### LSI

prim:  $\uparrow$ NodLSI //adresa primului nod din listă

#### **Funcția creeazaNodLSI(e)**

*{pre: e: TElement}*

*{post: se returnează un  $\uparrow$ NodLSI conținând e ca informație utilă}*  
*{se alocă un spațiu de memorare pentru un NodLSI }*

*{p:  $\uparrow$ NodLSI}*

aloca(p)

[p].e  $\leftarrow$  e

[p].urm  $\leftarrow$  NIL

*{rezultatul returnat de funcție}*

**creeazaNodLSI**  $\leftarrow$  p

**SfFuncție**

#### **Funcția creare(i, n)**

*{post: se returnează un  $\uparrow$ NodLSI, pointer spre capul listei înlănțuite formate }*  
*{ din elementele i, i+1,..., n }*

**Dacă** i > n **atunci**

**creare**  $\leftarrow$  NIL

**altfel**

*{ se alocă un spațiu de memorare pentru un NodLSI având }*  
*{ informația utilă e }*

q  $\leftarrow$  **creeazaNodLSI**(i)

*{ se creează legătura între nodul q și capul listei înlănțuite }*  
*{ formate din elementele i+1,..., n }*

[q].urm  $\leftarrow$  **creare**(i+1, n)

**creare**  $\leftarrow$  q

**SfDacă**

**SfFuncție**

#### **Funcția creareLista(n)**

*{post: se returnează un  $\uparrow$ NodLSI, pointer spre capul listei înlănțuite formate }*  
*{ din elementele 1, 2,..., n }*

**creareLista**  $\leftarrow$  **creare**(1, n)

**SfFuncție**

2. Dându-se un număr natural  $n$ , să se calculeze suma  $1+2+3+\dots+n$ .

a) direct recursiv

$$suma(n) = \begin{cases} 0 & \text{daca } n = 0 \\ n + suma(n-1) & \text{altfel} \end{cases}$$

b) folosind o funcție auxiliară recursivă pentru calculul sumei  $i+(i+1)+\dots+n$

$$suma\_aux(n, i) = \begin{cases} 0 & \text{daca } i > n \\ i + suma(n, i+1) & \text{altfel} \end{cases}$$

$$suma(n) = suma\_aux(n, 0)$$

3. Să se construiască lista obținută prin adăugarea unui element la sfârșitul unei liste.

// construirea listei  $(l_1, l_2, \dots, l_n, e)$

$$adaug(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus adaug(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

4. Să se verifice apariția unui element în listă.

$$apare(E, l_1 l_2 \dots l_n) = \begin{cases} fals & \text{daca } l \text{ e vida} \\ adevarat & \text{daca } l_1 = E \\ apare(E, l_2 \dots l_n) & \text{altfel} \end{cases}$$

5. Să se numere de câte ori apare un element în listă.

$$nrap(E, l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca } l \text{ e vida} \\ 1 + nrap(E, l_2 \dots l_n) & \text{daca } l_1 = E \\ nrap(E, l_2 \dots l_n) & \text{altfel} \end{cases}$$

6. Să se verifice dacă o listă numerică este mulțime.

$$eMultime(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l \text{ e vida} \\ \text{fals} & \text{daca } l_1 \in (l_2 \dots l_n) \\ eMultime(l_2 \dots l_n) & \text{altfel} \end{cases}$$

7. Să se construiască lista obținută prin transformarea unei liste numerice în mulțime.

$$multime(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ multime(l_2 \dots l_n) & \text{daca } l_1 \in (l_2 \dots l_n) \\ l_1 \oplus multime(l_2 \dots l_n) & \text{altfel} \end{cases}$$

8. Să se returneze inversa unei liste.

a) direct recursiv

$$invers(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ invers(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

b) folosind o variabilă colectoare

<b>L</b>	<b>Col</b>
(1, 2, 3)	$\emptyset$
(2, 3)	(1)
(3)]	(2, 1)
$\emptyset$	(3, 2, 1)

$$invers\_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ invers\_aux(l_2 \dots l_n, l_1 \oplus Col) & \text{altfel} \end{cases}$$

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \emptyset)$$

9. Să se construiască lista obținută prin ștergerea aparițiilor unui element dintr-o listă.

$$stergere(E, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ l_1 \oplus stergere(E, l_2 \dots l_n) & \text{daca } l_1 \neq E \\ stergere(E, l_2 \dots l_n) & \text{altfel} \end{cases}$$

10. Să se determine al  $k$ -lea element al unei liste ( $k \geq 1$ ).

$$element(l_1 l_2 \dots l_n, k) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ l_1 & \text{daca } k = 1 \\ element(l_2 \dots l_n, k-1) & \text{altfel} \end{cases}$$

11. Să se determine diferența a două mulțimi reprezentate sub formă de listă.

$$diferenta(l_1 l_2 \dots l_n, p_1 p_2 \dots p_m) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ diferenta(l_2 \dots l_n, p_1 p_2 \dots p_m) & \text{daca } l_1 \in (p_1 p_2 \dots p_m) \\ l_1 \oplus diferenta(l_2 \dots l_n, p_1 p_2 \dots p_m) & \text{altfel} \end{cases}$$

### Temă

1. Să se verifice dacă un număr natural este sau nu prim.
2. Să se calculeze suma primelor  $k$  elemente dintr-o listă numerică ( $l_1 l_2 \dots l_n$ ).
3. Să se șteargă primele  $k$  numere pare dintr-o listă numerică.

## CURS 2

### Introducere în limbajul PROLOG

#### Cuprins

Bibliografie .....	1
1. Limbajul Prolog .....	1
1.1 Elemente de bază ale limbajului SWI-Prolog .....	4
1.2 “Matching”. Cum își primesc valori variabilele? .....	6
1.3 Modele de flux .....	7
1.4 Sintaxa regulilor .....	7
1.5 Operatori de egalitate .....	8
1.6 Operatori aritmetici .....	8
2. Exemplu – predicatul “factorial” .....	10

#### Bibliografie

Capitolul 11, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

#### 1. Limbajul Prolog

- Limbajul Prolog (PROgrammation en LOGique) a fost elaborat la Universitatea din Marsilia în jurul anului 1970, ca instrument pentru programarea și rezolvarea problemelor ce implicau reprezentări simbolice de obiecte și relații dintre acestea.
- Prolog are un câmp de aplicații foarte larg: baze de date relaționale, inteligență artificială, logică matematică, demonstrarea de teoreme, sisteme expert, rezolvarea de probleme abstracte sau ecuații simbolice, etc.
- Există standardul ISO-Prolog.
- Nu există standard pentru programare orientată obiect în Prolog, există doar extensii: TrincProlog, SWI-Prolog.
- Vom studia implementarea SWI-Prolog – sintaxa e foarte apropiată de cea a standardului ISO-Prolog.
  - Turbo Prolog, Visual Prolog, GNUProlog, Sicstus Prolog, Parlog, etc.
- SWI-Prolog – 1986
  - oferă o interfață bidirecțională cu limbajele C și Java



- folosește XPCe – un sistem GUI orientat obiect
- *multithreading* – bazat pe suportul *multithreading* oferit de limbajul standard C.

## Program Prolog

- caracter descriptiv: un program Prolog este o colecție de definiții ce descriu relații sau funcții de calculat – reprezentări simbolice de obiecte și relații între obiecte. Soluția problemelor nu se mai vede ca o execuție pas cu pas a unei secvențe de instrucțiuni.
  - program – colecție de declarații logice, fiecare fiind o clauză Horn de forma  $p, p \rightarrow q, p_1 \wedge p_2 \dots \wedge p_n \rightarrow q$
  - **concluzie** de demonstrat – de forma  $p_1 \wedge p_2 \dots \wedge p_n$
- **Structura de control** folosită de interpretorul Prolog
  - Se bazează pe declarații logice numite **clauze**
    - **fapt** – ceea ce se cunoaște a fi adevărat
    - **regulă** - ce se poate deduce din fapte date (indică o concluzie care se știe că e adevărată atunci când alte concluzii sau fapte sunt adevărate)
  - **Concluzie** ce trebuie demonstrată - GOAL
    - Prolog folosește *rezoluția* (liniară) pentru a demonstra dacă concluzia (teorema) este adevărată sau nu, pornind de la ipoteza stabilită de faptele și regulile definite (axiome).
    - Se aplică raționamentul înapoi pentru a demonstra concluzia
    - Programul este citit de sus în jos, de la dreapta la stânga, căutarea este în adâncime (*depth-first*) și se realizează folosind **backtracking**.
- $p \rightarrow q ( \neg p \vee q )$  se transcrie în Prolog folosind clauza  $q :- p.$  ( $q$  if  $p.$ )
- $\wedge$  se transcrie în Prolog folosind ",",
  - $p_1 \wedge p_2 \dots \wedge p_n \rightarrow q$  se transcrie în Prolog folosind clauza  $q :- p_1, p_2, \dots, p_n.$
- $\vee$  se transcrie în Prolog folosind ";" sau o clauză separată.
  - $p_1 \vee p_2 \rightarrow q$  se transcrie în Prolog
    - folosind clauza  $q :- p_1; p_2.$
    - sau
    - folosind 2 clauze separate
      - $q :- p_1.$
      - $q :- p_2.$

## Exemple

- Logică**

$$\forall x p(x) \wedge q(x) \rightarrow r(x)$$

$$\forall x w(x) \vee s(x) \rightarrow p(x)$$

**(SWI-)Prolog**

$$r(X) : -p(X), q(X).$$

$$p(X) : -w(X).$$

$$p(X) : -s(X).$$

$w \vee s \rightarrow p$	$\neg(w \vee s) \vee p$	$(\neg w \wedge \neg s) \vee p$	$(\neg w \vee p) \wedge (\neg s \vee p)$	$(w \rightarrow p) \wedge (s \rightarrow p)$
--------------------------	-------------------------	---------------------------------	--	--

$$\forall x t(x) \rightarrow s(x) \wedge q(x)$$

$$s(X) : -t(X).$$

$$q(X) : -t(X).$$

$$t(a)$$

$$t(a).$$

$$w(b)$$

$$w(b).$$

**Concluzie**

$$r(a)$$

**Goal**

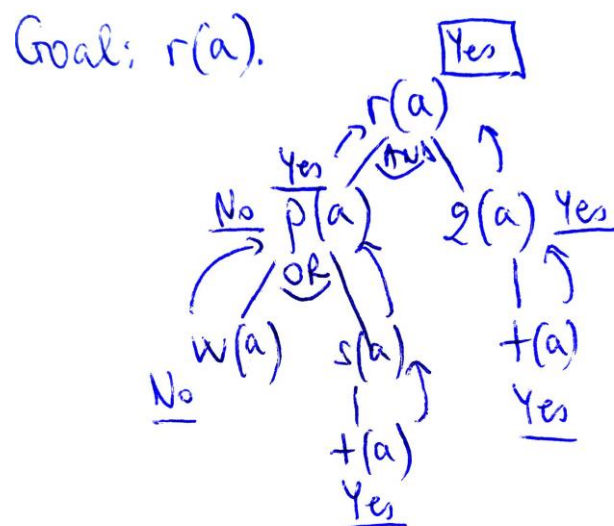
$$? r(a).$$

true

$$q(b)$$

$$? q(b).$$

**false**



- Logică**

$$\forall x s(x) \rightarrow p(x) \vee q(x)$$

**Prolog**

????

$s \rightarrow p \vee q$	... ..	... ..	$(s \wedge \neg p) \rightarrow q$
--------------------------	--------	--------	-----------------------------------

- [Lawrence C Paulson, \*Logic and Proof\*, University of Cambridge, 2000](#): rezoluție (subcapitolul 7.3), unificare (capitolul 10), Prolog

## 1.1 Elemente de bază ale limbajului SWI-Prolog

### 1. Termen

#### ▪ SIMPLU

##### a. constantă

- simbol (*symbol*)
  - secvență de litere, cifre, \_
  - începe cu **literă mică**
- număr =întreg, real (*number*)
- șir de caractere (*string*): ‘text’ (caracter: ‘c’, ‘\t’,...)

ATOM = SIMBOL + STRING +ȘIR-DE-CARACTERE-SPECIALE + [] (lista vidă)

- caractere speciale + \* / < > = : . & \_ ~

##### b. variabilă

- secvență de litere, cifre, \_
- începe cu **literă mare**
- variabila anonimă este reprezentată de caracterul underline (\_).

#### ▪ COMPUS (a se vedea **Curs 4**).

- listele (*list*) sunt o clasă specială de termeni compuși

### 2. Comentariu

% Acesta este un comentariu

/\* Acesta este un comentariu \*/

### 3. Predicat

a). standard (ex: **fail**, **number**, ...)

b). utilizator

- $\text{nume} [(\text{obiect}[, \text{obiect}....])]$   
 ↓  
 numele simbolic al relației

*Tipuri*

1. **number** (integer, real)
2. **atom** (symbol, string, șir-de-caractere-speciale)

### 3. **list** (secvență de elemente) specificat ca **list=tip\_de\_bază\***

ex. listă (omogenă) formată din numere întregi [1,2,3]

% definire tip:            el=integer        list=el\*

**!!! lista vidă [] este singura listă care e considerată în Prolog atom.**

### Convenții.

- În SWI-Prolog nu există declarații de predicate, nici declarații de domenii/tipuri (ex. ca în Turbo-Prolog).
- *Specificarea unui predicat*
  - % definire tipuri, dacă e cazul
  - % *nume* [(param<sub>1</sub>:tip<sub>1</sub>[,param<sub>2</sub>:tip<sub>2</sub>...)]
  - % modelul de flux al predicatului (i, o, ...) - vezi **Secțiunea 1.3**
  - % param<sub>1</sub> - semnificația parametrului 1
  - % param<sub>2</sub> - semnificația parametrului 2
  - ....

### 4. Clauza

- fapt
  - relație între obiecte
  - *nume\_predicat* [(obiect [, obiect....)]
- regula
  - permite deducere de fapte din alte fapte

#### Exemplu:

fie predicatele

**tata**(X, Y) reprezentând relația “Y este tatăl lui X”

**mama**(X, Y) reprezentând relația “Y este mama lui X”

și următoarele fapte corespunzătoare celor două predicate:

mama(a,b).

mama(e,b).

tata(c,d).

tata(a,d).

**Se cere:** folosind definițiile anterioare să se definească predicatele

**parinte**(X, Y) reprezentând relația “Y este părintele lui X”

**frate**(X, Y) reprezentând relația “Y este fratele lui X”

## Clauze în SWI-Prolog

```
parinte(X,Y) :-tata(X,Y).
parinte(X,Y) :-mama(X,Y).
% "\=" reprezintă operatorul "diferit" – a se vedea Secțiunea 1.5
frate(X,Y) :- parinte(X,Z),
               parinte(Y,Z),
               X \= Y.
```

### 5. Intrebare (goal)

- e de forma  $predicat_1 [(obiect [, obiect....)], predicat_2 [(obiect [, obiect....)]....]$ .
- **true**, **false**
- **CWA – Closed World Assumption**

Folosind definițiile anterioare, formulăm următoarele întrebări:

?- parinte(a,b). <b>true.</b>	?- parinte(a,X). X=d; X=b.
? - parinte(a,f). <b>false.</b>	
?- frate(a,X). X=c; X=e.	?- frate(a,_). <b>true.</b>

## 1.2 "Matching". Cum își primesc valori variabilele?

Prolog nu are instrucțiuni de atribuire. Variabilele în Prolog își primesc valorile prin **potrivire** cu constante din fapte sau reguli.

Până când o variabilă primește o valoare, ea este **liberă** (free); când variabila primește o valoare, ea este **legată** (bound). Dar ea stă legată atâta timp cât este necesar pentru a obține o soluție a problemei. Apoi, Prolog o dezleagă, face backtracking și caută soluții alternative.

**Observație.** Este important de reținut că nu se pot stoca informații prin atribuire de valori unor variabile. Variabilele sunt folosite ca parte a unui proces de potrivire, nu ca un tip de stocare de informații.

### Ce este o potrivire?

Iată câteva reguli care vor explica termenul 'potrivire':

1. Structuri identice se potrivesc una cu alta
  - $p(a, b)$  se potrivește cu  $p(a, b)$

2. De obicei o potrivire implică variabile libere. Dacă X e liberă,
  - $p(a, X)$  se potrivește cu  $p(a, b)$
  - X este legat la b.
3. Dacă X este legat, se comportă ca o constantă. Cu X legat la b,
  - $p(a, X)$  se potrivește cu  $p(a, b)$
  - $p(a, X)$  NU se potrivește cu  $p(a, c)$
4. Două variabile libere se potrivesc una cu alta.
  - $p(a, X)$  se potrivește cu  $p(a, Y)$

**Observație.** Mecanismul prin care Prolog încearcă să ‘potrivească’ partea din întrebare pe care dorește să o rezolve cu un anumit predicat se numește **unificare**.

### 1.3 Modele de flux

În Prolog, legările de variabile se fac în două moduri: la intrarea în clauză sau la ieșirea din clauză. Direcția în care se leagă o valoare se numește ‘**model de flux**’. Când o variabilă este dată la intrarea într-o clauză, aceasta este un parametru de intrare (i), iar când o variabilă este dată la ieșirea dintr-o clauză, aceasta este un parametru de ieșire (o). O anumită clauză poate să aibă mai multe modele de flux. De exemplu clauza

factorial (N, F)

poate avea următoarele modele de flux:

- (i,i) - verifică dacă  $N! = F$ ;
- (i,o) – determină  $F = N!$ ;
- (o,i) - găsește acel N pentru care  $N! = F$ .

**Observație.** Proprietatea unui predicat de a funcționa cu mai multe modele de flux depinde de abilitatea programatorului de a programa predicatul în mod corespunzător.

### 1.4 Sintaxa regulilor

Regulile sunt folosite în Prolog când un fapt depinde de succesul (veridicitatea) altor fapte sau succesiuni de fapte. O regulă Prolog are trei părți: capul, corpul și simbolul if (:-) care le separă pe primele două.

Iată sintaxa generică a unei reguli Prolog:

capul regulii :-  
           subgoal,  
           subgoal,  
           ...,  
           subgoal.

Fiecare subgoal este un apel la un alt predicat Prolog. Când programul face acest apel, Prolog testează predicatul apelat să vadă dacă poate fi adevărat. Odată ce subgoal-ul curent a fost

satisfăcut (a fost găsit adevărat), se revine și procesul continuă cu următorul subgoal. Dacă procesul a ajuns cu succes la punct, regula a reușit. Pentru a utiliza cu succes o regulă, Prolog trebuie să satisfacă toate subgoal-urile ei, creând o mulțime consistentă de legări de variabile. Dacă un subgoal eșuează (este găsit fals), procesul revine la subgoal-ul anterior și caută alte legări de variabile, și apoi continuă. Acest mecanism se numește **backtracking**.

## 1.5 Operatori de egalitate

$X=Y$  verifică dacă X și Y pot fi unificate

- Dacă X este variabilă liberă și Y legată, sau Y este variabilă liberă și X e legată, propoziția este satisfăcută unificând pe X cu Y.
- Dacă X și Y sunt variabile legate, atunci propoziția este satisfăcută dacă relația de egalitate are loc.

$?- [a,b]=[a,b].$ <b>true.</b>	$?- [X,Y]=[a,b].$ $X = a,$ $Y = b.$	$?- [a,b]=[X,Y].$ $X = a,$ $Y = b.$
-----------------------------------	---	---

$X \neq Y$  verifică dacă X și Y nu pot fi unificate  
 $\neq X=Y$

$?- [X,Y,Z] \neq [a,b].$ <b>true.</b>	$?- [X,Y] \neq [a,b].$ <b>false.</b>	$?- [a,b] \neq [X,Y].$ <b>false.</b>
$?- \neq a=a.$ <b>false.</b>	$?- \neq [X,Y]=[a,b].$ <b>false.</b>	$?- \neq [a,b]=[X,Y,Z].$ <b>true.</b>

$X == Y$  verifică dacă X și Y sunt legate la aceeași valoare.

$?- [2,3]==[2,3].$ <b>true.</b>	$?- a==a.$ <b>true.</b>	$?- R==1.$ <b>false.</b>
------------------------------------	----------------------------	-----------------------------

$X \neq Y$  verifică dacă X și Y nu au fost legate la aceeași valoare.

$?- [2,3] \neq [3,2].$ <b>true.</b>	$?- a \neq a.$ <b>false.</b>	$?- R \neq 1.$ <b>true.</b>
--	---------------------------------	--------------------------------

## 1.6 Operatori aritmetici

### !!! Important

- $2+4$  e doar o structură, utilizarea sa nu efectuează adunarea
- Utilizarea  $2+4$  nu e aceeași ca utilizarea lui 6.

## Operatori aritmetici

=, \=, ==, \==

A se vedea Secțiunea 1.5.

?- 2+4=6. <b>false.</b>	?- 2+4\=6. <b>true.</b>	?- 6==6. <b>true.</b>	?- 6\=7. <b>true.</b>	?- 6==2+4. <b>false.</b>
----------------------------	----------------------------	--------------------------	--------------------------	-----------------------------

?- 2+4=2+4. <b>true.</b>	?- 2+4=4+2. <b>false.</b>	?- X= 2+4-1. X=2+4-1.
-----------------------------	------------------------------	--------------------------

==

- testează egalitatea aritmetică
- forțează evaluarea aritmetică a ambelor părți
- operanzii trebuie să fie numerici
- variabilele sunt LEGATE

\=

testează operatorul aritmetic "diferit"

?- 2+4\:=6. <b>true.</b>	?- 2+4\=\=7. <b>true.</b>	?- 6\:=6. <b>true.</b>
-----------------------------	------------------------------	---------------------------

is

- partea dreaptă este LEGATĂ și numerică
- partea stângă trebuie să fie o variabilă
- dacă variabila este legată, verifică egalitatea numerică (ca și ==)
- dacă variabila nu este legată, evaluează partea dreaptă și apoi variabila este legată de rezultatul evaluării

?- X is 2+4-1. X=5	?- X is 5. X=5
-----------------------	-------------------

## Inegalități

<	mai mic
=<	mai mic sau egal
>	mai mare
>=	mai mare sau egal

- evaluează ambele părți
- variabile LEGATE

?- 2+4=<5+2. <b>true.</b>	?- 2+4=\=7. <b>true.</b>	?- 6\:=6. <b>true.</b>
------------------------------	-----------------------------	---------------------------

## Câteva funcții aritmetice predefinite SWI-Prolog

X mod Y                      întoarce restul împărțirii lui X la Y



X div Y	întoarce câtul împărțirii lui X la Y
abs(X)	întoarce valoarea absolută a lui X
sqrt(X)	întoarce rădăcina pătrată a lui X
round(X)	întoarce valoarea lui X rotunjită spre cel mai apropiat întreg (round(2.56) este 3, round (2.4) este 2)
...	

## 2. Exemplu – predicatul “factorial”

Dându-se un număr natural n, să se calculeze factorialul numărului.

$$fact(n) = \begin{cases} 1 & \text{daca } n = 0 \\ n \cdot fact(n-1) & \text{altfel} \end{cases}$$

Conform cerinței probleme, am dori să definim predicatul **fact(integer, integer)** în modelul de flux **(i, o)**. Vom vedea că predicatul definit în acest model de flux funcționează și în modelul de flux (i, i).

În Varianta 2, ! reprezintă predicatul “cut” (tăietura roșie, în acest context), folosit pentru a preveni luarea în calcul a subgoal-urilor alternative (backtracking-ul la următoarea clauză).

### 1. Varianta 1

```
% fact1(N:integer, F:integer)
% (i, i) (i, o)
fact1(0, 1).
fact1(N, F) :- N > 0,
               N1 is N-1,
               fact1(N1, F1),
               F is N * F1.
```

```
go1 :- fact1(3, 6).
```

### 2. Varianta 2

```
% fact1(N:integer, F:integer)
% (i, i) (i, o)
fact2(0, 1) :- !.
fact2(N, F) :- N1 is N-1,
               fact2(N1, F1),
               F is N * F1.
```

```
go2 :- fact2(3, 6).
```

```
SWI-Prolog (Multi-threaded, version 6.6.6)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
% d:/Docs/Didactice/Cursuri/2014-15/pfl/teste/fact.pl compiled 0.00 sec, 7 clauses
1 ?- fact1(3,6).
true ;
false.

2 ?- fact1(3,X).
X = 6 ;
false.

3 ?- go1.
true ;
false.

4 ?- fact2(3,6).
true.

5 ?- fact2(3,X).
X = 6.

6 ?- go2.
true.

7 ?- █
```

### 3. Varianta 3

**% fact3(N:integer, F:integer)**

**% (i, i), (i, o)**

```
fact3(N, F) :- N > 0,
               N1 is N-1,
               fact3(N1, F1),
               F is N * F1.
```

```
fact3(0, 1).
```

```
?- fact3(3, N).
```

```
N = 6.
```

## CURS 3

### Predicate deterministe și nedeterministe. Exemple

#### Cuprins

Bibliografie .....	1
1. Predicate deterministe și nedeterministe.....	1
1.1 Predicate predefinite .....	1
1.2 Predicatul „findall“ (determinarea tuturor soluțiilor) .....	2
1.3 Negatie - “not”, “\+” .....	2
1.4 Liste și recursivitate .....	2
1.4.1 Capul și coada unei liste (head&tail).....	3
1.4.2 Procesarea listelor .....	3
1.4.3 Utilizarea listelor.....	3
2. Exemple .....	4

#### Bibliografie

Capitolul 14, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

### 1. Predicate deterministe și nedeterministe

Tipuri de predicate

- **deterministe**
  - un predicat determinist are o singură soluție
- **nedeterministe**
  - un predicat nedeterminist are mai multe soluții

**Observație.** Un predicat poate fi determinist într-un model de flux, nedeterminist în alte modele de flux.

#### 1.1 Predicate predefinite

var(X)	= adevărat dacă X e liberă, fals dacă e legată
number(X)	= adevărat dacă X e legată la un număr
integer(X)	= adevărat dacă X e legată la un număr întreg
float(X)	= adevărat dacă X e legată la un număr real
atom(X)	= adevărat dacă X e legată la un atom
atomic(X)	= atom(X) or number(X)
....	

