

ACADEMIA DE STUDII ECONOMICE BUCUREȘTI  
FACULTATEA DE CIBERNETICĂ, STATISTICĂ ȘI INFORMATICĂ ECONOMICĂ  
CATEDRA DE INFORMATICĂ ECONOMICĂ

## **Bazele programării**

**Autori: Bogdan Ghilic-Micu, Marian Stoica, Marinela Mircea**

Acest material are la bază lucrarea  
*Bazele programării calculatoarelor. Teorie și aplicații în C,*  
Ion Gh. Roșca, Bogdan Ghilic-Micu, Cătălina Cocianu, Marian Stoica, Cristian Uscatu, Marinela  
Mircea, Lorena Bătăgan, Cătălin Silvestru  
Editura ASE, București, 2006, ISBN 973-594-591-6

București 2010

# TITLUL CURSULUI: Bazele programării

## INTRODUCERE

Cursul de *Bazele programării* se adresează studenților înscriși la programul de studiu ID, organizat de facultatea Cibernetică, Statistică și Informatică Economică și face parte din planul de învățământ aferent anului I, semestrul II. Pentru o bună înțelegere a noțiunilor teoretice și practice prezentate în acest curs, este recomandat să se fi parcurs anterior disciplina *Bazele tehnologiei informației*.

**OBIECTIVELE PRINCIPALE** ale acestui curs, vizează însușirea de către studenți a următoarelor elemente:

- noțiuni de bază în teoria programării;
- metodele de analiză a problemelor în vederea rezolvării lor cu calculatorul;
- logica elaborării algoritmilor structurați și modularizați;
- realizarea programelor în limbajul C.

Cursul *Bazele programării este structurat* în patru unități de învățare, corespunzătoare elementelor principale studiate. În cadrul procesului de instruire pot fi utilizate ca resurse suplimentare materialele puse la dispoziție de bibliotecile Academiei de Studii Economice, precum și laboratoarele catedrei de Informatică economică, cu o programare prealabilă și atunci când nu se desfășoară ore.

**EVALUAREA CUNOȘTINȚELOR**, respectiv stabilirea notei finale, se va realiza astfel:

- lucrare de control;
- un referat despre operațiile de intrare/ieșire efectuate cu tastatura/monitorul;
- un proiect, care va conține minim 20 de programe în limbajul C, lucru cu masive de date, însoțite de un dosar de prezentare.

Lucrarea va fi susținută în cadrul ultimei activități asistate. Referatul și proiectul se susțin numai în timpul activității didactice (activităților asistate), conform calendarului disciplinei.

Stabilirea notei finale se va realiza astfel:

- lucrarea de control constituie 50% din nota finală;
- referatul constituie 10% din nota finală;
- proiectul constituie 30% din nota finală;
- din oficiu se acordă 10%.

## Cuprins

1. Algoritmi și scheme logice .....	4
Obiectivele unității de învățare 1 .....	4
1.1. Caracteristicile și reprezentarea algoritmilor .....	4
1.2. Descrierea structurilor fundamentale .....	10
1.3. Structurarea și proiectarea algoritmilor .....	13
Răspunsuri și comentarii la testele de autoevaluare .....	17
Bibliografia unității de învățare .....	17
2. Organizarea internă a datelor .....	18
Obiectivele unității de învățare 2 .....	18
2.1. Informația, data și reprezentarea internă a datelor .....	18
2.2. Structuri de date .....	27
Răspunsuri și comentarii la testele de autoevaluare .....	39
Bibliografia unității de învățare .....	39
3. Etapele rezolvării problemelor cu calculatorul .....	40
Obiectivele unității de învățare 3 .....	40
3.1. Caracteristici generale ale PPAD .....	40
3.2. Fazele dezvoltării programelor .....	44
Răspunsuri și comentarii la testele de autoevaluare .....	46
Bibliografia unității de învățare .....	46
4. Caracteristicile limbajului C .....	47
Obiectivele unității de învățare 4 .....	47
4.1 Elementele de bază ale limbajului C .....	47
4.2 Tipurile de date în C .....	51
4.3 Expresii .....	60
4.4. Realizarea structurilor fundamentale de control .....	68
Răspunsuri și comentarii la testele de autoevaluare .....	74
Bibliografia unității de învățare .....	75
Bibliografie .....	75