## 1.2 Predicatul „findall” (determinarea tuturor soluțiilor)

Prolog oferă o modalitate de a găsi toate soluțiile unui predicat în același timp: predicatul **findall**, care colectează într-o listă toate soluțiile găsite.

**findall** (arg1, arg2, arg3)

Acesta are următoarele argumente:

- primul argument specifică argumentul din predicatul considerat care trebuie colectat în listă;
- al doilea argument specifică predicatul de rezolvat;
- al treilea argument specifică lista în care se vor colecta soluțiile.

### EXEMPLU

p(a, b).  
p(b, c).  
p(a, c).  
p(a, d).  
toate(X, L) :- findall(Y, p(X, Y), L).

? toate(a, L).  
L=[b, c, d]

## 1.3 Negație - “not”, “\+”

**not**(subgoal(Arg1, ..., ArgN))

adevărat dacă *subgoal* eșuează (nu se poate demonstra că este adevărat)

**\+** subgoal(Arg1, ..., ArgN)

?- \+ (2 = 4).  
**true.**

?- not(2 = 4).  
**true.**

## 1.4 Liste și recursivitate

În Prolog, o listă este un obiect care conține un număr arbitrar de alte obiecte. Listele în SWI-Prolog sunt eterogene (elementele componente pot avea tipuri diferite). Listele se construiesc folosind parantezele drepte. Elementele acestora sunt separate de virgulă.

Iată câteva exemple:

[1, 2, 3]  
[dog, cat, canary]  
[“valerie ann”, “jennifer caitlin”, “benjamin thomas”]

Dacă ar fi să declarăm tipul unei liste (omogenă) cu elemente numere întregi, s-ar folosi o declarație de domeniu de tipul următor

element = integer  
list = element\*

### 1.4.1 Capul și coada unei liste (head&tail)

O listă este un obiect realmente recursiv. Aceasta constă din două părți: **capul**, care este primul element al listei și **coada**, care este restul listei. Capul listei este element, iar coada listei este listă.

Iată câteva exemple:

Capul listei [a, b, c] este a

Coada listei [a, b, c] este [b, c]

Capul listei [c] este c

Coada listei [c] este []

Lista vidă [] nu poate fi împărțită în cap și coada.

### 1.4.2 Procesarea listelor

Prolog oferă o modalitate de a face explicite capul și coada unei liste. În loc să separăm elementele unei liste cu virgule, vom separa capul de coadă cu caracterul '|'.  
De exemplu, următoarele liste sunt echivalente:

[a, b, c]      [a | [b, c]]      [a | [b | [c]]]      [a | [b | [c | []]]]

De asemenea, înaintea caracterului '|' pot fi scrise mai multe elemente, nu doar primul. De exemplu, lista [a, b, c] de mai sus este echivalentă cu

[a | [b, c]]      [a, b | [c]]      [a, b, c | []]

- În urma unificării listei [a, b, c] cu lista [H | T] (H, T fiind variabile libere)
  - H se leagă la a; T se leagă la [b, c]
- În urma unificării listei [a, b, c] cu lista [H | [H1 | T]] (H, H1, T fiind variabile libere)
  - H se leagă la a; H1 se leagă la b; T se leagă la [c]

### 1.4.3 Utilizarea listelor

Deoarece o listă (înlănțuită) este o structură de date recursivă, pentru procesarea ei este nevoie de algoritmi recursivi. Modul de bază de procesare a listei este acela de a lucra cu ea, executând anumite operații cu fiecare element al ei, până când s-a atins sfârșitul.

Un algoritm de acest tip are nevoie în general de două clauze. Una dintre ele spune ce să se facă cu o listă vidă. Cealaltă spune ce să se facă cu o listă nevidă, care se poate descompune în cap și coadă.

## 2. Exemple

### EXEMPLU 2.1 Adăugarea unui element la sfârșitul unei liste

? adaug(3, [1, 2], L).

L = [1, 2, 3]

Formula recursivă:

$$adaug(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus adaug(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

#### Varianta 1

% adaug(e:element, L:list, LRez: list)

% (i, i, o) - determinist

adaug(E, [], [E]). % adaug(E, [], Rez) :- Rez = [E].

adaug(E, [H | T], [H | Rez]) :-

adaug(E, T, Rez).

% adaug(E, [H | T], Rez) :-

adaug(E, T, L), Rez = [H | L].

#### Varianta 2

% adaug(L:list, e:element, LRez: list)

% (i, i, o) - determinist

adaug([], E, [E]).

adaug([H | T], E, [H | Rez]) :-

adaug(T, E, Rez).

% adaug([H | T], E, Rez) :-

adaug(T, E, L), Rez = [H | L].

Complexitatea timp a operației de adăugare a unui element la sfârșitul unei liste cu  $n$  elemente este  $\theta(n)$ .

- Alte modele de flux?

% (i, i, i) - determinist ? adaug(1, [2, 3], [2, 3, 1]). <b>true.</b>	% (o, i, i) - determinist ? adaug(E, [2, 3], [2, 3, 1]). E=1.	% (i, o, i) - determinist ? adaug(1, L, [2, 3, 1]). L=[2, 3]; <b>false.</b>	% (o, o, i) - determinist ? adaug(E, L, [2, 3, 1]). E=1 L=[2, 3]; <b>false.</b>
---	---	--	---

### EXEMPLU 2.2 Verificați apartenența unui element într-o listă.

% (i, i) ? member(3, [1, 2, 3]).

Pentru a descrie apartenența la o listă vom contrui predicatul member(element, list) care va investiga dacă un anumit element este membru al listei. Algoritmul de implementat ar fi (din punct de vedere declarativ) următorul:

1. E este membru al listei L dacă este capul ei.
2. Altfel, E este membru al listei L dacă este membru al cozii lui L.

Din punct de vedere procedural,

1. Pentru a găsi un membru al listei L, găsește-i capul;
2. Altfel, găsește un membru al cozii listei L.

$$member(E, l_1 l_2 \dots l_n) = \begin{cases} fals & \text{daca } l \text{ e vida} \\ adevarat & \text{daca } l_1 = E \\ member(E, l_2 \dots l_n) & \text{altfel} \end{cases}$$

### 1. Varianta 1

```
% member(e:element, L:list)
% (i, i) - determinist, (o, i) - nedeterminist
member1(E,[E|_]).
member1(E,[_|L]) :- member1(E,L).
```

```
go1 :- member1(1,[1,2,1,3,1,4]).
```

### 2. Varianta 2

```
% member(e:element, L:list)
% (i, i) - determinist, (o, i) - nedeterminist
member2(E,[E|_]) :- !.
member2(E,[_|L]) :- member2(E,L).
```

```
go2 :- member2(1,[1,2,1,3,1,4]).
```

```
SWI-Prolog (Multi-threaded, version 6.6.6)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.6.6)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?-
% d:/Docs/Didactice/Cursuri/2014-15/pfl/teste/member.pl compiled 0.00 sec, 7 clauses
1 ?- member1(1,[1,2,1,3,1,4]).
true ;
true ;
true ;
false.

2 ?- member1(X,[1,2,1,3,1,4]).
X = 1 ;
X = 2 ;
X = 1 ;
X = 3 ;
X = 1 ;
X = 4 ;
false.

3 ?- member1(1,[2,1,3]).
true ;
false.

4 ?- member1(5,[2,1,3]).
false.

5 ?- go1.
true ;
true ;
true ;
false.

6 ?- member2(1,[1,2,1,3,1,4]).
true.

7 ?- member2(X,[1,2,1,3,1,4]).
X = 1.

8 ?- member2(1,[2,1,3]).
true.

9 ?- member2(5,[2,1,3]).
false.

10 ?- go2.
true.
```

### 3. Varianta 3

% member(e:element, L:list)  
% (i, i) - determinist, (o, i) - nedeterminist

member3(E,[\_|L]) :- member3(E,L).  
member3(E,[E|\_]).

?- member3(E,[1,2,3]). E=3 ; E=2 ; E=1.	?-member3(4, [1,2,3]). <b>false.</b>	?- member3(2,[1,2,3]). true ; <b>false.</b>
--	---	---

După cum se observă, predicatul **member** funcționează și în modelul de flux (o, i), în care este **nedeterminist**.

Pentru a descrie predicatul *member* în modelul de flux (o, i) – o soluție să fie câte un element al listei -

?- member3(E,[1,2,3]).  
E=1;  
E=2;  
E=3.

$member(l_1, l_2, \dots, l_n) =$

1.  $l_1$       *daca l e nevida*
2.  $member(l_2, \dots, l_n)$

**EXEMPLU 2.3** Dându-se un număr natural  $n$  nenul, se cere să se calculeze  $F=n!$ . Se va simula procesul iterativ de calcul.

$i \leftarrow 1$   
 $P \leftarrow 1$   
**CâtTimp**  $i < n$  execută  
     $i \leftarrow i + 1$   
     $P \leftarrow P * i$   
**SfCâtTimp**  
 $F \leftarrow P$

$fact(n) = fact\_aux(n, 1, 1)$

$fact\_aux(n, i, P) = \begin{cases} P & \text{daca } i = n \\ fact\_aux(n, i + 1, P * (i + 1)) & \text{altfel} \end{cases}$

- descrierea nu este direct recursivă, se folosesc variabile colectoare ( $i, P$ )



```
% fact(N:integer, F:integer)
% (i, i), (i, o) - determinist
fact(N, F) :- fact_aux(N, F, 1, 1).
```

```
% fact_aux(N:integer, F:integer, I:integer, P:integer)
% (i, i, i, i), (i, o, i, i) - determinist
fact_aux(N, F, N, F) :- !. % fact_aux(N, F, I, P) :- I is N, F is P, !.
fact_aux(N, F, I, P) :- I1 is I+1,
                        P1 is P*I1,
                        fact_aux(N, F, I1, P1).
```

Rezultatul este o recursivitate de coadă (a se vedea **Cursul 6**). Toate variabilele de ciclare au fost introduse ca argumente ale predicatului **fact\_aux**.

**TEMĂ** Scrieți un predicat factorial (N, F) care să funcționeze în toate cele 3 modele de flux (i, i), (i, o) și (o, i).

#### **EXEMPLU 2.4** Să se determine inversa unei liste.

**Varianta A** (direct recursiv)

$$invers(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ invers(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

```
% invers(L:list, LRez: list)
% (i, o) - determinist
invers([], []).
invers([H | T], Rez) :-
    invers(T, L), adaug(H, L, Rez).
```

Complexitatea timp a operației de inversare a unei liste cu  $n$  elemente (folosind adăugarea la sfârșit) este  $\theta(n^2)$ .

**Varianta B** (cu variabilă colectoare)

Se va folosi o variabilă colectoare **Col**, pentru a colecta inversa listei, pas cu pas.

L	Col
[1, 2, 3]	$\emptyset$
[2, 3]	[1]
[3]	[2, 1]
$\emptyset$	[3, 2, 1]

$$invers\_aux(l_1 l_2 \dots l_n Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ invers\_aux(l_2 \dots l_n l_1 \oplus Col) & \text{altfel} \end{cases}$$

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \emptyset)$$

*% invers(L:list, LRez: list)*

*% (i, o) – determinist*

*invers(L, Rez) :- invers\_aux([], L, Rez).*

*% invers\_aux(Col:list, L:list, LRez: list) – primul argument e colectoarea*

*% (i, i, o) – determinist*

*invers\_aux(Col, [], Col). % invers\_aux(Col, [], Rez) :- Rez = Col.*

*invers\_aux(Col, [H|T], Rez) :-*

*invers\_aux([H|Col], T, Rez).*

Complexitatea timp a operației de inversare a unei liste cu  $n$  elemente (folosind o variabilă colectoare) este  $\theta(n)$ .

**Observație.** Folosirea unei variabile colectoare nu reduce complexitatea, în toate cazurile. Sunt situații în care folosirea unei variabile colectoare crește complexitatea (ex: adăugarea în colectoare se face la sfârșitul acesteia, nu la început).

**EXEMPLU 2.5** Să se determine lista elementelor pare dintr-o listă (se va păstra ordinea elementelor din lista inițială).

**Varianta A** (direct recursiv)

$$pare(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ l_1 \oplus pare(l_2 \dots l_n) & \text{daca } l_1 \text{ par} \\ pare(l_2 \dots l_n) & \text{altfel} \end{cases}$$

*% pare(L:list, LRez: list)*

*% (i, o) – determinist*

*pare([], []).*

*pare([H|T], [H|Rez]) :-*

*H mod 2 =:= 0,*

*!,*

*pare(T, Rez).*

*pare([\_|T], Rez) :-*

*pare(T, Rez).*

Complexitatea timp a operației este  $\theta(n)$ ,  $n$  fiind numărul de elemente din listă.

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n-1) + 1 & \text{altfel} \end{cases}$$

**Varianta B** (cu variabilă colectoare)

$$pare\_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ pare\_aux(l_2 \dots l_n, Col \oplus l_1) & \text{daca } l_1 \text{ par} \\ pare\_aux(l_2 \dots l_n, Col) & \text{altfel} \end{cases}$$

$$pare(l_1 l_2 \dots l_n) = pare\_aux(l_1 l_2 \dots l_n, \emptyset)$$

```
% pare(L:list, LRez: list)
% (i, o) – determinist
pare(L, Rez) :-
    pare_aux(L, Rez, []).
% pare(L:list, LRez: list, Col: list)
% (i, o, i) – determinist
pare_aux([], Rez, Rez).
pare_aux([H|T], Rez, Col) :-
    H mod 2 =:= 0,
    !,
    adaug(H, Col, ColN), % adăugare la sfârșit
    pare_aux(T, Rez, ColN).
pare_aux([_|T], Rez, Col) :-
    pare_aux(T, Rez, Col).
```

Complexitatea timp în caz defavorabil este  $\theta(n^2)$ ,  $n$  fiind numărul de elemente din listă.

**EXEMPLU 2.6** Dându-se o listă numerică, se cere un predicat care determină lista perechilor de elemente strict crescătoare din listă.

```
? perechi([2, 1, 3, 4], L)
L = [[2, 3], [2, 4], [1, 3], [1, 4], [3, 4]]
```

```
? perechi([5, 4, 2], L)
false
```

Vom folosi următoarele predicate:

- predicatul nedeterminist **pereche**(element, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementul dat și elemente ale listei argument
 

```
? pereche(2, [1, 3, 4], L)
L = [2, 3]
L = [2, 4]
```

*pereche*(*e*, *l*<sub>1</sub>, *l*<sub>2</sub>, ..., *l*<sub>*n*</sub>) =

1. (*e*, *l*<sub>1</sub>)                *e* < *l*<sub>1</sub>
2. *pereche*(*e*, *l*<sub>2</sub>, ..., *l*<sub>*n*</sub>)

% *pereche*(*E*: element, *L*:list, *LRez*: list)

% (i, i, o) – nedeterminist

*pereche*(*A*, [*B*\_|], [*A*, *B*]) :-

*A* < *B*.

*pereche*(*A*, [\_|*T*], *P*) :-

*pereche*(*A*, *T*, *P*).

- predicatul nedeterminist **perechi**(lista, lista) (model de flux (i, o)), care va produce perechi în ordine crescătoare între elementele listei argument

? *perechi*([2, 1, 4], *L*)

*L* = [2, 4]

*L* = [1, 4]

*perechi*(*l*<sub>1</sub>, *l*<sub>2</sub>, ..., *l*<sub>*n*</sub>) =

1. *pereche*(*l*<sub>1</sub>, *l*<sub>2</sub>, ..., *l*<sub>*n*</sub>)
2. *perechi*(*l*<sub>2</sub>, ..., *l*<sub>*n*</sub>)

% *perechi*(*L*:list, *LRez*: list)

% (i, o) – nedeterminist

*perechi*([*H*|*T*], *P*) :-

*pereche*(*H*, *T*, *P*).

*perechi*([\_|*T*], *P*) :-

*perechi*(*T*, *P*).

- predicatul principal **toatePerechi**(lista, listad) (model de flux (i, o)), care va colecta toate soluțiile predicatului nedeterminist **perechi**.

% *toatePerechi*(*L*:list, *LRez*: list)

% (i, o) –determinist

*toatePerechi*(*L*, *LRez*) :-

findall(*X*, *perechi*(*L*, *X*), *LRez*).

**EXEMPLU 2.7** Se dă o listă eterogenă formată din numere, simboluri și/sau liste de numere. Se cere să se determine suma numerelor din lista eterogenă.

?- suma([1,a,[1,2,3],4],*S*).

*S* = 11.

?- suma([a,b,[],],*S*).

*S*=0.

**Observatie.** În SWI-Prolog listele sunt eterogene, elementele componente pot fi de tipuri diferite. Pentru a se determina tipul unui element al listei, se folosesc predicatele predefinite în SWI (**number**, **is\_list**, etc)

```
%(L:list of numbers, S: number)  
% (i,o) - determ  
sumalist([],0).  
sumalist([H|T],S) :- sumalist(T,S1),  
                     S is S1+H.
```

```
%(L:list, S: number)  
% (i,o) - determ  
suma([],0).  
suma([H|T],S):-number(H),  
               !,  
               suma(T,S1),  
               S is H+S1.  
suma([H|T],S):-is_list(H),  
               !,  
               sumalist(H,S1),  
               suma(T,S2),  
               S is S1+S2.  
suma([_|T],S):-suma(T,S).
```

# CURS 4

## Controlarea backtrackingului

### Cuprins

1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail.....	1
1.1 Predicatul ! (cut) – “tăietura” .....	1
1.2 Predicatul “fail” .....	3
2. Exemple .....	5

## 1. Controlarea backtracking-ului. Predicatele “cut” (tăietura) și fail

### 1.1 Predicatul ! (cut) – “tăietura”

Limbajul Prolog conține predicatul cut (!) folosit pentru a preveni backtracking-ul. Când se procesează predicatul !, apelul reușește imediat și se trece la subgoalul următor. O dată ce s-a trecut peste o tăietură, nu este posibilă revenirea la subgoal-urile plasate înaintea ei și nu este posibil backtracking-ul la alte reguli ce definesc predicatul în execuție.

Există două utilizări importante ale tăieturii:

1. Când știm dinainte că anumite posibilități nu vor duce la soluții, este o pierdere de timp să lăsăm sistemul să lucreze. În acest caz, tăietura se numește **tăietură verde**.
2. Când logica programului cere o tăietură, pentru prevenirea luării în considerație a subgoal-urilor alternative, pentru a evita obținerea de soluții eronate. În acest caz, tăietura se numește **tăietură roșie**.

### Prevenirea backtracking-ului la un subgoal anterior

În acest caz tăietura se utilizează astfel:  $r1 :- a, b, !, c.$

Aceasta este o modalitate de a spune că suntem mulțumiți cu primele soluții descoperite cu subgoal-urile a și b. Deși Prolog ar putea găsi mai multe soluții prin apelul la c, nu este autorizat să revină la a și b. De-asemenea, nu este autorizat să revină la altă clauză care definește predicatul r1.

### Prevenirea backtracking-ului la următoarea clauză

Tăietura poate fi utilizată pentru a-i spune sistemului Prolog că a ales corect clauza pentru un predicat particular. De exemplu, fie codul următor:

```
r(1) :- !, a, b, c.  
r(2) :- !, d.  
r(3) :- !, e.  
r(_) :- write("Aici intră restul apelurilor").
```

Folosirea tăieturii face predicatul *r* determinist. Aici, Prolog apelează predicatul *r* cu un argument întreg. Să presupunem că apelul este *r*(1). Prolog caută o potrivire a apelului. O găsește la prima clauză. Faptul că imediat după intrarea în clauză urmează o tăietură, împiedică Prolog să mai caute și alte potriviri ale apelului *r*(1) cu alte clauze.

**Observație.** Acest tip de structură este echivalent cu o instrucțiune de tip *case* unde condiția de test a fost inclusă în capul clauzei. La fel de bine s-ar fi putut spune și

```
r(X) :- X = 1, !, a, b, c.
r(X) :- X = 2, !, d.
r(X) :- X = 3, !, e.
r(_) :- write("Aici intra restul apelurilor").
```

**Notă.** Deci, următoarele secvențe sunt echivalente:

case X of	
v1: corp1;	predicat(X) :- X = v1, !, corp1.
v2: corp2;	predicat(X) :- X = v2, !, corp2.
...	...
else corp_else.	predicat(X) :- corp_else.

De asemenea, următoarele secvențe sunt echivalente:

if cond1 then	predicat(...) :-
corp1	cond1, !, corp1.
else if cond2 then	predicat(...) :-
corp2	cond2, !, corp2.
...	...
else	predicat(...) :-
corp_else.	corp_else.

## EXAMPLE

### 1. Fie următorul cod Prolog

<pre>% p(E: integer) % (o) – nedeterminist p(1). p(2). % q(E: integer) % (o) – nedeterminist q(3). q(4).</pre>	<pre>% r(E1: integer, E2: integer) % (o, o) – nedeterminist % V1 r(X, Y) :- !, p(X), q(Y). % V2 r(X, Y) :- p(X), !, q(Y). % V3 r(X, Y) :- p(X), q(Y), !.</pre>
--	--

Care sunt soluțiile furnizate la goal-ul

? *r*(X,Y).

în cele 3 variante (**V1**, **V2**, **V3**) de definire a preducatului *r*?

## R:

- V1 - sunt 4 soluții:

X=1 Y=3

X=1 Y=4

X=2 Y=3

X=2 Y=4

- V2 - sunt 2 soluții:

X=1 Y=3

X=1 Y=4

- V3 – e 1 soluție:

X=1 Y=3

## 2. Fie următorul cod Prolog

% p(L: list, S:integer)

% (i, o) –determinist

p([], 0).

p([H|T], S) :-

H > 0,

!,

p(T, S1),

S is S1 + H.

p([\_|T], S) :- p(T, S).

? p([1, 2, 3, 4], S).

S=10.

? p([1, -1, 2, -2], S).

S=3.

Ce se întâmplă dacă se mută tăietura înaintea condiției, adică dacă în cea de-a doua clauză se mută tăietura înaintea condiției?

% p(L: list, S:integer)

% (i, o) –determinist

p([], 0).

p([H|T], S) :-

!,

H > 0,

p(T, S1),

S is S1 + H.

p([\_|T], S) :- p(T, S).

? p([1, 2, 3, 4], S).

S=10.

? p([1, -1, 2, -2], S).

**false**

## 1.2 Predicatul “fail”

Valoarea lui *fail* este eșec. Prin aceasta el încurajează backtracking-ul. Efectul lui este același cu al unui predicat imposibil, de genul  $2 = 3$ . Fie următorul exemplu:

predicat(a, b).

predicat(c, d).

predicat(e, f).

toate :-

predicat(X, Y),



```

        write(X),write(Y),nl,
        fail.
toate1 :-
    predicat(X, Y),
    write(X),write(Y),nl.

```

Predicatele **toate** și **toate1** sunt fără parametri, și ca atare sistemul va trebui să răspundă dacă există X și Y astfel încât aceste predicate să aibă loc.

```

?- toate.
ab
cd
ef
false.

```

```

?-toate1.
ab
true;
cd
true;
ef
true.

```

```

?-predicat(X,Y).
X = a,
Y = b ;
X = c,
Y = d ;
X = e,
Y = f.

```

Faptul că apelul predicatului **toate** se termină cu *fail* (care eșuează întotdeauna) obligă Prolog să înceapă backtracking prin corpul regulii **toate**. Prolog va reveni până la ultimul apel care poate oferi mai multe soluții. Predicatul **write** nu poate da alte soluții, deci revine la apelul lui **predicat**.

### Observații.

- Acel **false** de la sfârșitul soluțiilor semnifică faptul că predicatul ‘toate’ nu a fost satisfăcut.
- După *fail* nu are rost să puneți nici un predicat, deoarece Prolog nu va ajunge să-l execute niciodată.

**Notă.** Secvențele următoare sunt echivalente:

```

cât timp condiție execută
    corp

```

```

predicat :-
    condiție,
    corp,
    fail.

```

### **EXEMPLU**

Fie următorul cod Prolog

```

% q(E: integer)
% (o) – nedeterminist

```

q(1).  
q(2).  
q(3).  
q(4).  
p :- q(I), I<3, write (I), nl, **fail**.

Care este rezultatul următoarei întrebări ?p. , în condițiile în care apare/nu apare fail la finalul ultimei clauze?

- cu **fail** la finalul clauzei, soluțiile sunt  
1  
2  
**false**
- fără **fail** la finalul clauzei, soluțiile sunt  
1  
true;  
2  
true;  
**false**

## 2. Exemple

**EXEMPLU 2.1** Să se scrie un predicat care concatenează două liste.

? concatene([1, 2], [3, 4], L).  
L = [1, 2, 3, 4].

Pentru combinarea listelor L1 si L2 pentru a forma lista L3 vom utiliza un algoritm recursiv de genul:

1. Dacă L1 = [] atunci L3 = L2.
2. Altfel, capul lui L3 este capul lui L1 și coada lui L3 se obține prin combinarea cozii lui L1 cu lista L2.

Să explicăm puțin acest algoritm. Fie listele L1 = [a<sub>1</sub>, ..., a<sub>n</sub>] și L2 = [b<sub>1</sub>, ..., b<sub>m</sub>]. Atunci lista L3 va trebui să fie L3 = [a<sub>1</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>] sau, dacă o separăm în cap și coadă, L3 = [a<sub>1</sub> | [a<sub>2</sub>, ..., a<sub>n</sub>, b<sub>1</sub>, ..., b<sub>m</sub>]]. De aici rezultă că:

1. capul listei L3 este capul listei L1;
2. coada listei L3 se obține prin concatenarea cozii listei L1 cu lista L2.