## 1. Algoritmi și scheme logice

### Cuprins

Obiectivele unității de învățare 1

1.1. Caracteristicile și reprezentarea algoritmilor

1.2. Descrierea structurilor fundamentale

1.3. Structurarea și proiectarea algoritmilor

Răspunsuri și comentarii la testele de autoevaluare

Bibliografia unității de învățare

### Obiectivele unității de învățare 1

Dupa studiul acestei unitati de învățare, studenții vor avea cunoștințe teoretice și abilități practice despre:

- ➡ caracteristicile algoritmilor;
- ➡ reprezentarea algoritmilor;
- ➡ descrierea structurilor fundamentale;
- ➡ structurarea algoritmilor;
- ➡ proiectarea algoritmilor.



*Durata medie a unității de studiu individual - 8 ore*

### 1.1. Caracteristicile și reprezentarea algoritmilor

Algoritmul desemnează o mulțime exhaustivă și univoc determinată de operații, împreună cu succesiunea în care trebuie aplicate asupra datelor inițiale ale problemei, pentru a obține soluția.

**Principalele caracteristici ale unui algoritm sunt:**

❖ **Generalitate:** un algoritm nu trebuie conceput pentru o problemă particulară, ci pentru o clasă generală de probleme.



**Exemplu:**

Nu se va concepe un algoritm pentru rezolvarea ecuației particulare  $5x^2 - 2x = 7$ , ci un algoritm pentru rezolvarea ecuației de gradul al doilea cu o necunoscută  $ax^2 + bx + c = 0$ , cu  $a, b, c, x \in \mathbb{R}$ ,  $a \neq 0$  sau, mai general, pentru rezolvarea ecuației de forma  $ax^2 + bx + c = 0$ , cu parametrii  $a, b, c, x \in \mathbb{R}$ .

❖ **Determinare (claritate):** algoritmul trebuie să prevadă modul de soluționare a tuturor situațiilor care pot apărea în rezolvarea problemei respective, într-o manieră fără ambiguități sau neclarități, lucru impus de caracterul de automat al calculatorului electronic.

### Exemplu:

Să se elaboreze algoritmul pentru rezolvarea ecuației:  $ax^2 + bx + c = 0$ ,  $a, b, c, x \in \mathbb{R}$ . Analiza arată că există patru situații posibile care trebuie cuprinse în algoritm:

1.  $a \neq 0$ , ecuație de gradul II;
2.  $a = 0$  și  $b \neq 0$ , ecuație de gradul I;
3.  $a = 0$  și  $b = 0$  și  $c \neq 0$ , ecuație imposibilă;
4.  $a = 0$  și  $b = 0$  și  $c = 0$ , nedeterminare.

Rămâne să fie rezolvate fiecare din aceste situații sau unele din ele să fie grupate în funcție de cerințe.

● **Finitudine:** operațiile trebuie astfel concepute încât algoritmul să se termine într-un număr finit de pași, cunoscut sau necunoscut.

### Exemplu:

✖ Pentru însumarea tuturor elementelor unui vector **B**, de dimensiune **n**, formula de recurență  $S = S + b_i$  se repetă de **n** ori (număr cunoscut de pași);

✖ La determinarea celui mai mare divizor comun (CMMDC) dintre două numere întregi (A și B) se împarte A la B:  $A = B \cdot Q + R$ , apoi se continuă împărțirea împărțitorului (B) la rest (R), până când se obține un rest nul, caz în care CMMDC este ultimul împărțitor (număr necunoscut de pași).

## Descrierea algoritmilor. Iterativitate și recursivitate

*Iterativitatea* este procesul prin care rezultatul este obținut ca urmare a execuției repetate a unui set de operații, de fiecare dată cu alte valori de intrare. Numărul de iterații poate fi cunoscut sau necunoscut, dar determinabil pe parcursul execuției. Indiferent de situație, numărul de iterații trebuie să fie totdeauna finit. În cazul în care numărul de repetări nu este cunoscut inițial, oprirea ciclării se face combinând instrucțiunile de calcul cu instrucțiunile de control. Condițiile finale ce trebuie îndeplinite pentru terminarea ciclării se bazează pe rezultatele calculelor până la un moment dat, de cele mai multe ori fiind corelate cu *viteza de convergență* a calculelor către o valoare stabilă și, implicit, cu *gradul de aproximare* impus unui rezultat.