Mai departe, deoarece recursivitatea constă din reducerea complexității problemei prin scurtarea primei liste, rezultă că ieșirea din recursivitate va avea loc odată cu epuizarea listei L1, deci când L1 este []. De remarcat că condiția inversă, anume dacă L2 este [] atunci L3 este L1, este inutilă.

Programul SWI-Prolog este următorul:

## Model recursiv

$$\text{concatenare}(l_1 l_2 \dots l_n, l'_1 \dots l'_m) = \begin{cases} l'_1 & \text{daca } l = \emptyset \\ l_1 \oplus \text{concatenare}(l_2 \dots l_n, l'_1 \dots l'_m) & \text{altfel} \end{cases}$$

%( concatenare(L1: list, L2:list, L3:list)

% (i, i, o) - determinist

concatenare([], L, L).

concatenare([H|L1], L2, [H|L3]) :-

concatenare(L1, L2, L3).

Se observă că predicatul **concatenare** descris mai sus funcționează cu mai multe modele de flux, fiind nedeterminist în unele modele de flux, determinist în altele.

De exemplu, pentru întrebările

? concatenare(L1, L2, [1, 2, 3]).

/\* model de flux (o,o,i) - nedeterminist \*/

L1=[] l2=[1, 2, 3]

L1=[1] l2=[2, 3]

? concatenare(L, [3, 4], [1, 2, 3, 4]).

/\* model de flux (o,i,i) sau (i,o,i) - determinist\*/

L=[1, 2]

**EXEMPLU 2.2** Să se scrie un predicat care determină lista submulțimilor unei mulțimi reprezentate sub formă de listă.

? submulțimi([1, 2], L)

va produce L = [[], [2], [1], [1, 2]]

**Observație:** Dacă lista e vidă, submulțimea sa e lista vidă. Pentru determinarea submulțimilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

- i. determină o submulțime a listei L
- ii. plasează elementul E pe prima poziție într-o submulțime a listei L

$\text{subm}(l_1, l_2, \dots, l_n) =$

1.  $\emptyset$                       *daca l e vida*
2.  $\text{subm}(l_2, \dots, l_n)$
3.  $l_1 \oplus \text{subm}(l_2, \dots, l_n)$

Vom folosi predicatul nedeterminist **subm** care va genera submulțimile, după care vor fi colectate toate soluțiile acestuia, folosind predicatul **findall**.

Codul SWI-Prolog este indicat mai jos

```
% subm(L: list, Subm:list)
% (i, o) - nedeterminist
subm([], []).
subm([_|T], S) :- subm(T, S).
subm([H|T], [H|S]) :- subm(T, S).

% submultimi(L: list, LRez:list of lists)
% (i, o) - determinist
submultimi(L, LRez):-findall(S, subm(L,S), LRez).
```

### **EXEMPLU 2.3**

<pre>% sP(L:list of numbers, L: list of number) % (i,o) – nondeterm sP([], []). sP ([_ T],S):-sP(T,S). sP ([H T],[H S]):- H mod 2 =:=0,                     !,                     sP(T,S). sP ([H T],[H S]):-sI(T,S).</pre>	<pre>% sI(L:list of numbers, L: list of number) % (i,o) – nondeterm sI([H],[H]):-H mod 2 =\=0, !. sI([_ T],S):-sI(T,S). sI([H T],[H S]):-H mod 2 =:=0,                     !,                     sI(T,S). sI([H T],[H S]):-sP(T,S).</pre>
--	--

? sP([1, 2, 3], S).

[]  
[2]  
[1, 3]  
[1, 2, 3]

?sI([1, 2, 3], S).

[3]  
[2, 3]  
[1]  
[1, 2]  
**false**

**EXEMPLU 2.4** Fie următoarele definiții de predicate. Care este efectul următoarei interogări?

?- det([1,2,1,3,1,7,8],1,L).

```
% det(L:list of elements, E:element, LRez: list of numbers)
%(i, i, o) - determinist
det(L, E, LRez):- det_aux(L, E, LRez, 1).
```

```
% det_aux( L:list of elements, E:element, LRez: list of numbers, P:intreg)
% (i, i, o, i) - determinist
```

```

det_aux([], _, [], _).
det_aux([E|T], E, [P|LRez], P) :-
    !,
    P1 is P+1,
    det_aux(T, E, LRez, P1).
det_aux([_|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, LRez, P1).

```

**Soluția pentru refactorizare** (secvență marcată cu albastru)

- folosirea unui predicat auxiliar

```

det_aux([], _, [], _).
det_aux([H|T], E, LRez, P) :-
    P1 is P+1,
    det_aux(T, E, L, P1),
    prel(H, E, P, L, LRez).

```

```

% prel(H: element, E:element, P: number, L:list, LRez: list of numbers)
% (i, i, i, i, o) - determinist
prel(E, E, P, L, [P|L]) :- !.
prel(_, _, _, L, L).

```

**EXEMPLU 2.5** Fie următoarele definiții de predicate.

```

% g(L:list, E: element, LRez: list)
% (i, i, o) – nedeterminist
g([H|_], E, [E,H]).
g([_|T], E, P):-
    g(T, E, P).

```

```

% f(L:list, LRez: list)
% (i, o) – nedeterminist
f([H|T],P):-
    g(T, H, P).
f([_|T], P):-
    f(T, P).

```

- Care este efectul următoarelor interogări? Apare **false** la finalul căutării?

```

? g([1, 3, 4], 2, P).
? f([1,2,3,4], P).

```

- Funcționează predicatele în alte modele de flux?

```

?- g([1,2,3], E, [4,1]).

```

## CURS 5

### Nedeterminism. Backtracking

#### Cuprins

1. Exemple predicate nedeterministe (continuare).....	1
2. Backtracking .....	5

% depanare cod Prolog  
% trace.  
% notrace.

#### 1. Exemple predicate nedeterministe (continuare)

**EXEMPLU 3.1** Să se scrie un predicat nedeterminist care generează combinații cu  $k$  elemente dintr-o mulțime nevidă reprezentată sub forma unei liste.

```
? comb([1, 2, 3], 2, C). /*model de flux (i, i, o) - nedeterminist*/  
C = [2, 3];  
C = [1, 2];  
C = [1, 3].
```

**Observație:** Pentru determinarea combinațiilor unei liste  $[E|L]$  (care are capul  $E$  și coada  $L$ ) luate câte  $K$ , sunt următoarele cazuri posibile:

- dacă  $K=1$ , atunci o combinație este chiar  $[E]$
- determină o combinație cu  $K$  elemente a listei  $L$ ;
- plasează elementul  $E$  pe prima poziție în combinațiile cu  $K-1$  elemente ale listei  $L$  (dacă  $K>1$ ).

Modelul recursiv pentru generare este:

$comb(l_1 l_2 \dots l_n, k) =$

- $(l_1)$   $daca k = 1$
- $comb(l_2 \dots l_n, k)$
- $l_1 \oplus comb(l_2 \dots l_n, k - 1)$   $daca k > 1$

Vom folosi predicatul nedeterminist **comb** care va genera toate combinările. Dacă se dorește colectarea combinărilor într-o listă, se va putea folosi predicatul **findall**.

Programul SWI-Prolog este următorul:

```
% comb(L: list, K:integer, C:list)
% (i, i, o) - nedeterminist
comb([H|_], 1, [H]).
comb([_|T], K, C) :-
    comb(T, K, C).
comb([H|T], K, [H|C]) :-
    K > 1,
    K1 is K-1,
    comb(T, K1, C).
```

**EXEMPLU 3.2** Să se scrie un predicat nedeterminist care inserează un element, pe toate pozițiile, într-o listă.

```
? insereaza(1, [2, 3], L).    /*model de flux (i, i, o) - nedeterminist*/
L = [1, 2, 3];
L = [2, 1, 3];
L = [2, 3, 1].
```

### Model recursiv

$insereaza(e, l_1 l_2 \dots l_n) =$

1.  $e \oplus l_1 l_2 \dots l_n$
2.  $l_1 \oplus insereaza(e, l_2 \dots l_n)$

```
% insereaza(E: element, L:List, LRez:list)
% (i, i, o) - nedeterminist
insereaza(E, L, [E|L]).
insereaza(E, [H|T], [H|Rez]) :-
    insereaza(E, T, Rez).
```

Observăm că, pe lângă modelul de flux (i, i, o) descris anterior, predicatul **insereaza** funcționează cu mai multe modele de flux (în unele modele de flux preducatorul fiind determinist, în altele nedeterminist).

- $insereaza(E, L, [1, 2, 3])$ , cu modelul de flux (o, o, i) și soluțiile  
 $E=1, L = [2, 3]$   
 $E=2, L = [1, 3]$

- $E=3, L = [1, 2]$   
insereaza(1, L, [1, 2, 3]), cu modelul de flux (i, o, i) și soluția  
 $L = [2, 3]$
- insereaza(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) și soluția  
 $E = 2$

**EXEMPLU 3.3** Să se scrie un predicat nedeterminist care șterge un element, pe rând, de pe toate pozițiile pe care acesta apare într-o listă.

```
? elimin(1, L, [1, 2, 1, 3]). /*model de flux (i, o, i) - nedeterminist*/
L = [2, 1, 3];
L = [1, 2, 3];
```

$elimin(e, l_1 l_2 \dots l_n) =$

1.  $l_2 \dots l_n$       *daca*  $e = l_1$
2.  $l_1 \oplus elimin(e, l_2 \dots l_n)$

% elimin(E: element, LRez:list, L:list)

% (i, o, i) – nedeterminist

elimin(E, L, [E|L]).

elimin(E, [A|L], [A|X]) :-

elimin(E, L, X).

Observăm că predicatul **elimin** funcționează cu mai multe modele de flux. Astfel, următoarele întrebări sunt valide:

- elimin(E, L, [1, 2, 3]), cu modelul de flux (o, o, i) și soluțiile  
 $E=1, L = [2, 3]$   
 $E=2, L = [1, 3]$   
 $E=3, L = [1, 2]$
- elimin(1, [2, 3], L), cu modelul de flux (i, i, o) și soluțiile  
 $L = [1, 2, 3]$   
 $L = [2, 1, 3]$   
 $L = [2, 3, 1]$
- elimin(E, [1, 3], [1, 2, 3]), cu modelul de flux (o, i, i) și soluția  
 $E = 2$

**EXEMPLU 3.4** Să se scrie un predicat nedeterminist care generează permutările unei liste.

```
? perm([1, 2, 3], P). /*model de flux (i, o,) - nedeterminist*/
P = [1, 2, 3];
P = [1, 3, 2];
P = [2, 1, 3];
P = [2, 3, 1];
P = [3, 1, 2];
P = [3, 2, 1]
```



Cum obținem permutările listei [1, 2, 3] dacă știm să generăm permutările sublistei [2, 3] (adică [2, 3] și [3, 2])?

Pentru determinarea permutărilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

1. determină o permutare L1 a listei L;
2. plasează elementul E pe toate pozițiile listei L1 și produce în acest fel lista X care va fi o permutare a listei inițiale [E|L].

Modelul recursiv este:

$perm(l_1 l_2 \dots l_n) =$

1.  $\emptyset$  *daca l e vida*
2. *insereaza*( $l_1, perm(l_2 \dots l_n)$ ) *altfel*

% perm(L:list, LRez:list)

% (i, o) – nedeterminist

perm([], []).

perm([E|T], P) :-

perm(T, L),

insereaza(E, L, P). % (i, i, o)

% alternativa pentru clauza 2

perm(L, [H|T]) :-

elimin(H, Z, L), % (o o, i)

perm(Z, T).

% alternativa pentru clauza 2

perm(L, [H|T]) :-

insereaza(H, Z, L), % (o o, i)

perm(Z, T).

% alternativa pentru clauza 2

perm([E|T], P) :-

perm(T, L),

elimin(E, L, P), % (i, i, o)

**TEMĂ** Să se scrie un predicat nedeterminist care generează aranjamente cu  $k$  elemente dintr-o mulțime nevidă reprezentată sub forma unei liste.

? aranj([1, 2, 3], 2, A). /\*model de flux (i, i, o) - nedeterminist\*/

A = [2, 3];

A = [3, 2];

A = [1, 2];

A = [2, 1] ;

$A = [1, 3];$   
 $A = [3, 1];$

## 2. Backtracking

Metoda **backtracking** (căutare cu revenire)

- aplicabilă, în general, unor probleme ce au mai multe soluții.
- permite generarea tuturor soluțiilor unei probleme.
- căutare în adâncime limitată (*depth limited search*) în spațiul soluțiilor problemei
- exponențială ca și timp de execuție
- metodă generală de rezolvare a problemelor din clasa *Constraint Satisfaction Problems*, (CSP)
- Prolog-ul este potrivit pentru rezolvarea problemelor din clasa CSP
  - **structura de control** folosită de interpretorul Prolog se bazează pe backtracking

### Formalizare

- soluția problemei este un vector/listă  $(x_1 x_2 \dots x_n)$  ,  $x_i \in D_i$
- vectorul soluție se generează incremental
- notăm cu *col* lista care colectează o soluție a problemei
- notăm cu **condiții-finale** funcția care verifică dacă lista *col* e o soluție a problemei
- notăm cu **condiții-continuare** funcția care verifică dacă lista *col* poate conduce la o soluție a problemei

Pentru determinarea unei soluții a problemei, sunt următoarele cazuri posibile:

- i. dacă *col* verifică **condiții-finale**, atunci e o soluție a problemei;
- ii. (altfel) se completează *col* cu un element *e* (pe care îl vom numi **candidat**) astfel încât  $col \cup e$  să verifice **condiții-continuare**

### **Observație**

- dacă generarea elementului *e* cu care se va complete colectoarea *col* la punctul ii. este nedeterministă, atunci se vor putea genera toate soluțiile problemei.
- de menționat faptul că sunt probleme în care nu se impun **condiții-finale**

Modelul recursiv general pentru generarea soluțiilor este:

*solutie(col)* =

1. *col*                      *daca* **conditii – finale**(*col*)

2.  
*solutie(col ∪ e)*    *daca* **conditii – continuare**(*col ∪ e*), *e* fiind un posibil candidat

**EXEMPLU 1.1** Să se scrie un predicat nedeterminist care generează combinații cu  $k \neq 1$  elemente dintr-o mulțime nevidă ale cărei elemente sunt numere naturale nenule pozitive, astfel încât suma elementelor din combinație să fie o valoare  $S$  dată.

```
? combSuma([3, 2, 7, 5, 1, 6], 3, 9, C).    /* model de flux (i, i, i, o) – nedeterminist */
C = [2, 1, 6];                             /* k=3, S=9 */
C = [3, 5, 1]
```

!!! **false** la final?

```
? toateCombSuma([3, 2, 7, 5, 1, 6], 2, 9, LC).
LC=[[2, 1, 6], [3, 5, 1]].
```

Pentru rezolvarea acestei probleme, vom da 3 variante de rezolvare, ultima dintre ele fiind bazată pe metoda backtracking descrisă anterior.

**VARIANTA 1** Generăm soluțiile problemei direct recursiv

Pentru determinarea combinațiilor unei liste  $[H|L]$  (care are capul  $H$  și coada  $L$ ) luate câte  $K$ , de sumă dată  $S$  sunt următoarele cazuri posibile:

- i. dacă  $K=1$  și  $H$  este egal cu  $S$ , atunci o combinație este chiar  $[H]$
- i. determină o combinație cu  $K$  elemente a listei  $L$ , având suma  $H$ ;
- ii. plasează elementul  $H$  pe prima poziție în combinațiile cu  $K-1$  elemente ale listei  $L$ , de sumă  $S-H$  (dacă  $K>1$  și  $S-H>0$ ).

Modelul recursiv pentru generare este:

```
combSuma( $l_1 l_2 \dots l_n, k, S$ ) =
```

1.  $(l_1)$  dacă  $k = 1$  și  $l_1 = S$
2.  $comb(l_2 \dots l_n, k, S)$
3.  $l_1 \oplus comb(l_2 \dots l_n, k-1, S-l_1)$  dacă  $k > 1$  și  $S-l_1 > 0$

Vom folosi predicatul nedeterminist **combSuma** care va genera toate combinațiile. Dacă se dorește colectarea combinațiilor într-o listă, se va putea folosi predicatul **findall**.

Programul SWI-Prolog este următorul:

```
% combSuma(L: list, K:integer, S: integer, C:list)
% (i, i, i, o) - nedeterminist
combSuma([H|_], 1, H, [H]).
combSuma([_|T], K, S, C) :-
    combSuma(T, K, S, C).
combSuma([H|T], K, S, [H|C]) :-
    K>1,
    S1 is S-H,
    S1>0,
```

K1 is K-1,  
combSuma(T, K1, S1, C).

```
% toateCombSuma (L: list, K:integer, S: integer, LC:list of lists)
```

% (i, i, i, o) - determinist

toateCombSuma(L, K, S, LC) :-

$$\text{findall}(C, \text{combSuma}(L, K, S, C), LC).$$

**VARIANTA 2** Folosim un predicat neterminist

**candidat**(E:element, L:list) - model de flux **(o,i)**

care generează, pe rând, câte un element al unei liste. O soluție a acestui predicat va fi un element care poate fi adăugat în soluție.

? candidat(E, [1, 2,3]).

E=1;

$$E=2;$$

E=3

$$candidat(l_1l_2, \dots, l_n) =$$

1.  $l_1$  *daca  $l$  e nevida*

2. *candidat*( $l_2 \dots l_n$ )

```
% candidat(E: element, L:list)
```

% (o, i) - nedeterminist

candidat(E,[E|\_]).

candidat(E,[\_|T]) :-

candidat(E,T).

Predicatul de bază va genera un candidat E și va începe generarea soluției cu acest element.

```
% combSuma(L:list, K:integer, S:integer, C: list)
```

% (i, i, i, o) - nedeterminist

combSuma(L, K, S, C) :-

$$\text{candidat}(\mathbf{E}, \mathbf{L}),$$
$$E < S,$$

combaux(L, K, S, C, 1, E, [E]).

Predicatul auxiliar nedeterminist **comb\_aux** va genera câte o combinaire de sumă dată.

$$\text{combaux}(l, k, s, lg, \text{sum}, \text{col}) =$$

1. *col* *daca* (*sum* = *s* și *k* = *lg*)

2.

$combaux(l, k, s, lg + 1, sum + e, e \cup col)$     *daca*  $(sum \neq s \text{ sau } k \neq lg)$  și  $(lg < k)$  și

$(e = candidat(l_1, l_2, \dots, l_n))$  și  $(e < col_1)$  și  $(sum + e \leq s)$

%  $combaux(L:list, K:integer, S:integer, C: list, Lg:integer, Sum:integer, Col:list)$

% (i, i, i, o, i, i, i) - *nedeterminist*

$combaux(\_, K, S, C, K, S, C) :- !.$

$combaux(L, K, S, C, Lg, Sum, [H|T]) :-$

$Lg < K,$

$candidat(E, L),$

$E < H,$

$Sum1 \text{ is } Sum + E,$

$Sum1 \leq S,$

$Lg1 \text{ is } Lg + 1,$

$combaux(L, K, S, C, Lg1, Sum1, [E|H|T]).$

% *alternativa la clauza II – refactorizare - un predicat auxiliar pentru verificarea conditiei*

%  $conditie(L:list, K:integer, S:integer, Lg:integer, Sum:integer, H:integer, E:integer)$

% (i, i, i, i, i, i, o) - *nedeterminist*

$conditie(L, K, S, Lg, Sum, H, E) :-$

$Lg < K,$

$candidat(E, L),$

$E < H,$

$Sum1 \text{ is } Sum + E,$

$Sum1 \leq S.$

$combaux(L, K, S, C, Lg, Sum, [H|T]) :-$

$conditie(L, K, S, Lg, Sum, H, E),$

$Lg1 \text{ is } Lg + 1,$

$Sum1 \text{ is } Sum + E,$

$combaux(L, K, S, C, Lg1, Sum1, [E|H|T]).$

**VARIANTA 3** Se generează combinările cu  $k$  elemente și apoi se verifică dacă suma unei combinări este  $S$ . Această soluție este varianta **brute force** (GTS – *generate and test*), având în vedere că se generează (în plus) combinări care e posibil să nu fie de sumă  $S$ .

%  $combSuma(L: list, K:integer, S: integer, C:list)$

% (i, i, i, o) - *nedeterminist*

$combSuma(L, K, S, C) :-$

$comb(L, K, C),$

$suma(C, S).$

- predicatul **comb** pentru generarea unei combinări cu  $k$  elemente dintr-o listă.
- predicatul **suma** ( $L$ :list of numbers,  $S$ :integer), model de flux (i, i) care verifică dacă suma elementelor listei  $L$  este egală cu  $S$ .

**NOTĂ** *Această variantă nu e acceptată la examen.*

**EXEMPLU 1.2** Se dă o listă cu elemente numere întregi distincte. Să se genereze toate submulțimile având cel puțin 2 elemente, cu elemente în ordine strict crescătoare.

```
? submCresc([2, 1, 3], S).    /* model de flux (i, o) – nedeterminist */
S = [1, 3];
S = [2, 3];
S = [1, 2];
S = [1, 2, 3];
```

**EXEMPLU 1.3** Dându-se o mulțime reprezentată sub formă de listă, se cere să se genereze submulțimi de sumă pară formate doar din numere impare.

```
? submSP([1, 2, 3, 4, 5], S).    /* model de flux (i, o) – nedeterminist */
S = [1, 3];
S = [1, 5];
S = [3, 5];
```

**EXEMPLU 1.4** Dându-se două valori naturale  $n$  ( $n > 2$ ) și  $v$  ( $v$  nenul), se cere un predicat Prolog care determină permutările elementelor  $1, 2, \dots, n$  cu proprietatea că orice două elemente consecutive au diferența în valoare absolută mai mare sau egală cu  $v$ .

```
? permutari(4, 2, P)           (n=4, v=2)
P = [2, 4, 1, 3];
P = [3, 1, 4, 2];
....
```

```
? candidat(4, I).
I=4;
I=3;
I=2;
I=1
```

### **Modele recursive:**

*candidat*( $n$ ) =

1.  $n$
2. *candidat*( $n - 1$ ) *daca*  $n > 1$

Se vor folosi următoarele predicate

- predicatul nedeterminist **candidat**( $N, I$ ) (**i, o**) generează un candidat la soluție (o valoare între 1 și  $N$ );
- predicatul nedeterminist **permutari\_aux**( $N, V, LRezultat, Lungime, LColectoare$ ) (**i, i, o, i, i**) colectează elementele unei permutări în **LColectoare** de lungime **Lungime**. Colectarea se

va opri atunci când numărul de elemente colectate (**Lungime**) este N. În acest caz, **LColectoare** va conține o permutare soluție, iar **LRezultat** va fi legată de **LColectoare**.

- predicatul nedeterminist **permutari**(N, V, L) (**i, i, o**) generează o permutare soluție;
- predicatul **apare**(E, L) care testează apartenența unui element la o listă (pentru a colecta în soluție doar elementele distincte).

Folosim un predicat neterminist **candidat**(N:intreg, I:intreg), model de flux (i,o), care generează, pe rând, elementele N, N-1,...,1.

```
% candidat(N:integer, I:integer)
% (i,o) - nedeterminist
candidat(N, N).
candidat(N, I) :-
    N>1,
    N1 is N-1,
    candidat(N1,I).
% permutari(N:integer, V:integer, L:list)
% (i,i,o) - nedeterminist
permutari(N, V, L) :-
    candidat(N, I),
    permutari_aux(N, V, L, 1, [I]).
% permutari_aux(N:integer, V:integer, L:list, Lg:integer, Col:list)
% (i,i,o,i,i) - nedeterminist
permutari_aux(N, _, Col, N, Col) :- !.
permutari_aux(N, V, L, Lg, [H|T]) :-
    candidat(N, I),
    abs(H-I)>=V,
    \+ apare(I, [H|T]), % (i,i)
    Lg1 is Lg+1,
    permutari_aux(N, V, L, Lg1, [I|H|T]).
```

**EXEMPLU 1.5** Se dă o mulțime de numere naturale nenule reprezentată sub forma unei liste. Să se determine toate posibilitățile de a scrie un număr N dat sub forma unei sume a elementelor din listă.

```
% list=integer*
% candidat(list, integer) (i, o) - nedeterminist
% un element posibil a fi adaugat in lista solutie

candidat([E|_],E).
candidat([_|T],E):-candidat(T,E).

% solutie(list,integer,list) (i,i,o) nedeterminist
% solutie_aux(list,integer,list,list,integer) (i,i,o,i,i) nedeterminist
% al patrulea argument colecteaza solutia, al cincilea argument
```

% reprezinta suma elementelor din colectoare

```
solutie(L, N, Rez) :-  
    candidat(L, E),  
    E =< N,  
    solutie_aux(L, N, Rez, [E], E).
```

```
solutie_aux(_, N, Rez, Rez, N):-!.  
solutie_aux(L, N, Rez, [H | Col], S) :-  
    candidat(L, E),  
    E < H,  
    S1 is S+E,  
    S1 =< N,  
    solutie_aux(L, N, Rez, [E | [H | Col]], S1).
```

### EXEMPLU 1.6 PROBLEMA CELOR 3 CASE.

1. Englezul locuiește în prima casă din stânga.
2. În casa imediat din dreapta celei în care se află lupul se fumează Lucky Strike.
3. Spaniolul fumează Kent.
4. Rusul are cal.

Cine fumează LM? Al cui este câinele?

Se observă că problema are două soluții:

I.	englez	câine	LM
	spaniol	lup	Kent
	rus	cal	LS
II.	englez	lup	LM
	rus	cal	LS
	spaniol	câine	Kent

Este o problemă de satisfacere a limitărilor (**constraint satisfaction**).