Utilizarea iterativității în descrierea algoritmilor este uneori o cale simplificatoare, alteori singura alternativă de a preciza modul de desfășurare a unui proces de calcul. În general, când numărul de iterații nu este cunoscut sau este variabil de la o execuție la alta, singura modalitate de descriere a algoritmului presupune iterativitatea. Spre exemplu, ridicarea la pătrat a elementelor unui vector cu 3 elemente se poate descrie prin următoarea succesiune de operații:

$$V(1) = V(1) \cdot V(1)$$

$$V(2) = V(2) \cdot V(2)$$

$$V(3) = V(3) \cdot V(3)$$

Realizarea aceluiași lucru pentru un vector **n**-dimensional, cu **n** variind de la o execuție la alta, nu este posibilă în această manieră, deoarece un algoritm este corect dacă are precizați toți pașii. Singura modalitate de realizare utilizează operația  $V(I) = V(I) \cdot V(I)$ , plasată într-un ciclu variind după **I**, de la **1** la o valoare finală **N**, dată explicit de utilizator la fiecare execuție a programului.

*Recursivitatea* este procesul iterativ prin care valoarea unei variabile se determină pe baza uneia sau mai multora dintre propriile ei valori anterioare. După cum valoarea curentă a variabilei depinde de una sau mai multe valori anterioare, procesul este *unirecursiv*, respectiv *multirecursiv*.

Recursivitatea presupune definirea uneia sau mai multor *formule de start* (în funcție de numărul valorilor anterioare de care depinde valoarea curentă) și a unei *formule recursive* (de recurență). Recursivitatea apare și în cazuri dintre cele mai simple, cum ar fi numărarea, factorialul, însumarea sau înmulțirea elementelor unui șir.

### **Exemple:**

✖ A număra înseamnă a adăuga o unitate la valoarea anterior calculată. Dacă se notează  $v_0 = 0$  valoarea inițială, numărarea se desfășoară după următorul proces recursiv:

$$v_1 = v_0 + 1$$

$$v_2 = v_1 + 1$$

$$v_3 = v_2 + 1$$

.....

$$v_i = v_{i-1} + 1$$

.....

$$v_n = v_{n-1} + 1$$

Valoarea care interesează este  $v_n$ . În procesul descris anterior se disting:

$$v_0 = 0 \quad \blacktriangleright \text{ formula de start}$$

$$v_i = v_{i-1} + 1 \quad \blacktriangleright \text{ formula recursivă, pentru } i=1, n$$

Când valorile obținute prin numărare nu necesită păstrarea în memorie, nu sunt necesare mai multe locații, ci una singură, care va fi suprascrisă la fiecare iterație:

$$V = 0 \quad \blacktriangleright \text{ formula de start}$$

$$V = V + 1 \quad \blacktriangleright \text{ formula recursivă, pentru } i=1, n$$

Utilizarea mai multor zone de memorie pentru calculele intermediare devine obligatorie în situația multirecursivității. Practic, prin transferul dinamic al valorilor intermediare, sunt necesare atâtea zone de memorie câte valori precedente sunt necesare calculului termenului curent.

### **Exemplu:**

Algoritmul lui Euclid pentru determinarea celui mai mare divizor comun a două numere **A** și **B**.

Restul curent se calculează pornind de la ultimul și penultimul rest, după relația:  $r_{k+2} = r_k - \left\lfloor \frac{r_k}{r_{k+1}} \right\rfloor \cdot r_{k+1}$ ,

cu  $r_0 = A$ ;  $r_1 = B$  și  $k = 0, 1, 2, \dots$ . Prin  $\lfloor \ ]$  s-a notat partea întreagă a expresiei. Ultimul rest nenul reprezintă cel mai mare divizor comun. Dacă se notează cu **D** variabila deîmpărțit, cu **I** variabila împărțitor și cu **R** restul, formulele de start devin:  $D \leftarrow a$ ;  $I \leftarrow b$ . Iterativ, se calculează restul  $R \leftarrow D - \left\lfloor \frac{D}{I} \right\rfloor \cdot I$  și se execută

transferurile:  $D \leftarrow I$ ;  $I \leftarrow R$ . Împărțitorul devine noul deîmpărțit, iar ultimul rest obținut devine împărțitor în iterația următoare a algoritmului. Procesul iterativ continuă până se obține restul zero. Ultimul rest nenul reprezintă cel mai mare divizor comun al numerelor **A** și **B**.