Codificăm datele problemei și observăm că o soluție e formată din triplete de forma (N, A, T) unde:

N	aparține mulțimii	[eng, spa, rus]
A	aparține mulțimii	[caine, lup, cal]
T	aparține mulțimii	[lm, ls, ken]

Vom folosi următoarele predicate:

- predicatul nedeterminist **rezolva**(N, A, T) (**o, o, o**) care generează o soluție a problemei
- predicatul nedeterminist **candidati**(N, A, T) (**o, o, o**) care generează toți candidații la soluție
- predicatul determinist **restricții**(N, A, T) (**i, i, i**) care verifică dacă un candidat la soluție satisface restricțiile impuse de problemă
- predicatul nedeterminist **perm**(L, L1) (**i, o**) care generează permutările listei L



```
SWI-Prolog -- d:/Gabi/gabi/FACULTAT/2014-2015/PLF/DOC/Exemple_SWI/exemple.pl
File Edit Settings Run Debug Help
% library(win_menu) compiled into win_menu 0.00 sec, 29 clauses
% c:/users/istvan/appdata/roaming/swi-prolog/pl.ini compiled 0.00 sec, 1 clauses
% d:/Gabi/gabi/FACULTAT/2014-2015/PLF/DOC/Exemple_SWI/exemple.pl compiled 0.00 sec, 95 clauses
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version 6.2.0)
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- rezolva(N,A,T).
N = [eng, spa, rus],
A = [caine, lup, cal],
T = [lm, kent, ls] ;
N = [eng, rus, spa],
A = [lup, cal, caine],
T = [lm, ls, kent] ;
false.

2 ?- █
```

**% rezolva - (o,o,o)**

**% candidati - (o,o,o)**

**% restrictii - (i,i,i)**

rezolva(N, A, T) :-

    candidati(N, A, T),

    restrictii(N, A, T).

candidati(N, A, T) :-

    perm([eng, spa, rus], N),

    perm([caine, lup, cal], A),

    perm([lm, kent, ls], T).

restrictii(N, A, T) :-

    aux(N, A, T, eng, \_, \_, 1),

    aux(N, A, T, \_, lup, \_, Nr),

    dreapta(Nr, M),

    aux(N, A, T, \_, \_, ls, M),

    aux(N, A, T, spa, \_, kent, \_),

    aux(N, A, T, rus, cal, \_, \_).

**% dreapta - (i,o)**

dreapta(I,J) :- J is I+1.

**% aux (i,i,i,o,o,o,o)**

aux([N1, \_, \_], [A1, \_, \_], [T1, \_, \_], N1, A1, T1, 1).

aux([\_, N2, \_], [\_, A2, \_], [\_, T2, \_], N2, A2, T2, 2).

aux([\_, \_, N3], [\_, \_, A3], [\_, \_, T3], N3, A3, T3, 3).

**% insereaza (i,io)**

insereaza(E, L, [E|L]).

insereaza(E, [H|L], [H|T]) :- insereaza(E,L,T).

**% perm (i,o)**

perm([], []).

perm([H|T], L):-perm(T, P),insereaza(H, P, L).

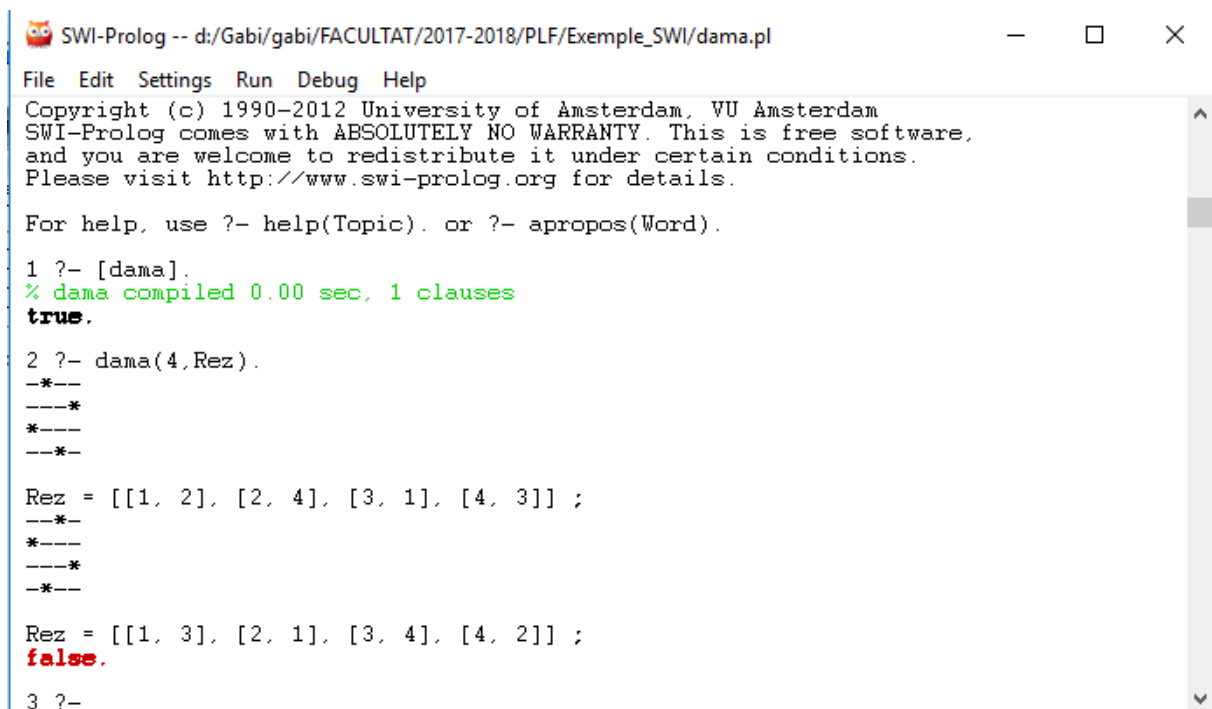
### PROBLEMA CELOR 5 CASE.

5 people live in the five houses in a street. Each has a different profession, animal, favorite drink, and each house is a different color.

1. The Englishman lives in the red house
2. The Spaniard owns a dog
3. The Norwegian lives in the first house on the left
4. The Japanese is a painter
5. The green house is on the right of the white one
6. The Italian drinks tea
7. The fox is in a house next to the doctor
8. Milk is drunk in the middle house
9. The horse is in a house next to the diplomat
10. The violinist drinks fruit juice
11. The Norwegians house is next to the blue one
12. The sculptor breeds snails
13. The owner of the green house drinks coffee
14. The diplomat lives in the yellow house

Who owns the zebra? Who drinks water?

**EXEMPLU 1.7** Să se dispună N dame pe o tablă de șah NxN, încât să nu se atace reciproc.



```
SWI-Prolog -- d:/Gabi/gabi/FACULTAT/2017-2018/PLF/Exemple_SWI/dama.pl
File Edit Settings Run Debug Help
Copyright (c) 1990-2012 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

1 ?- [dama].
% dama compiled 0.00 sec, 1 clauses
true.

2 ?- dama(4,Rez).
*--
*--
*--
*--
Rez = [[1, 2], [2, 4], [3, 1], [4, 3]] ;
*--
*--
*--
Rez = [[1, 3], [2, 1], [3, 4], [4, 2]] ;
false.

3 ?-
```

**% (integer,list\*) – (i, o) nedeterm.**

```
dama(N, Rez) :-  
    candidat(E, N),  
    dama_aux(N, Rez, [[N, E]], N),  
    tipar(N, Rez).
```

**% (integer,integer) – (o, i) nedeterm.**

```
candidat(N, N).  
candidat(E, I) :-  
    I>1,  
    I1 is I-1,  
    candidat(E, I1).
```

**% (integer,list\*,list\*,integer) – (i,o, i, i) nedeterm.**

```
dama_aux(_, Rez, Rez, 1) :- !.  
dama_aux(N, Rez, C, Lin) :-  
    candidat(Col1, N),  
    Lin1 is Lin-1,  
    valid(Lin1, Col1, C),  
    dama_aux(N, Rez, [[Lin1, Col1] | C], Lin1).
```

**% (integer,integer,list\*) – (i,i, i) determ.**

```
valid(_, _, []).  
valid(Lin, Col, [[Lin1,Col1] | T]) :-  
    Col \= Col1,  
    DLin is Col-Col1,  
    DCol is Lin-Lin1,  
    abs(DLin) \= abs(DCol),  
    valid(Lin, Col, T).
```

**% (integer, list\*) – (i,i) determ.**

```
tipar(_, []) :- nl.  
tipar(N, [[_, Col] | T]) :-  
    tipLinie(N, Col),  
    tipar(N, T).
```

**% (integer, char) – (i,o) determ.**

```
caracter(1, '*') :- !.  
caracter(_, '-').
```

**% (integer, list\*) – (i,i) determ.**

```
tipLinie(0, _) :- nl, !.  
tipLinie(N, Col) :-  
    caracter(Col, C),  
    write(C),  
    N1 is N-1,  
    Col1 is Col-1,  
    tipLinie(N1, Col1).
```

## CURS 6

### Evitare apeluri recursive repetate. Obiecte compuse.

#### Recursivitate de coadă

#### Cuprins

1. Evitare apeluri recursive repetate.....	1
2. Obiecte simple și obiecte compuse. ....	3
3. Optimizarea prin recursivitate de coadă (tail recursion).....	6
Funcționarea recursivității de coadă .....	6
Utilizarea tăieturii pentru păstrarea recursivității de coadă .....	8

### 1. Evitare apeluri recursive repetate

**EXEMPLU 1.1** Să se calculeze minimul unei liste de numere întregi.

```
% minim(L: list of integer, M:integer)
% (i, o) - determinist
minim([A], A).
minim([H|T], Rez) :-
    minim(T, M),
    H =< M, !, Rez=H.
minim([_|T], M) :-
    minim(T, M).
```

**Soluția 1.** (generală) Se folosește un predicat auxiliar, pentru a evita apelul (recursiv) repetat din clauzele 2 și 3.

```
minim([A], A).
minim([H|T], Rez) :-
    minim(T, M), aux(H, M, Rez).
% aux(H: integer, M:integer, Rez:integer)
% (i, i, o) - determinist
aux(H, M, Rez) :-
    H =< M, !, Rez=H.
aux(_, M, M).
```

**Soluția 2** (specifică)

```
minim([A], A).
minim([H|T], M) :-
```

```

    minim(T, M),
    H > M, !.
minim([H|_], H).

```

**EXEMPLU 1.2** Se dă o listă numerică. Să se dea o soluție pentru evitarea apelului recursiv repetat. *Nu se vor redefini clauzele.*

```

% f(L:list of numbers, E: number)
% (i,o) – determinist
f([E],E).
f([H|T],Y):- f(T,X),
              H>=X,
              !,
              Y=H.
f([_|T],X):- f(T,X).

```

**Soluție.** Se folosește un predicat auxiliar. Soluția nu presupune înțelegerea semanticii codului.

```

f([E],E).
f([H|T],Y):- f(T,X), aux(H, X, Y).
% aux(H: integer, X:integer, Y:integer)
% (i, i, o) - determinist
aux(H, X, Y) :-
    H>=X,
    !,
    Y=H.
aux(_, X, X).

```

**EXEMPLU 1.3** Să se dea o soluție pentru evitarea apelului recursiv repetat.

```

% f(K:number, X:number)
% (i,o) – determinist
f(1,1):-!.
f(2,2):-!.
f(K,X):- K1 is K-1,
          f(K1, Y),
          Y>1,
          !,
          K2 is K-2,
          X=K2.
f(K,X):- K1 is K-1,
          f(K1, X).

```

**Soluție.** Se folosește un predicat auxiliar.

```
f(1,1):-!.  
f(2,2):-!.  
f(K,X):- K1 is K-1,  
         f(K1, Y),  
         aux(K, Y, X).  
% aux(K: integer, Y:integer, X:integer)  
% (i, i, o) - determinist  
aux(K, Y, X):-  
    Y>1,  
    !,  
    K2 is K-2,  
    X=K2.  
aux(_, Y, Y).
```

## 2. Obiecte simple și obiecte compuse.

### Obiecte simple

Un obiect simplu este fie o variabilă, fie o constantă. O constantă este fie un caracter, fie un număr, fie un atom (simbol sau string).

Variabilele Prolog sunt locale, nu globale. Adică, dacă două clauze conțin fiecare câte o variabilă numită X, cele două variabile sunt distincte și, de obicei, nu au efect una asupra celeilalte.

### Obiecte compuse și functori

**Obiectele compuse** ne permit să tratăm mai multe informații ca pe un singur element, într-un astfel de mod încât să-l putem utiliza și pe bucăți. Fie, de exemplu, data de *2 februarie 1998*. Constă din trei informații, ziua, luna și anul, dar e util să o tratăm ca un singur obiect cu o structură arborescentă:

```
      DATA  
    /  |  \  
2 februarie 1998
```

Acest lucru se poate face scriind obiectul compus astfel:

```
data(2, "februarie", 1998)
```

Aceasta seamănă cu un fapt Prolog, dar nu este decât un obiect (o dată) pe care îl putem manevra la fel ca pe un simbol sau număr. Din punct de vedere sintactic, începe cu un nume (sau **functor**, în acest caz cuvântul **data**) urmat de trei argumente.

**Notă.** Functorul în Prolog nu este același lucru cu funcția din alte limbaje de programare. Este doar un nume care identifică un tip de date compuse și care ține argumentele laolaltă.

Argumentele unei date compuse pot fi chiar ele compuse. Iată un exemplu:

naștere(persoana("Ioan", "Popescu"), data(2, "februarie", 1918))

## Unificarea obiectelor compuse

Un obiect compus se poate unifica fie cu o variabilă simplă, fie cu un obiect compus care se potrivește cu el. De exemplu,

data(2, "februarie", 1998)

se potrivește cu variabila liberă X și are ca rezultat legarea lui X de data(...). De asemenea, obiectul compus de mai sus se potrivește și cu

data(Zi, Lu, An)

și are ca rezultat legarea variabilei Zi de valoarea 2, a variabilei Lu de valoarea "februarie" și a variabilei An de valoarea 1998.

### Observații

- Declarație pentru *specificarea* unui domeniu cu alternative  
    % domeniu = alternativa<sub>1</sub>(dom, dom, ..., dom);  
    %           alternativa<sub>2</sub>(dom, dom, ..., dom);  
    %           ...  
▪ Functorii pot fi folosiți pentru controla argumentele care pot avea tipuri multiple  
    % element = i(integer); r(real); s(string)

## 2.1 Liste eterogene

Se dă o listă de numere întregi sau/și simboluri. Se cere suma numerelor întregi pare din listă.

?- suma([i(1),s(a),i(2)],S).  
S = 2.

?- suma([s(a),s(b),i(1)],S).  
S=0.

% suma(L: heterogeneous list, S: number)

% (i,o) - determ

suma([],0).

sumalist([i(H)|T], S) :-

    H mod 2 =:=0, !,

    suma(T,S1),

    S is S1+H.

% atenție la o clauză de forma suma ([i(\_)|T], S) :- suma(T,S).

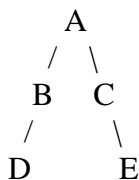
suma ([\_|T], S) :- suma(T,S).

## 2.2 Arbori binari

Folosind obiecte compuse, se pot defini și prelucra în Prolog diverse structuri de date, precum arborii.

```
% domeniul corespunzător AB – domeniu cu alternative
% arbore=arb(integer, arbore, arbore);nil
% Functorul nil îl asociem arborelui vid
```

De exemplu, arborele



se va reprezenta astfel

```
arb(A, arb(B,
           arb(D, nil, nil),
           nil
         ),
     arb(C, nil,
          arb(E, nil, nil)
        )
    )
```

Se dă o listă numerică. Se cere să se afișeze elementele listei în ordine crescătoare. Se va folosi sortarea arborescentă (folosind un ABC).

**Indicație.** Se va construi un ABC cu elementele listei. Apoi, se va parcurge ABC în inordine.

```
% domeniul corespunzător ABC – domeniu cu alternative
% arbore=arb(integer, arbore, arbore);nil
% Functorul nil îl asociem arborelui vid
```

$$inserare(e, arb(r, s, d)) = \begin{cases} arb(e, \emptyset, \emptyset) & \text{daca } arb(r, s, d) \text{ e vid} \\ arb(r, inserare(e, s), d) & \text{daca } e \leq r \\ arb(r, s, inserare(e, d)) & \text{altfel} \end{cases}$$

```
% (integer, arbore, arbore) – (i,i,o) determinist
% insereaza un element într-un ABC
inserare(E, nil, arb(E, nil, nil)).
inserare(E, arb(R, S, D), arb(R, SNou, D)) :-
    E <= R,
    !,
```



```

inserare(E, S, SNou).
inserare(E, arb(R, S, D), arb(R, S, DNou)) :-
    inserare(E, D, DNou).

```

```

% (arbore) – (i) determinist
% afișează nodurile arborelui în inordine
inordine(nil).
inordine(arb(R,S,D)) :-
    inordine(S),
    write(R),
    nl,
    inordine(D).

```

$$creeazaArb(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inserare(l_1, creeazaArb(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

```

% (arbore, list) – (i,o) determinist
% creează un ABC cu elementele unei liste
creeazaArb([], nil).
creeazaArb([H|T], Arb) :-
    creeazaArb(T, Arb1),
    inserare(H, Arb1, Arb).

```

```

% (list) – (i) determinist
% afișează elementele listei în ordine crescătoare (folosind sortare arborescentă)
sortare(L) :-
    creeazaArb(L, Arb),
    inordine(Arb).

```

### 3. Optimizarea prin recursivitate de coadă (tail recursion)

Recursivitatea are o mare problemă: consumă multă memorie. Dacă o procedură se repetă de 100 ori, 100 de stadii diferite ale execuției procedurii (cadre de stivă) sunt memorate.

Totuși, există un caz special când o procedură se apelează pe ea fără să genereze cadru de stivă. Dacă procedura apelatoare apelează o procedură ca ultim pas al sau (după acest apel urmează punctul). Când procedura apelată se termină, procedura apelatoare nu mai are altceva de făcut. Aceasta înseamnă că procedura apelatoare nu are sens să-și memoreze stadiul execuției, deoarece nu mai are nevoie de acesta.

#### Funcționarea recursivității de coadă

Iată două reguli depre cum să faceți o recursivitate de coadă:

1. Apelul recursiv este ultimul subgoal din clauza respectivă.
2. Nu există puncte de backtracking mai sus în acea clauză (adică, subgoal-urile de mai sus sunt deterministe).

Iată un exemplu:

```
tip(N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip(Nou).
```

Această procedură folosește recursivitatea de coadă. Nu consumă memorie, și nu se oprește niciodată. Eventual, din cauza rotunjirilor, de la un moment va da rezultate incorecte, dar nu se va opri.

### Exemple greșite de recursivitate de coadă

Iată cateva reguli despre cum să NU faceți o recursivitate de coadă:

1. Dacă apelul recursiv nu este ultimul pas, procedura nu folosește recursivitatea de coadă.

Exemplu:

```
tip (N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip (Nou),  
    nl.
```

2. Un alt mod de a pierde recursivitatea de coadă este de a lăsa o alternativă neîncercată la momentul apelului recursiv.

Exemplu:

```
tip(N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    tip(Nou).  
tip(N) :-  
    N < 0,  
    write('N este negativ.').
```

Aici, prima clauză se apelează înainte ca a doua să fie încercată. După un anumit număr de pași intră în criză de memorie.

3. Alternativa neîncercată nu trebuie neaparat să fie o clauză separată a procedurii recursive. Poate să fie o alternativă a unei clauze apelate din interiorul procedurii recursive.

Exemplu:

```
tip (N) :-  
    write(N),  
    nl,  
    Nou is N + 1,  
    verif(Nou),  
    tip(Nou).  
verif(Z) :- Z >= 0.  
verif(Z) :- Z < 0.
```

Dacă N este pozitiv, prima clauză a predicatului **verif** a reușit, dar a doua nu a fost încercată. Deci, **tip** trebuie să-și pastreze o copie a cadrului de stivă.

### Utilizarea tăieturii pentru păstrarea recursivității de coadă

A doua și a treia situație de mai sus pot fi înlăturate dacă se utilizează tăietura, chiar dacă există alternative neîncercate.

Exemplu la situația a doua:

```
tip (N) :-  
    N >= 0,  
    !,  
    write(N),  
    nl,  
    Nou = N + 1,  
    tip(Nou).  
tip(N) :-  
    N < 0,  
    write("N este negativ.").
```

Exemplu la situația a treia:

```
tip(N) :-  
    write(N),  
    nl,  
    Nou = N + 1,  
    verif(Nou),  
    !,  
    tip(Nou).  
verif(Z) :- Z >= 0.  
verif(Z) :- Z < 0.
```

## CURS 7

### Programare funcțională. Introducere în limbajul LISP

#### Cuprins

Bibliografie .....	1
1. Programare funcțională.....	1
1.1 Evaluare întârziată ( <i>lazy evaluation</i> ) .....	2
2. Limbajul Lisp.....	3
3. Elemente de bază ale limbajului Lisp .....	4
4. Structuri dinamice de date .....	5
4.1 Exemple .....	5
5. Reguli sintactice .....	6
6. Reguli de evaluare.....	7
6.1 Exemple .....	8
7. Funcții Lisp .....	8
(CONS $e_1$ $e_2$ ): l sau pp .....	8
(CAR l sau pp): e .....	9
(CDR l sau pp): e .....	9
7.1 Exemple .....	9

#### Bibliografie

Capitolul 2, Czibula, G., Pop, H.F., *Elemente avansate de programare în Lisp și Prolog. Aplicații în Inteligența Artificială.*, Ed. Albastră, Cluj-Napoca, 2012

#### Programare logică

1. *“Learning Prolog provides you with a different perspective on programming that, once understood, can be applied to other languages as well.”*
2. *“You should learn Prolog as a part of your personal quest to become a better programmer.”*
3. Constraint programming
4. Inductive logic programming

#### 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 nerekursive de ciclare de tip FOR
- **CE , NU CUM.**
- Limbaje funcționale – LISP (1958), Hope, ML, Scheme, Miranda, Haskell, Erlang (1995)
  - **Haskell**
    - pur funcțional, concurent
    - aplicații industriale, aplicații web
  - **Erlang** – programare funcțională concurentă
    - aplicații industriale: **telecomunicații**
- LISP nu e PUR funcțional
- Programare funcțională în limbaje precum Python, Scala, F#

## 1.1 Evaluare întârziată (*lazy evaluation*)

**Evaluarea întârziată** (*lazy evaluation*) este una dintre caracteristicile limbajelor funcționale.

- se întârzie evaluarea unei expresii până când valoarea ei e necesară (*call by need*)
  - în limbajul Haskell se evaluează expresiile doar când se știe că e necesară valoarea acestora
- în contrast cu *lazy evaluation* este *eager evaluation* (*strict evaluation*)
  - se evaluează o expresie când se știe că e posibil să fie folosită valoarea acesteia
- **exemplu:**  $f(x, y) = 2 * x; k=f(d, e)$ 
  - *lazy evaluation* – se evaluează doar  $d$

- *eager evaluation* – evaluăm  $d$  și  $e$  când calculăm  $k$ , chiar dacă  $y$  nu e folosit în funcție
- majoritatea limbajelor de programare (C, Java, Python) folosesc evaluarea strictă ca și mecanism implicit de evaluare
- Lazy Python

## 2. Limbajul Lisp

- 1958 John McCarthy elaborează o primă formă a unui limbaj (**LISP** - **LIS**t **P**rocessing) 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;
    - funcția Lisp **eval** care servește deopotrivă ca definiție formală a limbajului și ca interpretor;
    - colectarea spațiului disponibil (*garbage collection*) ca mijloc de tratare a problemei dealocării.
  - Consideră *funcția* ca obiect fundamental – transmis ca parametru, returnat ca rezultat al unei prelucrări, parte a unei structuri de date.
  - Domeniul de utilizare al limbajului Lisp este cel al calculului simbolic, adică al prelucrărilor efectuate asupra unor șiruri de simboluri grupate în expresii. Una dintre cauzele forței limbajului Lisp este posibilitatea de manipulare a unor obiecte cu structură ierarhizată.
  - **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.
- “Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.” (Paul Graham, 2001)*
- 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.
  - **Common Lisp** e un *limbaj funcțional* (dar și imperativ, orientat-obiect, putând fi folosit în manieră declarativă).
- Mecanismul implicit de evaluare în **CommonLisp** – evaluarea strictă (*eager/strict evaluation*).
  - Extensie CLAZY – evaluare întârziată (*lazy evaluation*) în Common Lisp
- 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.
- În LISP nu există nedeterminism.

## 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

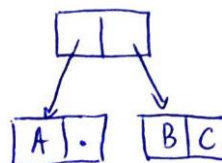
- Variațiuni ale conținutului acestor două câmpuri generează alte genuri de structuri: dacă C1 conține un pointer atunci se generează arbori binari, iar dacă C2 poate fi și altceva decât pointer atunci apar așa-numitele *perechi cu punct* (elemente cu două câmpuri-dată).
  - se pot astfel forma elemente ale căror câmpuri pot fi în egală măsură ocupate de informații atomice (numere sau simboluri) și de informații structurale (referințe spre alte elemente). Reprezentarea grafică a unei astfel de structuri este cea a unui arbore binar (structura arborescentă).
- **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 (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct <p> atunci lista (l<sub>1</sub> l<sub>2</sub>...l<sub>n</sub>) este echivalentă cu perechea cu punct (l<sub>1</sub> . <p>)

### 4.1 Exemple

- (A . B) nu are echivalent listă;



- ((A . NIL) . (B . C)) nu are echivalent listă.



- (A) ↔ (A . NIL)





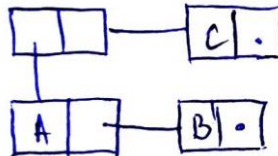
- $(A\ B) \leftrightarrow (A \cdot (B))$



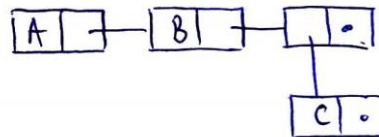
- $(A\ B\ C) \leftrightarrow (A \cdot (B \cdot (C \cdot NIL)))$



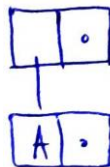
- $((A\ B)\ C) \leftrightarrow ((A \cdot (B \cdot NIL)) \cdot (C \cdot NIL))$



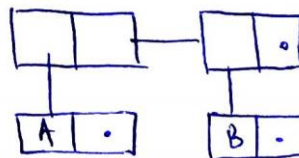
- $(A\ B\ (C)) \leftrightarrow (A \cdot (B \cdot ((C \cdot NIL) \cdot NIL)))$



- $((A)) \leftrightarrow ((A \cdot NIL) \cdot NIL)$



- $((A)\ (B)) \leftrightarrow ((A \cdot NIL) \cdot ((B \cdot NIL) \cdot NIL))$



**TEMĂ** Desenați structura fiecăreia din listele de mai jos:

1.  $((A \cdot NIL) \cdot ((B \cdot NIL) \cdot NIL)) = ((A)\ (B))$
2.  $((NIL)) = ((NIL \cdot NIL) \cdot NIL)$
3.  $(( )) = (NIL \cdot NIL)$
4.  $(A\ (B \cdot C)) = (A \cdot ((B \cdot C) \cdot NIL))$

## 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 !!! Lista vida este singura lista care este considerată atom
  - (e<sub>1</sub> e<sub>2</sub> ... e<sub>n</sub>), n ≥ 1  
unde e, e<sub>1</sub>, ..., e<sub>n</sub> sunt **S-expresii**
6. **pereche cu punct** (pp) – este o construcție de forma (e<sub>1</sub> . e<sub>2</sub>) unde e<sub>1</sub> și e<sub>2</sub> 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/operator, 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.*

## 6.1 Example

> 'A A	> (quote A) A
> (A) the function A is undefined	> (NIL) the function NIL is undefined
> ' (A) (A)	> ' (NIL) (NIL)
> () NIL	> () the function NIL is undefined
> NIL NIL	> ' () (())
> (A.B) the function A\B is undefined	> ' (A.B) (A\B)

## 7. Funcții Lisp

Prelucrările de liste se pot face la:

- nivel superficial
- la orice nivel

**Exemplu.** Să se calculeze suma atomilor numerici dintr-o listă neliniară

- la nivel superficial (suma ' (1 (2 a (3 4) b 5) c 1))  $\rightarrow 2$
- la toate nivelurile (suma ' (1 (2 a (3 4) b 5) c 1))  $\rightarrow 16$

**(CONS e<sub>1</sub> e<sub>2</sub>): l sau pp**

- funcția **constructor**
- se evaluează argumentele și apoi se trece la evaluarea funcției
- formează o pereche 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.

Iată câteva exemple:

- (CONS 'A 'B) = (A . B)
- (CONS 'A '(B)) = (A B)
- (CONS '(A B) '(C)) = ((A B) C)
- (CONS '(A B) '(C D)) = ((A B) C D)
- (CONS 'A '(B C)) = (A B C)
- (CONS 'A (CONS 'B '(C))) = (A B C)

**(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.

Iată câteva exemple:

- $(CAR '(A B C)) = A$
- $(CAR '(A . B)) = A$
- $(CAR '((A B) C D)) = (A B)$
- $(CAR (CONS '(B C) '(D E))) = (B C)$
- $(CDR '(A B C)) = (B C)$
- $(CDR '(A . B)) = B$
- $(CDR '((A B) C D)) = (C D)$
- $(CDR (CONS '(B C) '(D E))) = (D E)$

CONS reface o listă pe care CAR și CDR au secționat-o. De observat, însă, că obiectul obținut ca urmare a aplicării funcțiilor CAR, CDR și CONS, nu este același cu obiectul de la care s-a plecat:

- $(CONS (CAR '(A B C)) (CDR '(A B C))) = (A B C)$
- $(CAR (CONS 'A '(B C))) = A$
- $(CDR (CONS 'A '(B C))) = (B C)$

La folosirea repetată a funcțiilor de selectare se poate utiliza prescurtarea  $Cx_1x_2...x_nR$ , echivalentă cu  $Cx_1R$  o  $Cx_2R$  o ... o  $Cx_nR$ , unde caracterele  $x_i$  sunt fie 'A', fie 'D'. În funcție de implementări, se vor putea utiliza în această compunere cel mult trei sau patru  $Cx_iR$ :

- $(CAADDR '((A B) C (D E))) = D$
- $(CDAAAR '((((A) B) C) (D E))) = NIL$
- $(CAR '(CAR (A B C))) = CAR$

## 7.1 Example

### Operare CLisp

- (ed) – editor
- se scrie funcția **fct** în fișierul **f.lisp**. De exemplu, fișierul **f.lisp** conține următoarea definiție  

```
(defun fct(l)
  (cdr l)
)
```

- se încarcă fișierul **f.lsp**  
`> (load 'f)`
- dacă nu sunt erori, se evaluează funcția **fct**  
`> (fct '(1 2)) → (2)`

**EXEMPLU 7.1.1** Să se genereze o listă cu numerele întregi din intervalul [a, b].

```
(defun interval (a b)
  ;; returnează o listă cu numerele întregi din intervalul [a, b]
  (if (> a b)
      nil
      (cons a (interval (+ a 1) b)))
)
```

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la nivelul superficial dintr-o listă neliniară .

Model recursiv

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ suma(l_2 \dots l_n) & \text{altfel} \end{cases}$$

; suma atomilor de la nivelul superficial al unei liste neliniare

; (suma '(1 (2 (3 4) 5) 1)) → 2

(defun suma(l)

; **forma COND** – forma condițională: permite ramificarea prelucrărilor

(cond

((null l) 0)

; **NUMBERP** – returnează T dacă argumentul e număr

((numberp (car l)) (+ (car l) (suma (cdr l)))))

(t (suma (cdr l))))

)

)

**EXEMPLU 7.1.2** Să se calculeze suma atomilor numerici de la orice nivel dintr-o listă neliniară.

## Model recursiv

$$suma(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ suma(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom} \\ suma(l_1) + suma(l_2 \dots l_n) & \text{altfel} \end{cases}$$

; să se calculeze suma atomilor numerici dintr-o listă neliniară  
; (la toate nivelurile) (suma '(1 (2 a (3 4) b 5) c 1)) → 16

```
(defun suma(l)
  (cond
    ((null l) 0)
    ((numberp (car l)) (+ (car l) (suma (cdr l))))
    ; ATOM – returnează T dacă argumentul e atom
    ((atom (car l)) (suma (cdr l)))
    ; ultima clauză e pentru primul element listă
    (t (+ (suma (car l)) (suma (cdr l)))))
)
```

## CURS 8

### Funcții LISP. Exemple

#### Cuprins

1. Funcții Lisp (cont.).....	1
1.1 Predicate de bază în Lisp .....	2
1.2 Operații logice.....	3
1.3 Operații aritmetice .....	4
1.4 Operatori relaționali .....	4
2. Ramificarea prelucrărilor. Funcția COND.....	4
3. Definirea funcțiilor utilizator. Funcția DEFUN.....	5
4. Exemple .....	7

### 1. Funcții Lisp (cont.)

#### (LIST e1 e2...): l

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

Câteva exemple:

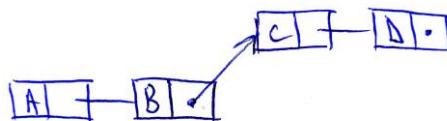
- (LIST 'A 'B) = (A B)
- (LIST '(A B) 'C) = ((A B) C)
- (LIST 'A) = (A) ; echivalent cu (CONS A NIL)
- (LIST '(A B C) NIL) = ((A B C) NIL)

#### (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.
- dacă toate S-expresiile argument sunt liste, atunci efectul este concatenarea listelor
- în CLisp argumentele nu pot fi atomi, cu excepția ultimului
- funcția APPEND este puternic consumatoare de memorie; prima listă argument este recopiată înainte de a fi legată de următoarea

Câteva exemple:

- (APPEND '(A B) '(C D)) = (A B C D)



- (APPEND '(A B) 'C) = (A B . C)



- (APPEND '(A B C) '()) = (A B C)
- (APPEND (cons 'A 'B) 'C) = (A . C)
- (APPEND '(A . B) 'F) = (A . F)

### (LENGTH l): n

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

## 1.1 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.

Câteva exemple:

- (ATOM 'A) = T;
- (ATOM A) = depinde la ce anume se evaluează A;
- (ATOM '(A B C)) = NIL;
- (ATOM NIL) = T;

### (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.

Câteva exemple:

- (LISTP 'A) = NIL;
- (LISTP A) = T dacă A se evaluează la o listă;
- (LISTP '(A B C)) = T;
- (LISTP NIL) = T;

### (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)

De exemplu,



e <sub>1</sub>	e <sub>2</sub>	EQUAL
'(A B)	'(A B)	T
'(A B)	'(A (B))	NIL
3.0	3.0	T
8	8	T
'a	'a	T

#### (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.

## 1.2 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.

De exemplu, forma (AND 1 (car 1)) este echivalentă cu următoarea structură alternativă

#### Dacă 1=Ø atunci

returnează Ø

#### altfel

returnează (car 1)

#### SfDacă

#### (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.

De exemplu, forma (OR (cdr 1) (car 1)) este echivalentă cu următoarea structură alternativă

#### Dacă (cdr 1) ≠ Ø atunci

returnează (cdr 1)

#### altfel

returnează (car l)  
**SfDacă**

!!! AND și OR *nu sunt funcții, sunt operatori speciali*

### 1.3 Operații aritmetice

**(+ n1 n2 ... ) : n**

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n1+n2+\dots$

**(- n1 n2 ... ) : n**

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n1-n2-\dots$

**(\* n1 n2 ... ) : n**

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n1*n2*\dots$

**(/ n1 n2 ... ) : n**

- se evaluează argumentele și apoi se trece la evaluarea funcției
- rezultat:  $n1/n2/\dots$

**(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 poate folosi MOD (rest).

### 1.4 Operatori relaționali

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

## 2. 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 **f<sub>1</sub>, f<sub>2</sub>, ..., f<sub>n</sub>** sunt liste nevide de lungime arbitrară (**f<sub>1</sub> f<sub>2</sub> ... f<sub>n</sub>**) 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<sub>2</sub>, f<sub>3</sub>, ... f<sub>n</sub>. Se întoarce valoarea ultimei forme evaluate din clauza selectată;
- dacă nu se selectează nici o clauză, COND întoarce NIL.

Următoarea secvență

```
(COND
  ((> X 5) (CONS 'A '(B)))
  (T 'A)
)
```

returnează (A B) dacă X este 7, respectiv A dacă X este 4.

Ce returnează secvența

```
(COND
  ((> X 5) (LIST 'A) (CONS 'A '(B)))
  (T 'A)
)
```

în cazul în care X este 10?

### 3. Definirea funcțiilor utilizator. Funcția DEFUN

**(DEFUN s l f<sub>1</sub> f<sub>2</sub>... ): s**

Funcția DEFUN creează 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<sub>i</sub>** sunt forme; evaluarea apelului decurge astfel:

- (a) se evaluează argumentele  $arg_1, arg_2, \dots, arg_n$ ; fie  $v_1, v_2, \dots, v_n$  valorile lor;
- (b) fiecare parametru formal din definiția funcției 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 parametrii 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  $p_1, p_2 \dots p_n$  se “dezleagă” de valorile  $v_1, v_2, \dots$  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).

Următorul exemplu întoarce argumentul dacă acesta este atom, NIL dacă acesta este listă vidă și primul element dacă argumentul e listă.

```
(DEFUN PRIM (X)
  (COND
    ((ATOM X) X)
    ((NULL X) NIL) ;inutil
    (T (CAR X))
  )
)
```

Următorul exemplu întoarce maximum valorilor celor două argumente.

```
(DEFUN MAX (X Y)
  (COND
    ((> X Y) X)
    (T Y)
  )
)
```

Următorul exemplu întoarce ultimul element al unei liste, la nivel superficial.

```
(DEFUN ULTIM (X)
  (COND
    ((ATOM X) X)
    ((NULL (CDR X)) (CAR X))
    (T (ULTIM (CDR X)))
  )
)
```

Următorul exemplu rescrie CAR pentru a întoarce NIL dacă argumentul este atom și nu produce mesaj de eroare.

```
(DEFUN XCAR (X)
  (COND
```

```

((ATOM X) NIL)
(T (CAR X))
)
)

```

### **Observații.**

- (1). 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).
- (2). În cazul în care o funcție este redefinită cu o altă aritate (număr de argumente), este valabilă ultimă definiție a acesteia. De **exemplu**, fie următoarele definiții (în ordinea indicată)

```

(defun f(l)
  (car l)
)
(defun f(l1 l2)
  (cdr l2)
)

```

Evaluarea formei (f '(1 2 3)) produce eroare, pe când evaluarea (f '(1 2) '(3 4)) produce (4).

Fie următoarea definiție de funcție

```

(DEFUN F (X Y)
  (COND
    ((< X Y) X)
    (T Y)
  )
)

```

Care este efectul evaluării (F 2 5)?

## **4. Exemple**

**EXEMPLU 4.1** Să se calculeze suma atomilor numerici de la orice nivel dintr-o listă neliniară.

Modele recursive

**Varianta 1** (discutată în [Cursul 7](#))

$$\text{suma}(l_1 l_2 \dots l_n) = \begin{cases} 0 & \text{daca lista e vida} \\ l_1 + \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom numeric} \\ \text{suma}(l_2 \dots l_n) & \text{daca } l_1 \text{ este atom} \\ \text{suma}(l_1) + \text{suma}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

## Varianta 2

$$suma(l) = \begin{cases} l & \text{dacă } l \text{ atom numeric} \\ 0 & \text{dacă } l \text{ atom} \\ suma(l_1) \oplus suma(l_2 \dots l_n) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

; să se calculeze suma atomilor numerici dintr-o listă neliniară  
 ; (la toate nivelurile) (suma '(1 (2 a (3 4) b 5) c 1)) → 16

**EXEMPLU 4.2** Să se construiască lista obținută prin adăugarea unui element la sfârșitul unei liste.

(adaug '3 '(1 2)) → (1 2 3)  
 (adaug '(3) '(1 2)) → (1 2 (3))  
 (adaug '3 '()) → (3)

### Model recursiv

; se returnează lista  $(l_1, l_2, \dots, l_n, e)$

$$adaug(e, l_1 l_2 \dots l_n) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus adaug(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun adaug(e l)
  (cond
    ((null l) (list e)) ; (list e) sau (cons e nil)
    (t (cons (car l) (adaug e (cdr l))))
  )
)
```

**EXEMPLU 4.3.** Metoda variabilei colectoare. Să se definească o funcție care inversează o listă liniară.

(invers '(1 2 3)) va produce (3 2 1).

Modelul recursiv este

$$invers(l_1 l_2 \dots l_n) = \begin{cases} \phi & \text{daca } l \text{ e vida} \\ invers(l_2 \dots l_n) \oplus l_1 & \text{altfel} \end{cases}$$

O definiție posibilă pentru funcția INVERS este următoarea:

```

(DEFUN INVERS (L)
  (COND
    ((ATOM L) L)
    (T (APPEND (INVERS (CDR L)) (LIST (CAR L)))))
  )
)

```

Complexitatea timp e data de recurența

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n-1) + n & \text{altfel} \end{cases}$$

Problema este că o astfel de definiție consumă multă memorie. Eficiența aplicării funcțiilor Lisp (exprimată prin consumul de memorie) se măsoară prin numărul de CONS-uri pe care le efectuează. Să ne reamintim că (LIST arg) este echivalent cu (CONS arg NIL) și să subliniem de asemenea că funcția APPEND acționează prin copierea primului argument, care este apoi “lipit” de cel de-al doilea argument. Astfel, funcția REVERSE definită mai sus va realiza copierea fiecărui (REVERSE (CDR L)) înainte de “lipirea” sa la al doilea argument. De exemplu pentru lista (A B C D E) se vor copia listele NIL, (E), (E D), (E D C), (E D C B), deci pentru o listă de dimensiune N vom avea  $1 + 2 + \dots + (N-1) = N(N-1) / 2$  folosiri de CONS-uri. Deci, complexitatea timp este  $\theta(n^2)$ ,  $n$  fiind numărul de elemente din listă.

O soluție pentru a reduce complexitatea timp a operației de inversare este **folosirea metodei variabilei colectoare**: scrierea unei funcții auxiliare care utilizează doi parametri (Col = lista destinație și L = lista sursă), scopul ei fiind trecerea pe rând a câte unui element din L spre Col:

L	Col
(1, 2, 3)	$\emptyset$
(2, 3)	(1)
(3)	(2, 1)
$\emptyset$	(3, 2, 1)

Modelele recursive

$$invers\_aux(l_1 l_2 \dots l_n, Col) = \begin{cases} Col & \text{daca } l \text{ e vida} \\ invers\_aux(l_2 \dots l_n, l_1 \oplus Col) & \text{altfel} \end{cases}$$

$$invers(l_1 l_2 \dots l_n) = invers\_aux(l_1 l_2 \dots l_n, \emptyset)$$

```

(DEFUN INVERS_AUX (L Col)
  (COND
    ((NULL L) Col)

```

```

        (T (INVERS_AUX (CDR L) (CONS (CAR L) Col)))
    )
)

```

Ceea ce dorim noi se realizează prin apelul (INVERS\_AUX L ()), dar să nu uităm ca am pornit de la necesitatea definirii unei funcții ce trebuie apelată cu (INVERS L). De aceea, funcția INVERS se va defini:

```

(DEFUN INVERS (L)
  (INVERS_AUX L ()))
)

```

Funcția INVERS\_AUX are rolul de funcție auxiliară, ea punând în evidență rolul argumentului **Col** ca **variabilă colectoare** (variabilă ce colectează rezultatele parțiale până la obținerea rezultatului final). Se vor efectua atâtea CONS-uri cât este lungimea listei sursă L, deci complexitatea timp este  $\theta(n)$ ,  $n$  fiind numărul de elemente din listă.

$$T(n) = \begin{cases} 1 & \text{daca } n = 0 \\ T(n-1) + 1 & \text{altfel} \end{cases}$$

Să observăm deci că colectarea rezultatelor parțiale într-o variabilă separată poate contribui la micșorarea complexității algoritmului. Dar acest lucru nu este valabil în toate cazurile: sunt situații în care folosirea unei variabile colectoare crește complexitatea prelucrării, în cazul în care elementele se adaugă la finalul colectoarei (nu la începutul ei, ca în exemplul indicat).

**EXEMPLU 4.4.** Să se definească o funcție care să determine lista perechilor dintre un element dat și elementele unei liste.

```
(LISTA 'A '(B C D)) = ((A B) (A C) (A D))
```

Modelul recursiv este

$$lista(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ (e, l_1) \oplus lista(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```

(DEFUN LISTA (E L)
  (COND
    ((NULL L) NIL)
    (T (CONS (LIST E (CAR L)) (LISTA E (CDR L)))))
  )
)

```

**EXEMPLU 4.5.** Să se definească o funcție care să determine lista perechilor cu elemente în ordine strict crescătoare care se pot forma cu elementele unei liste numerice (se va păstra ordinea elementelor din listă).

```
(perechi '(3 1 5 0 4)) = ((3 5) (3 4) (1 5) (1 4))
```



Vom folosi o funcție auxiliară care returnează lista perechilor cu elemente în ordine strict crescătoare, care se pot forma între un element și elementele unei liste.

(per '2 '(3 1 5 0 4)) = ((2 3) (2 5) (2 4))

$$per(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ (e, l_1) \oplus per(e, l_2 \dots l_n) & \text{dacă } e < l_1 \\ per(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun per (e l)
  (cond
    ((null l) nil)
    (T (cond
      ((< e (car l)) (cons (list e (car l))(per e (cdr l))))
      (T (per e (cdr l)))
    )
  )
)
```

$$perechi(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ per(l_1, l_2 \dots l_n) \oplus perechi(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun perechi (l)
  (cond
    ((null l) nil)
    (t (append (per (car l) (cdr l)) (perechi (cdr l))))
  )
)
```

**EXEMPLU 4.6** Se dă o listă neliniară. Se cere să se dubleze valorile numerice de la orice nivel al listei, păstrând structura ierarhică a acesteia.

(dublare '(1 b 2 (c (3 h 4)) (d 6)))) → (2 b 8 (c (6 h 8)) (d 12)))

### Varianta 1

$$dublare(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l = \emptyset \\ 2l_1 \oplus dublare(l_2 \dots l_n) & \text{dacă } l_1 \text{ numeric} \\ l_1 \oplus dublare(l_2 \dots l_n) & l_1 \text{ atom} \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```

(defun dublare(l)
  (cond
    ((null l) nil)
    ((numberp (car l)) (cons (* 2 (car l)) (dublare (cdr l))))
    ((atom (car l)) (cons (car l) (dublare (cdr l))))
    (t (cons (dublare (car l)) (dublare (cdr l)))))
  )
)

```

### Varianta 2

$$dublare(l) = \begin{cases} 2l & \text{dacă } l \text{ numar} \\ l & \text{dacă } l \text{ atom} \\ dublare(l_1) \oplus dublare(l_2 \dots l_n) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```

(defun dublare(l)
  (cond
    ((numberp l) (* 2 l))
    ((atom l) l)
    (t (cons (dublare (car l)) (dublare (cdr l)))))
  )
)

```

**EXEMPLU 4.7** Folosirea unei variabile colectoare poate crește complexitatea. Se dă o listă liniară. Care este efectul următoarei evaluări:

(lista '(1 a 2 b 3 c)) → ???

Scrieți modelele matematice pentru fiecare funcție.

### Varianta 1 – direct recursiv (fără variabilă colectoare)

```

(defun lista (l)
  (cond
    ((null l) nil)
    ((numberp (car l)) (cons (car l) (lista (cdr l)))))
    (t (lista (cdr l)))
  )
)

```

Care este complexitatea timp în caz defavorabil?

## **Varianta 2** – cu variabilă colectoare

```
(defun lista_aux (l col)
  (cond
    ((null l) col)
    ((numberp (car l)) (lista_aux (cdr l) (append col (list (car l)))))
    (t (lista_aux (cdr l) col))
  )
)

(defun lista (l)
  (lista_aux l nil)
)
```

Care este complexitatea timp în caz defavorabil?

**EXEMPLU 4.8** Se dă o listă liniară. Care este efectul funcției PARCURG?

```
(defun parcurg_aux(L k col)
  (cond
    ((null L) nil)
    ((= k 0) (list col L))
    (t (parcurg_aux (cdr L) (- k 1) (cons (car l) col)))
  )
)

(defun parcurg (L k)
  (parcurg_aux L k nil)
)

(parcurg '(1 2 3 4 5) 3) → ???
```

## CURS 9

### Funcțiile SET, SETQ, SETF. Arbori binari. Exemple

#### Cuprins

1. Arbori binari.....	1
2. Exemple .....	2
3. SET, SETQ, SETF .....	9

### 1. Arbori binari

Un arbore binar se poate memora sub forma unei liste, în următoarele două moduri:

#### V1 Un arbore având

- rădăcină,
- subarborii subarbore-stâng și subarbore-drept

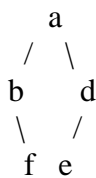
se va reprezenta sub forma unei liste neliniare de forma

(rădăcină lista-subarbore-stâng lista-subarbore-drept)

unde

- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V1) a subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V1) a subarborelui drept al nodului rădăcină

De exemplu arborele



se reprezintă, în varianta **V1** sub forma listei (a (b () (f)) (d (e)))

Reprezentarea V1 este cea mai potrivită pentru reprezentarea sub formă de listă a unui arbore cu rădăcină, fiind adecvată definiției recursive a unui arbore (binar).

#### V2 Un arbore având

- rădăcină,
- nr-arbori subarbori
- subarborii subarbore-stâng și subarbore-drept

se va reprezenta sub forma unei liste liniare de forma

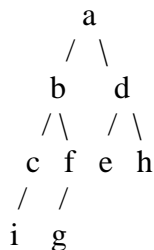
(rădăcină nr-subarbori lista-subarbore-stâng lista-subarbore-drept)

unde

- lista-subarbore-stâng reprezintă lista asociată (în memorarea sub forma V2) subarborelui stâng al nodului rădăcină
- lista-subarbore-drept reprezintă lista asociată (în memorarea sub forma V2) subarborelui drept al nodului rădăcină

Dezavantajul reprezentării **V2** este dat faptul că nu este potrivită pentru un arbore ordonat (de ex., în cazul arborelui binar nu se face distincție între subarboarele stâng și cel drept).

De exemplu arborele



se reprezintă, în varianta **V2** sub forma listei (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0)

**Observație.** Reprezentările V1 și V2 pot fi generalizate pentru arbori  $n$ -ari, varianta V1 fiind cea mai potrivită.

## 2. Exemple

**EXEMPLU 2.1** Se dă un arbore binar reprezentat în V1 (a se vedea Secțiunea 1). Să se determine lista liniară a nodurilor obținute prin parcurgerea arborelui în inordine.

(inordine '(a (b () (f)) (d (e)))) = (b f a e d)

În reprezentarea unui AB în varianta V1, sub forma unei liste  $l$  de forma (rădăcină lista-subarbore-stâng lista-subarbore-drept), observăm următoarele

- (car  $l$ ) – primul element al listei este rădăcina arborelui
- (cadr  $l$ ) – al doilea element al listei, la nivel superficial, este subarborele stâng
- (caddr  $l$ ) – al treilea element al listei, la nivel superficial, este subarborele drept

Model recursiv

$$inordine(l_1 l_2 l_3) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inordine(l_2) \oplus l_1 \oplus inordine(l_3) & \text{altfel} \end{cases}$$

```

(defun inordine(l)
  (cond
    ((null l) nil)
    (t (append (inordine (cadr l)) (list (car l)) (inordine (caddr l))))
    ; (t (append (inordine (cadr l)) (cons (car l) (inordine (caddr l)))))
  )
)

```

**EXEMPLU 2.2** Se dă un arbore binar reprezentat în V2 (a se vedea Secțiunea 2). Să se determine lista nodurilor obținute prin parcurgerea arborelui în inordine.