### **Teste de autoevaluare**

**1.** Caracteristicile oricărui algoritm sunt: 1. Generalitate; 2. Complementaritate; 3. Claritate; 4. Finitudine; 5. Recursivitate; 6. Iterativitate.

a) toate; b) 1,3,4,5 și 6; c) 1,2,3 și 4; d) 1,3 și 4; e) 1,2,5 și 6.

**2.** Un algoritm recursiv este: a) un algoritm care se autoapelează ; b) un proces repetitiv static; c) un proces repetitiv dinamic; d) un proces repetitiv prin care valoarea unei variabile se determină pe baza a cel puțin uneia dintre valorile ei anterioare; e) un proces alternativ prin care valoarea unei variabile se determină pe baza a cel puțin uneia dintre valorile ei anterioare.

### ➤ Reprezentarea algoritmilor prin scheme logice

Practica programării calculatoarelor dovedește că schema logică este forma cea mai utilizată de reprezentare a algoritmilor. Ea se dovedește utilă, în special, în inițierea în programare, deoarece oferă o „vizualizare” mai bună, o redare expresivă și sintetică, ușor de urmărit, a algoritmilor. Să considerăm un graf orientat în care arcele sunt etichetate cu anumite informații formând așa-zisele blocuri. În graf sunt admise următoarele blocuri:

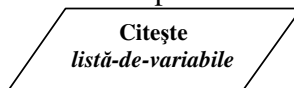
❶ *Blocul START* este un arc etichetat cu cuvântul START, pentru care vârful inițial nu este pus explicit în evidență, deoarece în el nu pot sosi arce:

START

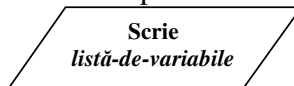
❷ *Blocul STOP* este un arc etichetat cu cuvântul STOP, pentru care vârful final nu este pus explicit în evidență, deoarece din el nu pot pleca arce:

STOP

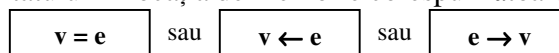
❸ *Blocul de citire* este un arc etichetat cu informația: citirea unor valori de pe suportul extern și înscrierea lor în locații de memorie corespunzătoare unor variabile.



❹ *Blocul de scriere* este un arc etichetat cu informația: înscrierea pe suportul extern a valorilor memorate în locațiile de memorie corespunzătoare unor variabile:



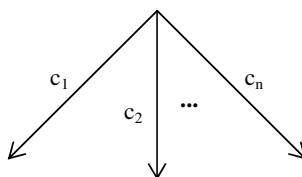
❺ *Blocul de atribuire* este un arc etichetat cu informația: evaluarea unei expresii  $e$  și înscrierea (atribuirea) rezultatului în locația de memorie corespunzătoare unei variabile  $v$ :



❻ *Blocul de ramificare* (selecție) este o mulțime de  $n$  arce care pleacă din același vârf, arce etichetate cu predicatele  $C_1, C_2, \dots, C_n$  (predicat = condiție = expresie logică, care poate fi adevărată (1) sau falsă (0)), care satisfac relațiile:

$$C_1 \vee C_2 \vee \dots \vee C_n = 1; C_i \wedge C_j = 0, (\forall) i \neq j; i, j = \overline{1, n}$$

Relațiile exprimă faptul că unul și numai unul din aceste predicate poate fi satisfăcut (adevărat).



Pentru cazul  $n=2$  se poate scrie, echivalent:



### Definiție:

Se numește schemă logică un graf orientat în care:

- Există un singur bloc START și un singur bloc STOP;
- Orice arc este etichetat cu una din următoarele informații: START sau STOP; o citire sau o scriere; o atribuire; un predikat, în care caz extremitatea inițială a arcului este extremitatea inițială a unui bloc de ramificație;

c) Orice arc face parte din cel puțin un drum care începe cu blocul START și se termină cu blocul STOP.