În reprezentarea unui AB în V2, față de reprezentarea V1, nu este clar identificat subarborele stâng și subarborele drept, pentru ca AB să poată fi prelucrat recursiv.

**Idee:** Dacă am reuși să extragem lista corespunzătoare (reprezentării în V2) subarborilor stâng și drept, problema s-ar reduce la cea prezentată în EXEMPLU 3.3.

Vom folosi o funcție auxiliară pentru a determina subarborele stâng al unui AB reprezentat în V2 (presupunem reprezentarea corectă).

(stang ' (a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0) ) = (b 2 c 1 i 0 f 1 g 0)

Vom folosi o funcție auxiliară, **parcure\_st**, care va parcurge lista începând cu al 3-lea element și care va returna subarborele stâng. **Idee:** se colectează elementele, începând cu al 3-lea element al listei, ....**până când?**

(parcure\_st ' (b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0) ) = (b 2 c 1 i 0 f 1 g 0)

$stang(l_1 l_2 \dots l_n) = parcure\_st(l_3 \dots l_n, 0, 0)$

nv – număr vârfuri

nm – număr muchii

$parcure\_st(l_1 l_2 \dots l_k, nv, nm)$

$$= \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ \emptyset & \text{daca } nv = 1 + nm \\ l_1 \oplus l_2 \oplus parcure\_st(l_3 \dots l_k, nv + 1, nm + l_2) & \text{altfel} \end{cases}$$

```

(defun parcure_st(arb nv nm)
  (cond
    ((null arb) nil)
    ((= nv (+ 1 nm)) nil)
    (t (cons (car arb) (cons (cadr arb) (parcure_st (caddr arb) (+ 1 nv) (+ (cadr arb) nm))))))
  )
)

```

(defun stang(arb)

(parcure\_st (cddr arb) 0 0)  
)

Temă. Scrieți funcția pentru determinarea subarborelui drept.

- 1) Se poate folosi aceeași idee ca la parcurgerea pentru subarborele stâng, doar că în momentul în care  $nv=1+nm$ , se va returna lista rămasă
- 2) Pentru a reduce complexitatea timp, se poate defini o sigură funcție care determină atât subarborele stâng, cât și cel drept.

(drept '(a 2 b 2 c 1 i 0 f 1 g 0 d 2 e 0 h 0) ) = (d 2 e 0 h 0)

Observație. Se poate folosi o funcție care să returneze atât subarborele stâng, cât și subarborele drept.

Având funcțiile anterioare care determină subarborele stâng și cel drept, lista nodurilor obținute prin parcurgerea arborelui în inordine se va face similar modelului recursiv descris în **EXEMPLU 2.1.**

$$\begin{aligned} inordine(l_1 l_2 \dots l_n) \\ = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inordine(stang(l_1 l_2 \dots l_n)) \oplus l_1 \oplus inordine(drept(l_1 l_2 \dots l_n)) & \text{altfel} \end{cases} \end{aligned}$$

**EXEMPLU 2.3** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) submulțimilor listei.

(subm '(1 2)) → (( ) (2) (1) (1 2))

? Cum obținem lista submulțimilor (1 2) dacă știm să generăm lista submulțimilor listei (2)?  
(( ) (2))

**Idee:** Dacă lista e vidă, submulțimea sa e lista vidă. Pentru determinarea submulțimilor unei liste [E|L], care are capul E și coada L, vom proceda în felul următor:

- i. determină o submulțime a listei L
- ii. plasează elementul E pe prima poziție într-o submulțime a listei L

Vom folosi o funcție auxiliară **insPrimaPoz**(*e*, *l*) care are ca parametri un element *e* și o listă *l* de liste liniare și returnează o copie a listei *l* în ale cărei liste se inserează *e* pe prima poziție.

(insPrimaPoz '3 '(( ) (2) (1) (1 2)) → ((3) (3 2) (3 1) (3 1 2))

; *l* este o listă de liste liniare

; se inserează  $e$  pe prima poziție, în fiecare listă din  $l$ , și se returnează rezultatul

$$\text{insPrimaPoz}(e, l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca lista } e \text{ vida} \\ (e, l_1) \oplus \text{insPrimaPoz}(e, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun insPrimaPoz(e l)
  (cond
    ((null l) nil)
    (t (cons (cons e (car l)) (insPrimaPoz e (cdr l))))
  )
)
```

;  $l$  este o mulțime reprezentată sub forma unei liste liniare

$$\text{subm}(l_1 l_2 \dots l_n) = \begin{cases} (\emptyset) & \text{daca lista } e \text{ vida} \\ \text{subm}(l_2 \dots l_n) \oplus \text{insPrimaPoz}(l_1, \text{subm}(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

```
(defun subm(l)
  (cond
    ((null l) (list nil))
    (t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l))))
  )
)
```

În codul Lisp scris anterior observăm faptul că apelul recursiv `(subm (cdr l))` din cea de-a doua clauză COND se repetă, ceea ce evident, nu este eficient din perspectiva complexității timp. O soluție pentru evitarea acestui apel repetat va fi folosirea unei funcții anonime LAMBDA (se va discuta în **Cursul 10**).

De asemenea, în **Cursul 11** vom discuta despre simplificarea implementării funcției `subm`, renunțând la utilizarea funcției auxiliare, folosind o funcție MAP.

**EXEMPLU 2.4** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se determine lista (mulțimea) permutărilor listei inițiale.

$$(\text{PERMUTĂRI } '(1\ 2\ 3)) \rightarrow ((1\ 2\ 3) (1\ 3\ 2) (2\ 1\ 3) (2\ 3\ 1) (3\ 1\ 2) (3\ 2\ 1))$$

? Cum obținem lista permutărilor `(1 2 3)` dacă știm să generăm lista permutărilor listei `(2 3)`?  
`((2 3) (3 2))`

**Idee:** Dacă lista  $e$  vidă, lista permutărilor sale este lista vidă. Pentru determinarea permutărilor unei liste  $[E|L]$ , care are capul  $E$  și coada  $L$ , vom proceda în felul următor:

1. determină o permutare  $L1$  a listei  $L$ ;



2. plasează elementul E pe toate pozițiile listei L1 și produce în acest fel lista X care va fi o permutare a listei inițiale [E|L].

Vom folosi câteva funcții auxiliare:

- 1) o funcție care returnează lista obținută prin inserarea unui element E pe o anumită poziție N într-o listă L (  $1 \leq N \leq \text{lungimea listei L}+1$ ).

(INS '1 2 '(2 3)) = (2 1 3)

(INS '1 3 '(2 3)) = (2 3 1)

Modelul recursiv

$$\text{ins}(e, n, l_1 l_2 \dots l_k) = \begin{cases} (e l_1 l_2 \dots l_k) & \text{daca } n = 1 \\ l_1 \oplus \text{ins}(e, n-1, l_2 \dots l_k) & \text{altfel} \end{cases}$$

```
(DEFUN INS (E N L)
  (COND
    ((= N 1) (CONS E L))
    (T (CONS (CAR L) (INS E (- N 1) (CDR L)))))
  )
)
```

- 2) o funcție care să returneze mulțimea formată din listele obținute prin inserarea unui element E pe pozițiile 1, 2,..., lungimea listei L+1 într-o listă L.

(INSERARE '1 '(2 3)) = ((2 3 1) (2 1 3) (1 2 3))

**Observație:** Se va folosi o funcție auxiliară (INSERT E N L) care returnează mulțimea formată cu listele obținute prin inserarea unui element pe pozițiile N, N-1, N-2,...,1 în lista L.

(INSERT '1 2 '(2 3)) = ((2 1 3) (1 2 3))

$$\text{insert}(e, n, l_1 l_2 \dots l_k) = \begin{cases} \emptyset & \text{daca } n = 0 \\ \text{ins}(e, n, l_1 l_2 \dots l_k) \oplus \text{insert}(e, n-1, l_1 l_2 \dots l_k) & \text{altfel} \end{cases}$$

```
(DEFUN INSERT (E N L)
  (COND
    ((= N 0) NIL)
    (T (CONS (INS E N L) (INSERT E (- N 1) L))))
  )
)
```

$$inserare(e, l_1 l_2 \dots l_n) = insert(e, n + 1, l_1 l_2 \dots l_n)$$

```
(DEFUN INSERARE (E L)
  (INSERT E (+ (LENGTH L) 1) L)
)
```

Temă Continuați implementarea pentru funcția PERMUTĂRI.

**EXEMPLU 2.5.** Se o listă liniară numerică. Să se determine lista sortată, folosind sortarea arborescentă (un ABC intermediar).

(sortare '(5 1 4 6 3 2)) = (1 2 3 4 5 6)

Pentru sortarea arborescentă a unei liste, vom proceda astfel:

1. Construim un ABC intermediar cu elementele listei inițiale
  - pentru memorarea ABC vom face folosi o listă liniară – varianta V1 (Secțiunea 1)
  - pentru construirea ABC intermediar vom avea nevoie de inserarea unui element în arbore
2. Parcurgerea în inordine a arborelui construit anterior ne va furniza lista inițială ordonată.

Model recursiv – inserarea unui element în ABC

$$inserare(e, l_1 l_2 l_3) = \begin{cases} (e) & \text{daca } l \text{ e vida} \\ l_1 \oplus inserare(e, l_2) \oplus l_3 & \text{daca } e \leq l_1 \\ l_1 \oplus l_2 \oplus inserare(e, l_3) & \text{altfel} \end{cases}$$

```
(defun inserare(e arb)
  (cond
    ((null arb) (list e))
    ((<= e (car arb)) (list (car arb) (inserare e (cadr arb)) (caddr arb)))
    (t (list (car arb) (cadr arb) (inserare e (caddr arb)))))
)
```

Model recursiv – construirea ABC din listă

$$construire(l_1 l_2 \dots l_n) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ inserare(l_1 construire(l_2 \dots l_n)) & \text{altfel} \end{cases}$$

```

(defun construire(l)
  (cond
    ((null l) nil)
    (t (inserare (car l) (construire (cdr l)))))
  )
)

; lista nodurilor unui ABC parcurs în inordine
(defun inordine (arb)
  (cond
    ((null arb) nil)
    (t (append (inordine (cadr arb)) (list (car arb)) (inordine (caddr arb)))))
  )
)

; sortarea arborescentă a listei
(defun sortare(l)
  (inordine (construire l))
)

```

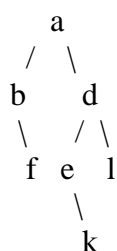
**Care este complexitatea timp a sortării arborescente?**

Temă Se dă un arbore binar, ale cărui noduri sunt distincte, reprezentat sub forma unei liste neliniare de forma (rădăcină lista-subarbore-stâng lista-subarbore-drept). Să se determine o listă liniară reprezentând calea de la rădăcină către un nod  $e$  dat.

De exemplu :

- (setq arb '(a (b () (f)) (d (e () (k)) (l))))

arborele



(cale 'm arb) → NIL  
 (cale 'f arb) → (a b f)  
 (cale 'k arb) → (a d e k)

$$\text{cale}(e, l_1 l_2 l_3) = \begin{cases} \emptyset & \text{daca } l \text{ e vida} \\ (e) & \text{daca } e = l_1 \\ l_1 \oplus \text{cale}(e, l_2) & \text{daca } e \in l_2 \\ l_1 \oplus \text{cale}(e, l_3) & \text{daca } e \in l_3 \\ \emptyset & \text{altfel} \end{cases}$$

### 3. 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 ( $f_n$ )

Iată câteva exemple

- |   |                        |
|---|------------------------|
| • (SET 'X 'A) = A                         | X se evaluează la A    |
| • (SET X 'B) = B                          | A se evaluează la B    |
| • (SET 'X (CONS (SET 'Y X) '(B))) = (A B) | X := (A B) și Y := A   |
| • (SET 'X 'A 'L (CDR L))                  | X := A și L := (CDR L) |

#### (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 ( $f_n$ )

Iată câteva exemple

- |                                 |                           |
|---------------------------------|---------------------------|
| • (SETQ X 'A) = A               |                           |
| • (SETQ A '(B C)) = (B C)       |                           |
| • (CDR A) = (C)                 |                           |
| • (SETQ X (CONS X A)) = (A B C) | X se evaluează la (A B C) |

Lista vidă este singurul caz de listă care are un nume simbolic, NIL, iar NIL este singurul atom care se tratează ca și listă, () și NIL reprezentând același obiect (elementul NIL are în câmpul CDR un pointer spre el însuși, iar în CAR are NIL).

- (CDR NIL) = NIL
- (CAR NIL) = NIL

**Observație.** Atenție la diferența dintre () = NIL, lista vidă, și (()) = (NIL), listă ce are ca singur element pe NIL.

- (CONS NIL NIL) = (())

### (SETF f<sub>1</sub> f'<sub>1</sub> ... f<sub>n</sub> f'<sub>n</sub>): e

- este un macro, are efect destructiv
- f<sub>1</sub>, ..., f<sub>n</sub> sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp
- f'<sub>1</sub>, ..., f'<sub>n</sub> sunt forme ale căror valori vor fi legate de locațiile desemnate de parametrii f<sub>1</sub>, ..., f<sub>n</sub> corespunzători.
- efect:
  - se evaluează formele de rang par (f'<sub>i</sub>) și valorile acestora se leagă de locațiile desemnate de formele f<sub>i</sub> corespunzătoare
- rezultatul întors valoarea ultimei forme evaluate (f'<sub>n</sub>)

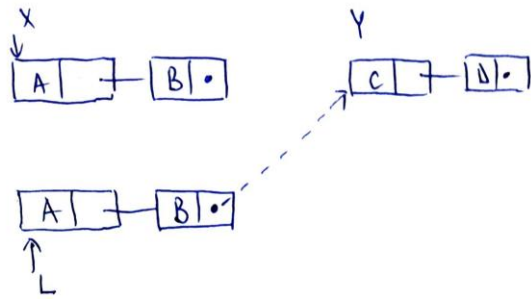
De remarcat că Lisp oferă pe lângă funcțiile SET și SETQ macro-ul SETF. Parametrii p<sub>1</sub>, ..., p<sub>n</sub> sunt forme care în momentul evaluării macro-ului accesează un obiect Lisp, iar e<sub>1</sub>, ..., e<sub>n</sub> sunt forme ale căror valori vor fi legate de locațiile desemnate de parametrii p<sub>1</sub>, ..., p<sub>n</sub> corespunzători. Rezultatul întors de SETF este valoarea ultimei expresii evaluate, e<sub>n</sub>.

Pentru a ne lămurii în legătură cu modul de operare a lui SETF, să vedem următorul exemplu:

- (SETQ A '(B C D))
    - în acest moment A se evaluează la (B C D)
  - (SETF (CADR A) 'X)
    - efectul este înlocuirea lui (CADR A) cu X
  - în acest moment A se evaluează la (B X D)
    - (SET (CADR A) 'Y)
  - efectul este evaluarea lui (CADR A) la X și inițializarea simbolului X la valoarea Y
  - în acest moment A se evaluează la (B X D)
  - în acest moment X se evaluează la Y
- 
- (SETQ X '(A B))                      X se evaluează la (A B)
  - (SETQ Y '(A B))                      Y se evaluează la (A B)
  - (SETF (CAR X) 'C)                    X se evaluează la (C B), Y se va evalua tot la (A B)

!!! Atenție la funcția APPEND

- (SETQ X '(A B))                      X se evaluează la (A B)
- (SETQ Y '(C D))                      Y se evaluează la (C D)
- (SETQ L (APPEND X Y))              L se evaluează la (A B C D)



- (SETF (CAR X) 'E)                      X se evaluează la (E B)
- L    ??
- (SETF (CADR Y) 'F)                    Y se evaluează la (C F)
- L    ??

## CURS 10

# Expresii LAMBDA. Mecanisme definiționale evaluate. Funcții MAP

### Cuprins

1. Definirea funcțiilor anonime. Expresii LAMBDA .....	1
1.1 Forma LABELS .....	2
1.2 Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetate .....	3
2. Mecanisme definiționale evaluate .....	5
Forme funcționale. Funcțiile APPLY și FUNCALL .....	8
3. Funcții MAP.....	11

## 1. 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<sub>1</sub> f<sub>2</sub> ... f<sub>m</sub>)**

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<sub>1</sub>...f<sub>m</sub> 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 f<sub>1</sub> f<sub>2</sub> ... f<sub>m</sub>) par<sub>1</sub> par<sub>2</sub> ... par<sub>n</sub>)**

### Exemple

Iată câteva exemple de utilizare a funcțiilor anonime

- **((lambda (l) (cons (car l) (cdr l))) '(1 2 3)) = (1 2 3)**
- **((lambda (l1 l2) (append l1 l2)) '(1 2) '(3 4)) = (1 2 3 4)**

- Să se definească o funcție care primește ca parametru o listă neliniară și returnează NIL dacă lista are cel puțin un atom numeric la nivel superficial și T, în caz contrar.

```
(defun f(l)
  (cond
    ((null l) t)
    (((lambda (v)
        (cond
          ((numberp v) t)
          (t nil)
        )
      )
     (car l)
     ) nil)
    (t (f (cdr l)))
  )
)
```

## 1.1 Forma LABELS

O formă specială pentru legarea locală a funcțiilor este forma LABELS.

### Example

**Ex1.** evaluarea

```
(labels ((fct(l)
          (cdr l)
        )
        )
  (fct '(1 2))
)
```

va produce (2).

**Ex2.**

```
(labels ((temp (n)
          (cond
            ((= n 0) 0)
            (t (+ 2 (temp (- n 1)))))
        )
        )
  (temp 3)
)
```

va produce 6



**Ex3.** Să se scrie o funcție care primește ca parametru o listă de liste formate din atomi și întoarce T dacă toate listele conțin atomi numerici și NIL în caz contrar.

```
(test '((1 2) (3 4))) = T
(test '((1 2) (a 4))) = NIL
(test '((1 (2)) (a 4))) = NIL
```

### Soluție

```
(DEFUN TEST (L)
  (COND
    ((NULL L) T)
    ((LABELS ((TEST1 (L)
                  (COND
                    ((NULL L) T)
                    ((NUMBERP (CAR L)) (TEST1 (CDR L)))
                    (T NIL)
                  )
                )
              )
      (TEST1 (CAR L))
    )
    (TEST (CDR L)))
  (T NIL)
)
```

## 1.2 Utilizarea expresiilor LAMBDA pentru evitarea apelurilor repetate

**Ex1.** Fie următoarea definiție de funcție

```
(defun g(l)
  (cond
    ((null l) nil)
    (t (cons (car (f l)) (cadr (f l))))
  )
)
```

Soluția pentru a evita apelul (**f l**) este folosirea unei funcții anonime utilizată local, care să poată fi apelată cu parametrul actual (**f l**).

### Varianta 1

```
(defun g(l)
  (cond
    ((null l) nil)
    (t ((lambda (v)
```

```

                                (cons (car v) (cadr v))
                              )
                            (f l)
                          )
                        )
                      )
                    )

```

### **Varianta 2**

```

(defun g(l)
  ((lambda (v)
    (cond
      ((null l) nil)
      (t (cons (car v) (cadr v)))
    )
  )
  (f l)
)
)

```

**Ex2.** Să considerăm definiția funcției care generează lista submulțimilor unei mulțimi reprezentate sub formă de listă (a se vedea **Cursul 9, Exemplul 2.3**).

```

(defun subm(l)
  (cond
    ((null l) (list nil))
    (t (append (subm (cdr l)) (insPrimaPoz (car l) (subm (cdr l)))))
  )
)

```

După cum se observă, apelul **(subm (cdr l))** este repetat. Pentru a evita apelul repetat, se va folosi o expresie LAMBDA.

O posibilă soluție este următoarea:

```

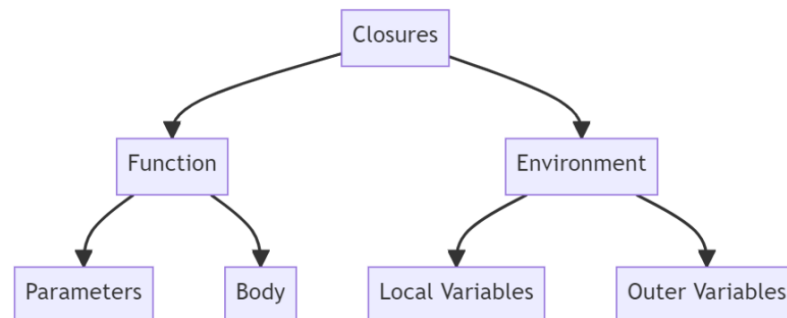
(defun subm(l)
  (cond
    ((null l) (list nil))
    (t ((lambda (s)
      (append s (insPrimaPoz (car l) s))
    )
      (subm (cdr l))
    )
  )
)
)

```

## 2. Mecanisme definiționale evolute

### Closure (închidere)

- *Lexical closure*
- <https://dept-info.labri.fr/~strandh/Teaching/MTP/Common/David-Lamkins/chapter15.html>
- combinație între o funcție și mediul lexical în care aceasta e definită
  - o închidere dă acces la domeniul de vizibilitate al unei funcții exterioare dintr-o funcție interioară



- o **închidere** este o funcție care acces la variabilele care sunt în afara domeniului lor de vizibilitate lexical, chiar după ce aceasta și-a încheiat execuția
    - astfel funcția își “memorează” mediul în care a fost creată
- expresiile Lambda în Lisp reprezintă **închideri**
- conceptul provine din programarea funcțională, dar apare și în programarea imperativă
  - [C++ 11](#) - funcțiile lambda construiesc o închidere
  - Javascript, Python

### Clojure

- <https://clojure.org/>
- dialect al limbajului Lisp pe platforme Java
- multi-paradigmă, orientat agent, concurent, funcțional, logic

### Closure Common Lisp (CCL)

- implementare Common Lisp
  - fire de execuție, compilare rapidă, mecanism pentru apelul funcțiilor externe (mecanism de *callback* – funcții Lisp apelate din cod extern)

Fie următoarele definiții și evaluări

```
> (defun f()
  10
)
F
> (setq f '11)
11
> (f)
10
> f
11
> (function f)
#<CLOSURE F NIL (DECLARE (SYSTEM::IN-DEFUN F)) (BLOCK F 10)>
> (setq g 7)
7
> (function g)
undefined function G
> (quote g) //echivalent cu 'g
G
> (function car)
#<SYSTEM-FUNCTION CAR>
> (function (lambda (l) (cdr l)))
#<CLOSURE :LAMBDA (L) (CDR L)>
```

**De remarcat faptul că AND și OR nu sunt considerate funcții, ci operatori speciali.**

```
> (function not)
#<SYSTEM-FUNCTION NOT>
> (function and)
undefined function AND
> (function or)
undefined function OR
```

!!! Din punct de vedere semantic, standardul CommonLisp impune să se indice dacă e vorba de o funcție sau un simbol.

Argument		
Funcție <b>f</b>	<b>#f</b>	(function <b>f</b> )
Simbol <b>x</b>	<b>'x</b>	(quote <b>x</b> )

	Funcție <b>f</b>
Standard	<b>#f</b>
CLisp	<b>'f</b>
GCLisp, Emacs Lisp, alte dialecte	<b>'f</b>

	Expresie Lambda
Standard	<b>#'(lambda ....)</b>

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 semnalând 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!**

**Exemplu** Fie lista L care pe prima poziție are un operator binar (+, -, \*, /), iar pe următoarele 2 poziții are 2 operanzi (numerici). De exemplu, L = (\* 2 3 - 4 1 ...). Se cere să returneze rezultatul aplicării operatorului (1-ul element al listei) asupra celor doi operanzi care urmează în listă. În exemplu nostru, ar trebui să se returneze valoarea 6.

Soluția este simplă: (EVAL (LIST (CAR L) (CADR L) (CADDR L))) = 6

## Forme funcționale. Funcțiile APPLY și FUNCALL

Există situații în care forma funcției nu se cunoaște, expresia ei trebuind să fie determinată dinamic. Ar trebui să avem ceva de genul

**(EXPR\_FUNC p<sub>1</sub> ... p<sub>n</sub>)**

Deoarece EXPR\_FUNC trebuie să genereze în cele din urmă o funcție ea este o așa-numită **formă funcțională**. Forma EXPR\_FUNC este evaluată până ce se obține o funcție sau, în general, o expresie ce poate fi aplicată parametrilor.

Într-o astfel de situație, evaluarea parametrilor este amânată până în momentul reducerii formei funcționale EXPR\_FUNC la funcția propriu-zisă F. Parametrii vor fi evaluați doar dacă F își evaluează, în prealabil, parametrii. Deci evaluarea formei de mai sus parcurge etapele:

- (i) reducerea formei EXPR\_FUNC la F (eventual o expresie LAMBDA sau macrodefiniție) și substituția lui EXPR\_FUNC prin F în forma de evaluat;
- (ii) evaluarea formei (F p<sub>1</sub> ... p<sub>n</sub>).

Există însă situații în care și numărul parametrilor trebuie stabilit dinamic, deci funcția determinată dinamic trebuie să accepte un număr variabil de parametri. Este nevoie deci de o modalitate de a permite aplicarea unei funcții asupra unei mulțimi de parametri sintetizată eventual dinamic. Acest lucru este oferit de funcțiile APPLY și FUNCALL.

**(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<sub>1</sub> p<sub>2</sub> ... p<sub>n</sub>).

### EXAMPLE

- (APPLY #'CONS '(A B)) va produce (A . B)
- (APPLY (FUNCTION CONS) '(A B)) va produce (A . B)
- (APPLY #'MAX '(1 2 3)) va produce 3
- (APPLY #'+ '(1 2 3)) va produce 6

- (DEFUN F(L) (CDR L))
- (APPLY #'F '((1 2 3)) va produce (2 3)
- (APPLY #'(LAMBDA (L) (CAR L)) '((A B C))) va produce A
- (SETQ P 'CAR)
- (APPLY P '((A B C))) va produce A
- (APPLY #'P '((A B C))) va produce eroare **undefined function P**
- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (APPLY (EVAL P) '((1 2 3))) va produce 1

**(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.

### EXAMPLE

- (FUNCALL #'CONS 'A 'B) va produce (A . B)
- (FUNCALL (FUNCTION CONS) 'A 'B) va produce (A . B)
- (FUNCALL #'MAX '1 '2 '3)) va produce 3
- (FUNCALL #'+ '1 '2 '3)) va produce 6
- (DEFUN F(L) (CDR L))
- (FUNCALL #'F '1 '2 '3) va produce (2 3)
- (FUNCALL #'(LAMBDA (L) (CAR L)) '(A B C)) va produce A
- (SETQ P 'CAR)
- (FUNCALL P '(A B C)) va produce A
- (FUNCALL # 'P '(A B C)) va produce eroare **undefined function P**
- (SETQ Q 'CAR)
- (SETQ P 'Q)
- (FUNCALL (EVAL P) '(1 2 3)) va produce 1

!!! **Atenție la folosirea AND și OR, deoarece nu sunt funcții.**

- (APPLY #'AND '((T NIL))) va produce eroare **undefined function AND**
- (APPLY #'OR '((T NIL))) va produce eroare **undefined function OR**

- (FUNCALL #'AND '(T NIL)) va produce eroare **undefined function AND**
- (FUNCALL #'OR '(T NIL)) va produce eroare **undefined function OR**

Soluția este definirea unei funcții al cărei efect să fie aplicarea AND/OR pe elementele unei liste cu valori logice T, NIL.

$$SI(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } l_1 \text{ e fals} \\ SI(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(defun SI(l)
  (cond
    ((null l) t)
    ; (t (and (car l) (SI (cdr l))))
    ((not (car l)) nil)
    (t (SI (cdr l)))
  )
)
```

- (FUNCALL #'SI '(T NIL T)) va produce **NIL**
- (SI '(T T T)) va produce **T**

### Exemple

#### Ex 1

- (DEFUN F()
 #'(LAMBDA (x) (CAR x))
 )
- (FUNCALL (F) '(1 2 3)) → ???
- (APPLY (F) '((1 2 3))) → ???