În practică, schemele logice alcătuite conform regulilor enunțate anterior pot lua forme foarte complicate, greu de urmărit, uneori chiar de cel care le-a conceput. De aceea s-a simțit nevoia ca în construcția schemelor logice să se folosească numai anumite configurații (structuri) și să se respecte reguli stricte, obținându-se, astfel, scheme logice structurate.

### Definiție:

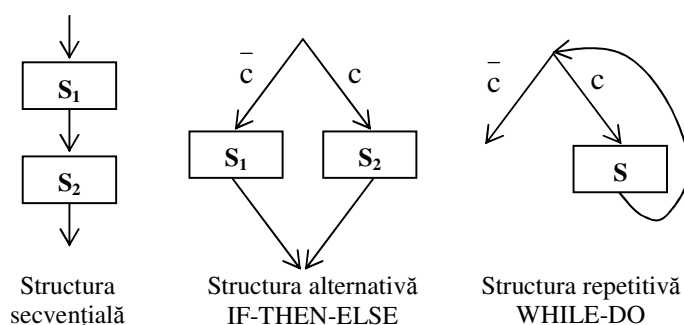
Se numește subschemă logică un graf orientat în care:

- a) Există un unic vârf inițial (în care nu sosesc arce) și un vârf final (din care nu pleacă arce);
- b) Oricare arc este etichetat cu una din următoarele informații: START sau STOP; o citire sau o scriere; o atribuire; un predicat, în care caz extremitatea inițială a arcului este extremitatea inițială a unui bloc de ramificație;
- c) Dacă subschema conține blocul START (STOP), atunci extremitatea inițială (finală) a blocului este chiar vârful inițial (final);
- d) Orice arc face parte din cel puțin un drum ce unește vârful inițial cu cel final.

În particular, o schemă logică este o subschemă logică în care vârful inițial este extremitatea inițială a blocului START, iar vârful final este extremitatea finală a blocului STOP.

Prin recurență, schema logică structurată (**s.l.s.**) se definește astfel:

- (I) Blocurile START, STOP, de intrare/ieșire și de atribuire sunt **s.l.s.**;
- (II) Dacă **s1** și **s2** sunt **s.l.s.**, atunci și subschemele din figura 1.1. sunt **s.l.s.** (cu respectarea condițiilor referitoare la START și STOP).
- (III) Orice **s.l.s.** se obține plecând de la (I) și aplicând de un număr finit de ori regulile (II).



**Fig. 1.1.** Subschemă logică structurate

O schemă logică structurată este o **s.l.s.** care este chiar o schemă logică.

Se pune întrebarea: se poate genera orice schemă logică folosind numai formele impuse schemelor logice structurate? Sau altfel spus, se poate transforma orice schemă logică într-o schemă logică bine structurată? Răspunsul, afirmativ, este dat de teorema fundamentală a programării structurate (teorema lui Böhm-Jacopini): *orice schemă logică este echivalentă cu o schemă logică structurată*. [Dodescu et al., 1987]

### ➤ Reprezentarea algoritmilor prin pseudocod

Descrierea algoritmilor cu ajutorul schemelor logice se dovedește, în unele cazuri, greoaie, practica impunând în ultima vreme descrierea lor printr-un text coerent, construit pe baza unor reguli. O astfel de descriere se numește *metalimbaj (pseudocod)*. Există multe variante de metalimbaje. Oricine poate crea un pseudocod propriu, perfect acceptabil, cu condiția ca el să conțină structurile de bază, suficiente pentru a descrie orice algoritm. Evident că și la pseudocoduri



se pune problema „portabilității”, mai ales atunci când se lucrează în echipă. Cel mai avansat pseudocod „universal” a fost propus în anii '60 de Niklaus Wirth, pseudocod care ulterior a devenit limbajul de programare Pascal. [Roșca et al., 1998] Se propune următorul pseudocod:

● **Cuvinte cheie.** Sunt mnemonice ale instrucțiunilor, unele fiind scrise în limba engleză, pentru compatibilizare cu literatura de specialitate (de exemplu WHILE-DO, IF-THEN-ELSE).