#### Ex 2

- (FUNCALL (FUNCTION (LAMBDA (x) x)) 1) → ???

### Exemplificare Closure Common Lisp

#### **Ex1**

```
(defun increment (x)
  (lambda (y)
    (+ x y)
  )
)

(setq inc5 (increment 5))
; returnează o nouă funcție (închidere) care adaugă 5 la argumentul său
```



```
(print (funcall inc5 3)) ; va afișa 8
```

## Ex2

```
(defun two-funs (x)
  (list
    (function (lambda () x))
    (function (lambda (y) (setq x y)))
  )
)
(setq funs1 (two-funs 6))

(funcall (first funs1)) => 6
(funcall (second funs1) 43) => 43
(funcall (first funs1)) => 43
(setq funs2 (two-funs 5))
(funcall (first funs2)) => 5
(funcall (second funs2) 13) => 13
(funcall (first funs2)) => 13
(funcall (first funs1)) => 43
```

## 3. 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 l<sub>1</sub> ... l<sub>n</sub>) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
  - CAR-ului listelor => e<sub>1</sub>
  - CADR-ului listelor => e<sub>2</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

Iată câteva exemple de aplicare ale funcției MAPCAR:

- (MAPCAR #'CAR '((A B C) (X Y Z))) se evaluează la (A X)
- (MAPCAR #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la (NIL T T)
- (MAPCAR #'LIST '(A B C)) se evaluează la ((A) (B) (C))
- (MAPCAR #'LIST '(A B C) '(1 2)) se evaluează la ((A 1) (B 2))
- (MAPCAR #'+'(1 2 3) '(4 5 6)) se evaluează la (5 7 9)

Aplicarea funcției LIST este posibilă indiferent de numărul listelor argument deoarece LIST este o funcție cu număr variabil de argumente.

**Observație.** Dacă F este o funcție unară, care se aplică unei liste  $L=(l_1 l_2 \dots l_n)$ , atunci

(MAPCAR #'F L) va produce lista  $(F(l_1), F(l_2), \dots, F(l_n))$

## Exemple

1. Să se definească o funcție MODIF care să modifice o listă dată ca parametru astfel: atomii nenumeriți rămân nemodificați iar cei numerici își dublează valoarea; modificarea trebuie făcută la toate nivelurile.

(MODIF '(1 (b (4) c) (d (3 (5 f))))) va produce (2 (b (8) c) (d (6 (10 f)))))

### Model recursiv

$$\text{MODIF}(l) = \begin{cases} 2l & \text{dacă } l \text{ numar} \\ l & \text{dacă } l \text{ atom} \\ \bigcup_{i=1}^n \text{MODIF}(l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(DEFUN MODIF (L)
  (COND
    ((NUMBERP L) (* 2 L)) ; determină operația asupra atomilor numerici
    ((ATOM L) L) ; determină operația asupra atomilor nenumeriți
    (T (MAPCAR #'MODIF L)) ; reprezintă strategia de parcurgere
  )
)
```

2. Să se construiască o funcție LGM ce determină lungimea celei mai lungi subliste dintr-o listă dată L (dacă lista este formată numai din atomi atunci lungimea cerută este chiar cea a listei L).

(LGM '(1 (2 (3 4) (5 (6)) (7)))) va produce 4

Descrierea algoritmului se poate exprima astfel:

- a). valoarea LGM este maximul dintre lungimea listei L și maximul valorilor de aceeași natură calculate prin aplicarea lui LGM pentru fiecare element al listei L în parte;
- b). LGM(atom) = 0.

### Model recursiv

$$\text{LGM}(l) = \begin{cases} 0 & \text{dacă } l \text{ e atom} \\ \max(n, \max(\text{LGM}(l_1), \text{LGM}(l_2), \dots, \text{LGM}(l_n))) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```

(DEFUN LGM(L)
  (COND
    ((ATOM L) 0)
    (T (MAX (LENGTH L) (APPLY #'MAX (MAPCAR #'LGM L)) ))
  )
)

```

Aplicarea funcțiilor ATOM și LENGTH calculează de fapt lungimile, (MAPCAR #'LGM L) realizând de fapt parcurgerea integrală a listei. Apelul (MAPCAR #'LGM L) furnizează o listă de lungimi. Deoarece trebuie să obținem maximum acestora, va trebui să aplicăm funcția MAX pe elementele acestei liste. Pentru aceasta folosim APPLY.

### (MAPCAN f l<sub>1</sub> ... l<sub>n</sub>) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
  - CAR-ului listelor => e<sub>1</sub>
  - CADR-ului listelor => e<sub>2</sub>
  - CADDR-ului listelor => e<sub>3</sub>
  - ...până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

### (NCONC l<sub>1</sub> l<sub>2</sub> ... l<sub>n</sub>) : 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.
- 
- (SETQ L1 '(A B C) L2 '(D E)) se evaluează la (D E)
  - (APPEND L1 L2) se evaluează la (A B C D E)
  - L1 se evaluează la (A B C)
  - L2 se evaluează la (D E)
  - (SETQ L3 (NCONC L1 L2)) se evaluează la (A B C D E)
  - L1 se evaluează la (A B C D E)
- 
- (SETQ L1 '(A) L2 '(B) L3 '(C)) se evaluează la (C)
  - L1 se evaluează la (A)
  - L2 se evaluează la (B)
  - L3 se evaluează la (C)
  - (NCONC L1 L2 L3) se evaluează la (A B C)
  - L1 se evaluează la (A B C)

- L2 se evaluează la (B C)
- L3 se evaluează la (C)

### Exemple MAPCAN

- (MAPCAN #'CAR '((A B C) (X Y Z))) se evaluează la NIL, deoarece NCONC cere liste, și ca atare (NCONC 'A 'X) este NIL
- (MAPCAN #'LIST '(A B C) '(1 2)) se evaluează la (A 1 B 2)
- (MAPCAN #'LIST '(A B C)) se evaluează la ( A B C )
- (MAPCAN #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la NIL
- (MAPCAN #'+'(1 2 3) '(4 5 6)) se evaluează la NIL

### Observație (MAPCAN vs. MAPCAR)

Fie următoarele definiții

```
(defun F(L)
  (cdr L)
)
```

```
(setq L '((1 2 3) (4 5 6) (7 8))) → ((1 2 3) (4 5 6) (7 8))
```

```
(mapcar #'F L) → ((2 3) (5 6) (8 9))
```

```
(mapcan #'F L) → (2 3 5 6 8 9)
```

⇒ (apply #'append (mapcar #'F L)) ≡ (mapcan #'F L)

### **(MAPLIST f l<sub>1</sub> ... l<sub>n</sub>) : l**

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
  - listelor => e<sub>1</sub>
  - CDR-ului listelor => e<sub>2</sub>
  - CDDR-ului listelor => e<sub>3</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **LIST** într-o listă ce e returnată rezultat

### Exemple MAPLIST

- (MAPLIST #'APPEND '(A B C) '(1 2 3)) furnizează ((A B C 1 2 3) (B C 2 3) (C 3))
- (MAPLIST #'(LAMBDA (X) X) '(A B C)) furnizează ((A B C) (B C) (C))
- (SETF TEMP '(1 2 7 4 6 5)) urmat de  
 (MAPLIST #'(LAMBDA (XL YL) (< (CAR XL)(CAR YL)))  
 TEMP (CDR TEMP)  
 ) va furniza lista (T T NIL T NIL)

Comparativ cu exemplele date la MAPCAR, aici vom obține:

- (MAPLIST #'CAR '((A B C) (X Y Z))) se evaluează la ((A B C) (X Y Z))
- (MAPLIST #'LIST '(A B C) '(1 2)) se evaluează la ( ((A B C) (1 2)) ((B C) (2)) )
- (MAPLIST #'LIST '(A B C)) se evaluează la ( ((A B C)) ((B C)) ((C)) )
- (MAPLIST #'EQUAL '(A (B C) D) '(Q (B C) D X)) se evaluează la (NIL NIL NIL)
- (MAPLIST #'+' (1 2 3) '(4 5 6)) va produce mesajul de eroare: “argument to + should be a number: (1 2 3)”.

### (MAPCON f l<sub>1</sub> ... l<sub>n</sub>) : l

- se evaluează argumentele și apoi se trece la evaluarea funcției
- efect: funcția n-ară **f** este aplicată pe rând asupra:
  - listelor => e<sub>1</sub>
  - CDR-ului listelor => e<sub>2</sub>
  - CDDR-ului listelor => e<sub>3</sub>
  - ....până când una din liste ajunge vidă
- Rezultatele sunt grupate cu **NCONC** într-o listă ce e returnată rezultat

### Exemple MAPCON

- (MAPCON #'CAR '((A B C) (X Y Z))) furnizează (A B C X Y Z)
- (MAPCON #'LIST '(A B C) '(1 2)) furnizează ((A B C) (1 2) (B C) (2))
- (MAPCON #'LIST '(A B C)) furnizează ((A B C) (B C) (C))
- (MAPCON #'EQUAL '(A (B C) D) '(Q (B C) D X)) furnizează NIL
- (MAPCON #'+' (1 2 3) '(4 5 6)) furnizează mesaj de eroare: : “argument to + should be a number: (1 2 3)”
- (DEFUN G(L)  
 (MAPCON #'LIST L)  
 )  
  
 (G '(1 2 3)) = ((1 2 3) (2 3) (3))
- (MAPCON #'(LAMBDA (L) (MAPCON #'LIST L)) '(1 2 3)) furnizează ((1 2 3) (2 3) (3) (2 3) (3) (3))

## CURS 11

### Exemple funcții MAP (cont).

#### EXEMPLU 2.1 Fie următoarele definiții

```
(defun f (L e)
  (list e L)
)
```

```
(setq L '(1 2 3))
(setq e 4)
```

(mapcar #'f L e) se evaluează la NIL

(mapcar #'(lambda (L) (f L e)) L) se evaluează la ((4 1) (4 2) (4 3))

#### EXEMPLU 2.2 Fie următoarea definiție de funcție

```
(defun f (L)
  (list L)
)
```

(mapcar #'f '(1 2 3)) se evaluează la ((1) (2) (3))

(mapcan #'f '(1 2 3)) se evaluează la (1 2 3)

De remarcat echivalența între

(mapcan #'f L) și (apply #'append (mapcar #'f L))

#### EXEMPLU 2.3 Să se definească o funcție care să returneze lungimea unei liste neliniare (în număr de atomi la orice nivel)

$(LG '(1 (2 (a) c d) (3))) = 6$

$$LG(L) = \begin{cases} 1 & \text{daca } L \text{ e atom} \\ \sum_{i=1}^n LG(L_i) & \text{daca } L \text{ e lista } (L_1 \dots L_n) \end{cases}$$

```
(DEFUN LG (L)
  (COND
    ((ATOM L) 1)
    (T (APPLY #'+ (MAPCAR #'LG L)))
  )
)
```

**EXEMPLU 2.4** Să se definească o funcție care având ca parametru o listă neliniară să returneze numărul de subliste (inclusiv lista) având lungime număr par (la nivel superficial).

$(nr '(1 (2 (3 (4 5) 6)) (7 (8 9)))) = 4$

Vom folosi o funcție auxiliară care returnează T dacă lista argument are număr par de elemente la nivel superficial, NIL în caz contrar.

$$nr(L) = \begin{cases} 0 & \text{daca } L \text{ e atom} \\ 1 + \sum_{i=1}^n lg(L_i) & \text{daca } L \text{ e lista } (L_1 \dots L_n) \text{ si } n \text{ e par} \\ \sum_{i=1}^n lg(L_i) & \text{altfel} \end{cases}$$

```
(DEFUN PAR (L)
  (COND
    ((= 0 (MOD (LENGTH L) 2)) T)
    (T NIL)
  )
)

(DEFUN nr (L)
  (COND
    ((ATOM L) 0)
    ((PAR L) (+ 1 (APPLY #'(MAPCAR #'nr L))))
    (T (APPLY #'(MAPCAR #'nr L))))
  )
)
```

**EXEMPLU 2.5** Să se definească o funcție care având ca parametru o listă neliniară să returneze lista atomilor (de la orice nivel) din listă.

$(atomi '(1 (2 (3 (4 5) 6)) (7 (8 9)))) = (1 2 3 4 5 6 7 8 9)$

$$atomi(L) = \begin{cases} (L) & \text{daca } L \text{ e atom} \\ \bigcup_{i=1}^n atomi(L_i) & \text{daca } L \text{ e lista } (L_1 \dots L_n) \end{cases}$$

```
(DEFUN atomi (L)
  (COND
    ((ATOM L) (LIST L))
    (T (MAPCAN #'(atomi L))))
  )
)
```

**Observație:** Aceeași cerință ar putea fi rezolvată folosind funcția MAPCAR

```
(DEFUN atomi (L)
  (COND
    ((ATOM L) (LIST L))
    (T (APPLY #'APPEND (MAPCAR #'atomi L)))
  )
)
```

**EXEMPLU 2.6** Să se definească o funcție care determină numărul de apariții, de la orice nivel, ale unui element într-o listă neliniară.

(nrap 'a '(1 (a (3 (4 a) a)) (7 (a 9)))) = 4

$$nrap(e, l) = \begin{cases} 1 & \text{daca } l = e \\ 0 & \text{dacă } l \text{ e atom} \\ \sum_{i=1}^n nrap(e, l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$

```
(defun nrap(e L)
  (cond
    ((equal L e) 1)
    ((atom L) 0)
    (t (apply #' + (mapcar #'(lambda(L)
                                (nrap e L))
                          L)
      )
    )
  )
)
```

**EXEMPLU 2.7** Se dă o listă neliniară. Se cere să se returneze lista din care au fost șterși atomii numerici negativi. Se va folosi o funcție MAP.

Ex: (stergere '(a 2 (b -4 (c -6)) -1)) → (a 2 (b (c)))

$$sterg(l) = \begin{cases} \emptyset & \text{daca } l \text{ numeric negativ} \\ l & \text{dacă } l \text{ e atom} \\ \bigcup_{i=1}^n sterg(l_i) & \text{altfel, } l = (l_1 l_2 \dots l_n) \text{ e lista} \end{cases}$$



```

(defun sterg(L)
  (cond
    ((and (numberp L) (minusp L)) nil)
    ((atom L) (list L))
    (t (list (apply #'append
                    (mapcar #'sterg L)
                    ; (sterg '((a) (c -2))) → ((a) (c))
                    ; (sterg '(a) → (a)
                    ; (sterg '(c -2) → (c)
                    ; (nconc '(a) '(c)) → (a c)
                    ; (nconc '((a)) '((c))) → ((a) (c))
                )
            )
      )
    )
  )
)

(defun stergere(L)
  (car (sterg L))
)

```

**EXEMPLU 2.8** Se dă un arbore n-ar nevid, reprezentat sub forma unei liste neliniare de forma (rădăcina lista\_sub1.....lista\_sub\_n) (V1 de reprezentare a arborilor binari, Curs 9). Se cere să se determine numărul de noduri din arbore.

(nrNoduri '(a (b (c) (d (e))) (f (g)))) va produce 7

Model recursiv

$$nrNoduri(l_1 l_2 \dots l_n) = \begin{cases} 1 & \text{daca } n = 1 \\ 1 + \sum_{i=2}^n nrNoduri(l_i) & \text{altfel} \end{cases}$$

```

(defun nrNoduri(L)
  (cond
    ((null (cdr L)) 1)
    (t (+ 1 (apply #'+
                  (mapcar #'nrNoduri (cdr L))
                )
      )
    )
  )
)

```

**EXEMPLU 2.9** Se dă un arbore n-ar nevid, reprezentat sub forma unei liste neliniare de forma (rădăcina lista\_sub1.....lista\_sub\_n) (V1 de reprezentare a arborilor binari, Curs 9). Se cere să se determine adâncimea arborelui. Observație. Nivelul rădăcinii este 0.

(*adancime* '(a (b (c) (d (e))) (f (g)))) va produce 3  
Model recursiv

*adancime*( $l_1 l_2 \dots l_n$ )

$$= \begin{cases} 0 & \text{daca } n = 1 \\ 1 + \max(\text{adancime}(l_2), \text{adancime}(l_3), \dots, \text{adancime}(l_n)) & \text{altfel} \end{cases}$$

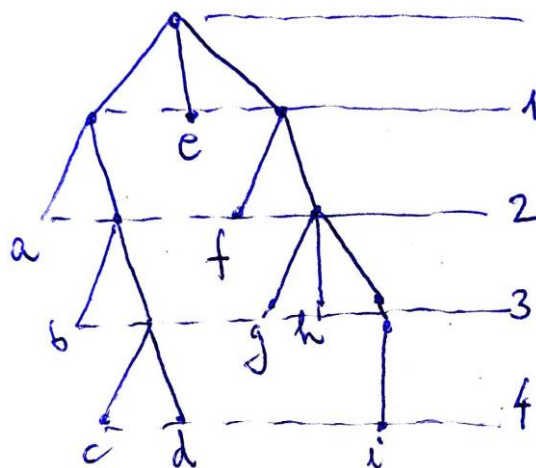
```
(defun adancime(L)
  (cond
    ((null (cdr L)) 0)
    (t (+ 1 (apply #'max
      (mapcar #'adancime (cdr L))
    )
    )
  )
)
```

**EXEMPLU 2.10** Să se determine lista atomilor de adâncime  $n$  dintr-o listă neliniară (nivelul superficial al listei se consideră 1).

(lista '((a (b (c d))) e (f (g h (i)))) 3) va produce (b g h)

(lista '((a (b (c d))) e (f (g h (i)))) 4) va produce (c d i)

(lista '((a (b (c d))) e (f (g h (i)))) 5) va produce NIL



## Model recursiv

$$lista(l, n) = \begin{cases} (l) & \text{dacă } n = 0 \text{ și } l \text{ atom} \\ \emptyset & \text{dacă } n = 0 \\ \emptyset & \text{dacă } l \text{ atom} \\ \bigcup_{i=1}^k lista(l_i, n-1) & \text{altfel, } l = (l_1 l_2 \dots l_k) \text{ e lista} \end{cases}$$

```
(defun lista(L n)
  (cond
    ((and (= n 0) (atom L)) (list L))
    ((= n 0) nil)
    ((atom L) nil)
    (t (mapcan #'(lambda(L)
                    (lista L (- n 1)))
                L)
        )
    )
  )
)
```

**EXEMPLU 2.11** Se dă o listă de liste. Se cere să se determine ....

```
(defun m (L)
  (cond
    ((numberp L) L)
    ((atom L) most-negative-fixnum)
    (t (apply #'max
               (mapcar #'m L)
              )
        )
    )
  )
)

(defun lista (L)
  (mapcan #'(lambda (L)
              ((lambda (v)
                 (cond
                   ((= 0 (mod v 2)) (list v))
                   (t nil)
                 )
                )
            )
          (m L)
        )
    L
  )
)
```

$(lista\ '(5\ a\ (2\ b\ (8)))\ (7\ a\ (9))\ (c\ d\ (10))) \rightarrow (8\ 10)$

**EXEMPLU 2.12** Se dă o listă liniară. Se cere....

```
(defun p (L)
  (mapcan #'(lambda (e1)
    (mapcar #'(lambda (e2)
      (list e1 e2)
    )
    L
  )
  L
)

(p '(1 2 3)) → ((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))
```

**EXEMPLU 2.13** O matrice se poate reprezenta sub forma unei liste formate din listele conținând elementele de pe linii, în ordine de la prima la ultima linie. De exemplu, lista ((1 2) (3 4)) corespunde matricei

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Fie L o listă asociată unei matrici cu elemente numerice. Care este rezultatul returnat de funcția *fct*?

```
(defun fct (L)
  (cond
    ((null (car L)) nil)
    (t (cons
      (mapcar #'car L)
      (fct (mapcar #'cdr L))
    )
  )
)

(fct '((1 2) (4 5) (7 8))) → ((1 4 7) (2 5 8))
```

**EXEMPLU 2.14** Să se definească o funcție care având ca parametru o listă neliniară returnează lista liniară a atomilor care apar pe orice nivel, dar în ordine inversă.

`(INVERSARE '(A (B C (D (E))) (F G))) = (G F E D C B A)`

```
(DEFUN INVERSARE (L)
  (COND
    ((ATOM L) (LIST L))
    (T (MAPCAN #'INVERSARE (INVERS L)))
  )
)
```

Unde INVERS este funcția care returnează o listă inversată la nivel superficial (Curs 8, Exemplu 4.3) – funcția REVERSE predefinită.

**EXEMPLU 2.15** O matrice se poate reprezenta în Lisp sub forma unei liste ale cărei elemente sunt liste reprezentând liniile matricei.

`( (linia1) (linia2)... )`

Să se definească o funcție care având ca parametri două matrice de ordin **n** returnează (sub formă de matrice) produsul acestora.

`(PRODUS '((1 2) (3 4)) '((2 -1) (3 1))) = ((8 1) (18 1))`

**Observație:** Vom folosi două funcții ajutoare: o funcție (COLOANE L) care returnează lista coloanelor matricei parametru L și o funcție (PR L1 L2) care returnează sub formă de matrice rezultatul înmulțirii matricei L1 (listă de linii) cu lista L2 (o listă de coloane ale unei matrice).

```
(DEFUN COLOANE (L)
  (COND
    ((NULL (CAR L)) NIL)
    (T (CONS (MAPCAR #'CAR L) (COLOANE (MAPCAR #'CDR L))))
  )
)
```

```

(DEFUN PR (L1 L2)
  (COND
    ((NULL (CAR L1)) NIL)
    (T (CONS (MAPCAR #'(LAMBDA (L)
                        (APPLY #'+ (MAPCAR #'* (CAR L1) L))
                      )
              L2)
          (PR (CDR L1) L2)
        )
    )
  )
)

(DEFUN PRODUS (L1 L2)
  (PR L1 (COLOANE L2))
)

```

**EXEMPLU 2.16** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se genereze lista submulțimilor mulțimii. Se va folosi o funcție MAP.

Ex: (*subm* '(1 2)) → (nil (1) (2) (1 2))

Observație

(setq e 1)  
 (mapcar #'(lambda(L) (cons e L)) '((2 3) (3 2))) va produce ((1 2 3) (1 3 2))

```

(defun subm (L)
  (cond
    ((null L) (list nil))
    (t ((lambda (s)
          (append s (mapcar #'(lambda (sb)
                              (cons (car L) sb)
                            )
                s)
        )
        (subm (cdr L))
      )
    )
  )
)

```

**EXEMPLU 2.17** Se dă o mulțime reprezentată sub forma unei liste liniare. Se cere să se genereze lista permutărilor mu lțimii. Se va folosi o funcție MAP.

Ex: (*permutari* '(1 2 3)) → ((1 2 3) (1 3 2) (2 1 3) (2 3 1) (3 1 2) (3 2 1))

```
(defun permutari (L)
  (cond
    ((null (cdr L)) (list L))
    (t (mapcan #'(lambda (e)
                    (mapcar #'(lambda (p)
                                (cons e p)
                                )
                                (permutari (remove e L)))
                    )
        L)
    )
  )
)
```

unde REMOVE este funcția care returnează lista din care s-a șters un element de la nivel superficial.

## CURS 12

### Generatori. Argumente opționale. OBLIST și ALIST. Macrodefiniții. Apostroful invers. Forme iterative

#### Cuprins

1. Generatori .....	1
2. Argumente opționale.....	3
3. OBLIST și ALIST.....	6
4. Macrodefiniții .....	7
5. Apostroful invers (backquote) .....	8
6. Forme iterative.....	9
7. Parametrii Lisp în context “Dynamic Scoping”.....	11
8. Universul expresiilor și universul instrucțiunilor .....	13
9. Independența ordinii de evaluare .....	15

#### 1. Generatori

Ca exemplu sugestiv pentru această problemă să considerăm exemplul unei funcții **VERIF** care primește o listă de liste și întoarce T dacă toate sublistele de la nivel superficial sunt liniare și NIL în caz contrar.

(VERIF '((1 2) (a (b)))) va produce NIL

(VERIF '((1 2) (a b))) va produce T

Vom folosi două funcții

- Funcția **LIN**, care verifică dacă o listă este liniară

Model recursiv

$$\text{LIN}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } l_1 \text{ nu e atom} \\ \text{LIN}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN LIN (L)
  (COND
    ((NULL L) T)
    (T (AND (ATOM (CAR L)) (LIN (CDR L) )))
  )
)
```



- Funcția **VERIF**, care verifică dacă o listă de liste are toate sublistele de la nivel superficial liste liniare.

Model recursiv

$$\text{VERIF}(l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg \text{LIN}(l_1) \\ \text{VERIF}(l_2 \dots l_n) & \text{altfel} \end{cases}$$

```
(DEFUN VERIF (L)
  (COND
    ((NULL L) T)
    (T (AND (LIN (CAR L)) (VERIF (CDR L) )))
  )
)
```

Să observăm că funcțiile LIN și VERIF de mai sus au aceeași structură, conformă șablonului

```
(DEFUN F (L)
  (COND
    ((NULL L) T)
    (T (AND (F1 (CAR L)) (F (CDR L) )))
  )
)
```

cu F1 = LIN în cazul F = VERIF și F1 = ATOM în cazul F = LIN. O astfel de structură este des folosită, “rețeta” ei de lucru fiind: elementele unei liste L sunt transmise pe rând (în ordinea apariției lor în listă) unei funcții F (în cazul de mai sus F1) care le va prelucra. Dacă F întoarce NIL acțiunea se încheie, rezultatul final fiind NIL. Altfel, parcurgerea continuă până la epuizarea tuturor elementelor, caz în care rezultatul final este T. Cum această rețetă se poate aplica pentru orice funcție F ce realizează prelucrarea într-un anume fel a tuturor elementelor unei liste, apare ideea de a scrie o funcție **generică** (șablon) GEN ce respecta “rețeta”, având doi parametri: funcția ce va realiza prelucrarea și lista asupra căreia se va acționa.

Terminologia adoptată pentru astfel de funcții (sau similare) este cea de **generatori**. Nu este necesar ca un generator să întoarcă T sau NIL. În funcție de implementare se pot concepe generatori care să întoarcă, de exemplu, lista tuturor rezultatelor non-NIL ale lui F culese până în momentul încetării acțiunii generatorului (L este vidă sau F întoarce NIL).

Model recursiv

$$\text{GEN}(F, l_1 l_2 \dots l_n) = \begin{cases} \text{adevarat} & \text{daca } l = \emptyset \\ \text{fals} & \text{dacă } \neg F(l_1) \\ \text{GEN}(F, l_2 \dots l_n) & \text{altfel} \end{cases}$$

```

(DEFUN GEN (F L)
  (COND
    ((NULL L) T)
    (T (AND (FUNCALL F (CAR L)) (GEN F (CDR L))))
  )
)

```

După ce a fost definit generatorul, se poate defini funcția VERIF (L) astfel:

```

(DEFUN VERIF (L)
  (GEN #'(LAMBDA (L)
    (GEN #'ATOM L)
  )
  L
)
)

```

## 2. 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

#### Exemplu 1

```

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

```

- (f 1 2) → 3
- (f 3) → 3

## Exemplu 2

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

- $(g\ 1\ 2\ 3) \rightarrow 6$
- $(g\ 4\ 5) \rightarrow 9$
- $(g\ 3) \rightarrow 3$

## Exemplu 3

```
(DEFUN F (L1 &REST L2)
  (COND
    ((NULL (CAR L2)) NIL)
    (T (CONS (MAPCAR #'* L1 (CAR L2)) (F L1 (CADR L2)))))
)
```

- $(F\ '(1\ 2)\ '(3\ 4)\ '(5\ 6))$  se evaluează la  $((3\ 8)\ (5\ 12))$
- **Exemplu 4**

Să presupunem că dorim să construim o funcție care să adune 1 la valoarea unei expresii:

```
(DEFUN INC (NUMAR)
  (+ NUMAR 1)
)
```

Această funcție se va utiliza în felul următor:

- $(INC\ 10)$  se va evalua la 11

Sigur că, în măsura în care dorim, putem adăuga funcții asemănătoare și pentru alte valori de incrementare. Alternativ, putem să rescriem funcția INC astfel încât să accepte un al doilea parametru:

```
(DEFUN INC (NUMAR INCREMENT)
  (+ NUMAR INCREMENT)
)
```

În marea majoritate a cazurilor, totuși, vom avea nevoie de incrementarea cu 1 a valorii expresiei corespunzătoare lui NUMAR. În această situație putem folosi facilitatea argumentelor opționale. Iată în continuare definiția funcției INC cu un argument opțional:

```
(DEFUN INC (NUMAR &OPTIONAL INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
```

```

        (T (+ NUMAR INCREMENT))
    )
)

```

Caracterul & din &OPTIONAL semnaleză faptul că &OPTIONAL este un separator de parametri, nu un parametru propriu-zis. Parametrii care urmează după &OPTIONAL sunt legați la intrarea în procedură la fel ca și ceilalți parametri. Dacă nu există parametru actual corespunzător, valoarea unui parametru formal opțional este considerată NIL. Iată câteva exemple de utilizare a funcției INC pe care tocmai am construit-o:

- (INC 10) se evaluează la 11
- (INC 10 3) se evaluează la 13
- (INC (\* 10 2) 5) se evaluează la 25

Putem, de asemenea, să luăm în considerare o valoare implicită pentru parametrii opționali. Iată o definiție a funcției INC în care parametrul opțional are valoarea implicită 1:

```

(DEFUN INC (NUMAR &OPTIONAL (INCREMENT 1))
  (+ NUMAR INCREMENT)
)

```

Observați că de această dată în locul parametrului opțional este precizată o listă formată din două elemente: primul element este numele parametrului opțional, iar al doilea element este valoarea cu care se inițializează acesta atunci când argumentul actual corespunzător nu este prezent.

Să notăm că putem avea oricâte argumente opționale. Toate aceste argumente vor fi trecute în lista parametrilor după simbolul &OPTIONAL, care va apare o singură dată. De asemenea, putem avea un singur argument opțional semnalat prin &REST, a cărui valoare devine lista tuturor argumentelor date cu excepția celor care corespund parametrilor obligatorii și opționali. Fie o nouă versiune a funcției INC:

```

(DEFUN INC (NUMAR &REST INCREMENT)
  (COND
    ((NULL INCREMENT) (+ NUMAR 1))
    (T (+ NUMAR (APPLY #'INCR INCREMENT))))
)

```

Iată câteva exemple de aplicare a acestei funcții:

- (INC 10) se evaluează la 11
- (INC 10 1 2 3) se evaluează la 16

### 3. 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.*

## 4. 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:

- parametrii 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.

Să vedem mai întâi un exemplu comparativ. Fie următoarele definiții și evaluări:

<pre>&gt; (DEFUN F (N)   (PRINT N)   ) &gt; (SETQ N 10) &gt; (F N) 10 10</pre> <p><b>Notă</b> Parametrul N este evaluat de la bun început, PRINT va tipări valoarea acestuia, și o va întoarce ca valoare a apelului funcției.</p>	<pre>&gt; (DEFMACRO G (N)   (PRINT N)   ) &gt; (SETQ N 10) &gt; (G N) N 10</pre> <p><b>Notă</b> În primul moment nu se face o încercare de a se evalua parametrul N. Ca atare, PRINT va tipări N, după care îl va întoarce pe N ca valoare a formei intermediare a macrodefiniției. Apoi această valoare intermediară este evaluată și se va produce 10.</p>
--	--

Ca un alt exemplu, dorim să producem o macrodefiniție DEC care să decrementeze cu 1 valoarea parametrului ei, ca în exemplul următor:

- (SETF X 10) se evaluează la 10
- (DEC X) va produce 9
- X se evaluează la 9

Aceasta înseamnă că forma intermediară va trebui să fie

(SETQ parametru (- parametru 1))

Iată o posibilitate:

```
(DEFMACRO DEC (N)
  (LIST 'SETQ N (LIST '- N 1))
)
```

## 5. 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)

Folosind apostroful invers, macrodefiniția DEC se va scrie mai simplu astfel:

```
(DEFMACRO DEC (N)
  `(SETQ ,N (- ,N 1))
)
```

Să observăm de asemenea că utilizarea apostrofului invers ușurează tipărirea rezultatelor intermediare.

## 6. Forme iterative

Limbajul Lisp prevede un set de forme iterative care permit controlul execuției într-un stil apropiat de cel cunoscut din limbajele clasice imperative.

### Forma PROG

#### (PROG I f<sub>1</sub> ... f<sub>n</sub>) : e

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- I este o listă de inițializare conținând elemente de forma  
    “(variabilă valoare)” sau doar “variabilă”  
    În primul caz variabila este inițializată cu valoarea specificată, iar în al doilea caz primește valoarea NIL.
- variabilele sunt temporare și locale lui PROG
- Corpul funcției PROG este alcătuit din formele f<sub>1</sub>, ..., f<sub>n</sub> care sunt evaluate secvențial.
- La întâlnirea sfârșitului corpului funcției PROG se returnează implicit NIL. Pentru returnarea unei alte valori trebuie utilizată forma

#### (RETURN e) : e

- se evaluează argumentul
- rezultatul este valoare argumentul

Controlul evaluării formelor din corpul lui PROG se face folosind forma

#### (GO f)

- f este un atom etichetă sau o formă evaluabilă la un atom etichetă.
- permite reluarea evaluării lui PROG de la un punct specificat

### Exemplu

Să se determine lungimea unei liste în număr de elemente la nivel superficial.

(lungime '(1 (2 (3)) (4))) → 3

```
(defun lungime(l)
  (prog ((lung 0) (lista l))
    et
    (cond
      ((null lista) (return lung))
      (t (setq lung (+ lung 1))
         (setq lista (cdr lista))
         (go et))
```



## Forma DO

**(DO l<sub>1</sub> l<sub>2</sub> f<sub>1</sub> ... f<sub>n</sub>) : e**

- evaluarea argumentelor face parte din procesul de evaluare al funcției
- lista **l<sub>1</sub>** numită **listă de inițializare** are forma

```
( (var1 val_init1 pas1)  
  ...  
  (varn val_initn pasn)  
)
```

efectul fiind legarea simbolurilor **var<sub>i</sub>** de valorile inițiale **val\_init<sub>i</sub>** corespunzătoare, sau NIL dacă acestea nu se specifică; de remarcat că această operație de legare are loc în același moment pentru toate variabilele;

- lista **l<sub>2</sub>** numită **specificația de terminare** are forma  
(test\_terminare g<sub>1</sub> ... g<sub>n</sub>)
- corpul ciclului, alcătuit din formele ce se vor evalua repetat (f<sub>1</sub> ... f<sub>n</sub>) ca efect al ciclării

Un ciclu iterativ se desfășoară astfel:

- (1) se evaluează forma **test\_terminare** (deci este vorba despre un ciclu cu test inițial, de tip WHILE). La evaluarea diferită de NIL a testului clauzei se vor evalua în ordine formele de ieșire g<sub>i</sub> și se va returna rezultatul ultimei evaluări, g<sub>i</sub> (dacă nu există g<sub>n</sub> se va returna NIL).
- (2) dacă **test\_terminare** se evaluează la NIL atunci se trece la evaluarea formelor ce formează corpul lui DO, adică f<sub>1</sub>, ..., f<sub>n</sub>.
- (3) La întâlnirea sfârșitului corpului funcției fiecărei variabile index **var<sub>i</sub>** îi este asignată valoarea formelor de actualizare **pas<sub>i</sub>** corespunzătoare. Dacă forma **pas<sub>i</sub>** lipsește, variabila **var<sub>i</sub>** nu se va mai actualiza, rămânând cu valoarea pe care o are. Operația de atribuire are loc în același moment pentru toate variabilele. Ciclul se reia începând cu (1).

**Ieșirea din DO se poate face la orice moment prin intermediul evaluării unei forme RETURN.**

### Exemplu

Să se determine lungimea unei liste în număr de elemente la nivel superficial.

(lungime '(1 (2 (3)) (4))) → 3

```
(defun lungime(l)
  (do ((lista l (cdr lista)) ;lista de inițializare
      (lung 0 (+ lung 1))
      )
    ((null lista) lung) ;specificația de terminare
  ) ;DO un are corp
)
```

*Folosind formele iterative, prelucrarea de liste se realizează la nivel superficial.*

## Alte aspecte ale programării funcționale

### 7. Parametrii Lisp în context “Dynamic Scoping”

Așa cum am văzut până acum, toate aparițiile parametrilor formali sunt înlocuite cu valorile argumentelor corespunzătoare. Deci, avem o variantă de apel prin nume, apelul prin text, deoarece în Lisp avem de-a face cu determinarea dinamică a domeniului de vizibilitate (dynamic scoping) simultan cu înlocuirea textuală a valorilor evaluate.

Deși parametrul se leagă de argument (în sensul apelului prin referință din limbajele imperative, adică numele argumentului și cel al parametrului devin sinonime), nu este apel prin referință, deoarece modificările valorilor parametrilor formali nu afectează valoarea argumentelor (ceea ce amintește de apelul prin valoare).

De exemplu:

- (DEFUN FCT(X) (SETQ X 1) Y) se evaluează la FCT
- (SETQ Y 0) se evaluează la 0
- (FCT Y) se evaluează la 0

În secvența de mai sus Y se evaluează la 0, iar X se leagă la 0. Apelul FCT este astfel echivalent cu (FCT 0). X e parametru formal, Y e parametru actual, și ca atare modificarea lui X nu afectează pe Y, deci NU avem de-a face cu apel prin referință.

Aceasta demonstrează că transmiterea de parametri în LISP nu se face prin referință. Cum justificăm însă că nu este nici apel prin valoare? Există totuși situații în care valoarea parametrului actual poate fi modificată, de exemplu prin acțiunea funcțiilor cu efect distructiv RPLACA și RPLACD.

Să mai remarcăm faptul că în ciuda caracteristicii “Dynamic scoping”, variabila Y nu se leagă la execuție de parametrul formal tocmai înlocuit cu Y, adică Y-ul “intern” funcției, ci își păstrează caracteristica de variabilă globală pentru FCT.

Corpul unei funcții LISP este domeniul de vizibilitate a parametrilor săi formali, care se numesc variabile legate în raport cu funcția respectivă. Celelalte variabile ce apar în definiția funcției se numesc variabile libere.

Contextul curent al execuției este alcătuit din toate variabilele din program împreună cu valorile la care sunt legate acestea în acel moment. Acest concept este necesar pentru stabilirea valorii variabilelor libere care intervin în evaluarea unei funcții, evaluarea în LISP făcându-se într-un context dinamic (dynamic scoping), prin ordinea de apel a funcțiilor, și nu static, adică relativ la locul de definire.

Să reamintim în acest scop diferența între determinarea statică și respectiv cea dinamică a domeniului de vizibilitate în cadrul limbajelor imperative. Fie programul Pascal:

```
var
    a:integer;

procedure P;
begin
    writeln(a);
end;

procedure Q;
var
    a: integer;
begin
    a := 7;
    P;
end;

begin
    a := 5;
    Q;
end.
```

**Determinarea statică a domeniului de vizibilitate** (DSDV, static scoping) presupune că procedura P acționează în mediul de definire și ca urmare ea va “vedea” întotdeauna variabila a globală. Este și cazul limbajului Pascal, situație în care P apelat în Q va tipări 5 și nu 7.

Pe de altă parte, dacă am avea de-a face cu **determinarea dinamică a domeniului de vizibilitate** (DDDV, dynamic scoping), procedura P va tipări întotdeauna ultima (în ordinea apelurilor) variabilă *a* definită (sau legată, în terminologie Lisp). În acest caz se va tipări 7, deoarece ultima legare a lui *a* este relativ la valoarea 7.

Limbajul Lisp dispune de DDDV, această abordare fiind mai naturală decât cea statică pentru cazul sistemelor bazate pe interpretoare. O variantă Lisp a programului de mai sus care pune în evidență DDDV este:

```
(DEFUN P () A)
(DEFUN Q ()
  (SETQ A 7))
```

```

(P)
)
(SETQ A 5)
(SETQ Y (LIST (P) (Q)))
(PRINT Y)

```

Se va tipări lista (5 7), valoarea întoarsă de P fiind de fiecare dată ultimul A legat.

Avantajul legării dinamice este adaptabilitatea, adică o flexibilitate sporită. În cazul argumentelor funcționale însă pot să apară interacțiuni nedorite. Pentru a ilustra genul de probleme care pot apărea să luăm exemplul formei funcționale *twice*, care aplica de două ori argumentul funcției unei valori:

```
(DEFUN twice (func val) (funcall func (funcall func val)))
```

Exemple de aplicare:

- (twice 'add1 5) returnează 7
- (twice '(lambda (x) (\* 2 x)) 3) returnează 12

Dacă se întâmplă însă să folosim identificatorul *val* și în cadrul lui *func*, ținând cont de DDDV, apare o coliziune de nume, cu următorul efect:

- (setq val 2) returnează 2
- (twice '(lambda (x) (\* val x)) 3) returnează 27, nu 12

(cum probabil am dori să obținem). Aceasta deoarece ultima legare (dinamică deci) a lui *val* s-a efectuat ca parametru formal al funcției *twice*, deci *val* a devenit 3.

Această problemă a fost denumită problema FUNARG (funcțional argument). Este o problemă de conflict între DSDV și DDDV. Rezolvarea ei nu a constat în renunțarea la DDDV pentru LISP ci în introducerea unei forme speciale numite **function**, care realizează legarea unei lambda expresii de mediul ei de definire (deci tratarea aceluși caz în mod static). Astfel,

- (twice (function (lambda (x) (\* val x))) 3) se evaluează la 12

Concluzia este deci că LISP are două reguli de DDV: DDDV implicit și DSDV prin intermediul construcției *function*.

## 8. Universul expresiilor și universul instrucțiunilor

Orice limbaj de programare poate fi împărțit în două așa-numite universuri (domenii):

1. universul expresiilor;
2. universul instrucțiunilor.

**Universul expresiilor** include toate construcțiile limbajului de programare al căror scop este producerea unei valori prin intermediul procesului de evaluare.

**Universul instrucțiunilor** include instrucțiunile unui limbaj de programare. Acestea sunt de două feluri:

(i) instrucțiuni ce influențează fluxul de control al programului:

- instrucțiuni conditionale;
- instrucțiuni de salt;
- instrucțiuni de ciclare;
- apeluri de proceduri și funcții.

(ii) instrucțiuni ce alterează starea memoriei:

- atribuirea (alterează memoria internă, primară);
- instrucțiuni de intrare/ieșire (alterează memoria externă, secundară)

Ca asemănare a acestor două universuri, putem spune că ambele alterează ceva. Există însă deosebiri importante. În universul instrucțiunilor ordinea în care se execută instrucțiunile este de obicei esențială. Adică instrucțiunile

$$i := i + 1; a := a * i;$$

au efect diferit fata de instrucțiunile

$$a := a * i; i := i + 1;$$

Fie

$$z := (2 * a * y + b) * (2 * a * y + c);$$

Multe compilatoare elimină evaluarea redundantă a subexpresiei comune  $2 * a * y$  prin următoarea substituție:

$$t := 2 * a * y; z := (t + b) * (t + c);$$

Această înlocuire s-a putut efectua în universul expresiilor deoarece în cadrul unei expresii o subexpresie va avea întotdeauna aceeași valoare.

În cadrul universului instrucțiunilor situația se schimbă, datorită posibilelor efecte secundare. De exemplu, pentru secvența

$$y := 2 * a * y + b; z := 2 * a * y + c;$$

factorizarea subexpresiei comune furnizează secvența neechivalentă

$$t := 2 * a * y; y := t + b; z := t + c;$$

Deși un compilator poate analiza un program pentru a determina pentru fiecare subexpresie în parte dacă poate fi factorizată sau nu, acest lucru cere tehnici sofisticate de analiză globală a fluxului (global flow analysis). Astfel de analize sunt costisitoare și dificil de implementat. Concluzionăm deci că universul expresiilor prezintă un avantaj asupra universului instrucțiunilor, cel puțin referitor la acest aspect al eliminării subexpresiilor comune (există, oricum, și alte avantaje care vor fi evidențiate în continuare).

Scopul programării funcționale este extinderea la nivelul întregului limbaj de programare a avantajelor pe care le promoveaza universul expresiilor față de universul instrucțiunilor.

## 9. Independența ordinii de evaluare

A evalua o expresie înseamnă a-i extrage valoarea. Evaluarea expresiei  $6 * 2 + 2$  furnizează valoarea 14. Evaluarea expresiei  $E = (2ax + b)(2ax + c)$  nu se poate face până nu precizăm valorile numerelor  $a$ ,  $b$ ,  $c$  și  $x$ . Deci, valoarea acestei expresii este dependentă de contextul evaluării. Odată acest lucru precizat, mai este important să subliniem că valoarea expresiei  $E$  nu depinde de ordinea de evaluare (adică de înlocuire a numerelor, mai întâi primul factor sau cel de-al doilea, etc). Este posibilă chiar evaluarea paralelă. Aceasta deoarece în cadrul expresiilor pure (așa cum sunt cele matematice) evaluarea unei subexpresii nu poate afecta valoarea nici unei alte subexpresii.

**Definiție.** O expresie pură este o expresie liberă de efecte secundare, adică nu conține atribuiri nici explicit (gen C) și nici implicit (apeluri de funcții).

Această proprietate a expresiilor pure, și anume independența ordinii de evaluare, se numește proprietatea Church-Rosser. Ea permite construirea de compilatoare ce aleg ordini de evaluare care fac uz într-un mod cât mai eficient de resursele masinii. Să subliniem din nou potențialul de paralelism etalat de această proprietate.

Proprietatea de transparență referențială presupune că într-un context fix înlocuirea subexpresiei cu valoarea sa este complet independentă de expresia înconjurătoare. Deci, odata evaluată, o subexpresie nu va mai fi evaluată din nou pentru că valoarea sa nu se va mai schimba. Transparența referențială rezultă din faptul că operatorii aritmetici nu au memorie, astfel orice apel al unui operator cu aceleași intrări va produce același rezultat.

Una dintre caracteristicile notației matematice este interfața manifestă, adică, conexiunile de intrare ieșire între o subexpresie și expresia înconjurătoare sunt vizual evidente (de exemplu în expresia  $3 + 8$  nu există intrări “ascunse”) și ieșirile depind numai de intrări. Producătorii de efecte secundare, deci și funcțiile în general, nu au interfață manifestă ci o interfață nemanifestă (hidden interface).

Să trecem în revistă proprietățile expresiilor pure:

- valoarea este independentă de ordinea de evaluare;
- expresiile pot fi evaluate în paralel;
- transparența referențială;
- lipsa efectelor secundare;
- intrările și efectele unei operații sunt evidente.

Scopul programării funcționale este extinderea tuturor acestor proprietăți la întreg procesul de programare.

Programarea aplicativă are o singură construcție sintactică fundamentală și anume aplicarea unei funcții argumentelor sale. Modurile de definire a funcțiilor sunt următoarele:

1. **Definire enumerativă.** Este posibilă doar când funcția are un domeniu finit de dimensiune redusă.
2. **Definire prin compunere de funcții deja definite.** Dacă e cazul unei definiri în termeni de un număr infinit sau neprecizat de compuneri, o astfel de metodă este utilă doar dacă se poate extrage un principiu de regularitate, principiu care să ne permită generarea cazurilor încă neenumerate pe baza celor specificate. Dacă un astfel de principiu există suntem în cazul unei definiții recursive. În programarea funcțională recursivitatea este metoda de bază pentru a descrie un proces iterativ.

O altă distincție ce trebuie făcută relativ la definirea de funcții se referă la definiții explicite și implicite. O definiție explicită ne spune ce este un lucru. O definiție implicită statuează anumite proprietăți pe care le are un anumit obiect.

Într-o definiție explicită variabila definită apare doar în membrul stâng al ecuației (de exemplu  $y = 2ax$ ). Definițiile explicite au avantajul că pot fi interpretate drept reguli de rescriere (reguli care specifică faptul că o clasă de expresii poate fi înlocuită de alta).

O variabilă este definită implicit dacă ea este definită de o ecuație în care ea apare în ambii membri (de exemplu  $2a = a + 3$ ). Pentru a-i găsi valoarea trebuie rezolvată ecuația.

Aplicarea repetată a regulilor de rescriere se termină întotdeauna. Este posibil însă ca anumite definiții implicite să nu ducă la terminare (adică ele nu definesc nimic). De exemplu, ecuația fără soluție  $a = a + 1$  duce la un proces infinit de substituție.

Un avantaj al programării funcționale este că, la fel ca și în cazul algebrei elementare, ea simplifică transformarea de definiții implicite în definiții explicite. Acest lucru este foarte important deoarece specificațiile formale ale sistemelor soft au de obicei forma unor definiții implicite, însă conversia specificațiilor în programe are loc mai ușor în cazul definițiilor explicite.

Să mai subliniem că definițiile recursive sunt prin natura lor implicite. Totuși, datorită formei lor regulate de specificare (adică membrul stâng e format numai din numele și argumentele funcției) ele pot fi folosite ca reguli de rescriere. Condiția de oprire asigură terminarea procesului de substituții.

Față de limbajele conventionale, limbajele funcționale diferă și în ceea ce privește modelele operaționale utilizate pentru descrierea execuției programelor. Nu este proprie de exemplu pentru limbajele funcționale urmărirea execuției pas cu pas, deoarece nu contează ordinea de evaluare a subexpresiilor. Deci nu avem nevoie de un depanator în adevăratul înțeles al cuvântului.

Totuși, limbajul Lisp permite urmărirea execuției unei forme Lisp. Pentru aceasta există macrodefiniția TRACE care primește ca argument numele funcției dorite, rezultatul întors de TRACE fiind T dacă funcția respectivă există și NIL în caz contrar.