● **Instrucțiuni.** Instrucțiunile pot fi scrise liber, în sensul că o instrucțiune se poate scrie pe mai multe rânduri, iar un rând poate conține mai multe instrucțiuni (separate prin ";"). Instrucțiunile se împart în *declarații* (instrucțiuni neexecutabile) și *instrucțiuni efective* (executabile).

O declarație este formată din cuvinte cheie (de exemplu ÎNTREG, REAL, CHARACTER etc.), urmat de un șir de variabile separate prin ",", variabile pentru care se indică faptul că au tipul întreg, real, caracter etc. Masivele se declară prin tipul elementelor sale, urmat de o construcție care desemnează numărul maxim de elemente pe fiecare dimensiune, sub forma  $[d_1][d_2]...[d_n]$ .



#### Exemple:

```
ÎNTREG a, b, c;
REAL vector[20];
ÎNTREG matrice [10][15];
```

Instrucțiunile executabile într-un program scris în pseudocod pot fi:

✗ Instrucțiunea de citire are forma: **CITEȘTE(listă\_de\_variabile)**.



#### Exemple:

```
CITEȘTE(A,B);
CITEȘTE(xi);
```

✗ **Instrucțiunea de scriere** are forma: **SCRIE(listă\_de\_variabile)**.



#### Exemple:

```
SCRIE(A,B);
SCRIE("VECTORUL ESTE =");
SCRIE(xi);
```

✗ Instrucțiunea de atribuire are forma: **v=e**, unde **v** este variabilă, **e** este expresie, ambele de aceeași natură (numerică, logică sau caracter).

✗ Instrucțiunile de ramificare (**IF-THEN-ELSE**, **IF-THEN**, **CASE-OF**) și cele repetitive (**WHILE-DO**, **DO-UNTIL**, **DO-FOR**) sunt prezentate împreună cu structurile fundamentale.

#### Observații:

a) Instrucțiunile corespund configurațiilor acceptate într-o schemă logică structurată, deci sunt compatibile cu programarea structurată.

b) Instrucțiunile acestui pseudocod se pot extinde la lucrul cu fișiere.



#### Teste de autoevaluare

3. Blocurile dintr-o subschemă logică sunt etichetate cu una din informațiile: 1)START; 2)citire; 3)scriere; 4)expresie aritmetică; 5)expresie logică; 6)expresie relațională; 7)sir de caractere; 8)atribuire; 9)salt necondiționat; 10)STOP. a)oricare; b)1,2,3,5,6,8 sau 10; c)1,2,3,4,8 sau 10; d)1,2,3,5,6,7,8 sau 10; e)1,2,3,4,6,8,9 sau 10
4. Reprezentarea prin arbori este permisă numai pentru structurile: 1)BLOCK; 2)IF-THEN-ELSE; 3)CASE-OF; 4)WHILE-DO; 5)DO-UNTIL; 6)DO-FOR. a) toate; b)1,2,3,4 și 5; c)2,3,4,5 și 6; d)1,2 și 4; e)1,2 și 5.

## 1.2. Descrierea structurilor fundamentale

În prezentarea structurilor fundamentale se vor folosi, în paralel, următoarele reprezentări: schemă logică, arbore, pseudocod și exprimare analitică. În cele ce urmează,  $s_1, s_2, \dots, s_n$  sunt **s.l.s.** (respectiv șiruri de instrucțiuni în pseudocod sau module structurate în schemele arbore).

● **Structura secvențială** (liniară) este prezentată în figura 1.2. și se notează analitic cu:  $\text{BLOCK}[2](S_1, S_2)$  sau  $\text{BLOCK}(S_1, S_2)$ . Structura secvențială poate conține mai multe blocuri:

$\text{BLOCK}_{[n]}(S_1, S_2 \dots S_n) = \text{BLOCK}(\text{BLOCK}_{[n-1]}(S_1, S_2 \dots S_{n-1}), S_n)$ , cu  $n \geq 2$ .

Poate fi definită o structură secvențială cu un singur bloc:  $\text{BLOCK}_{[1]}(S_1) = S_1$ .

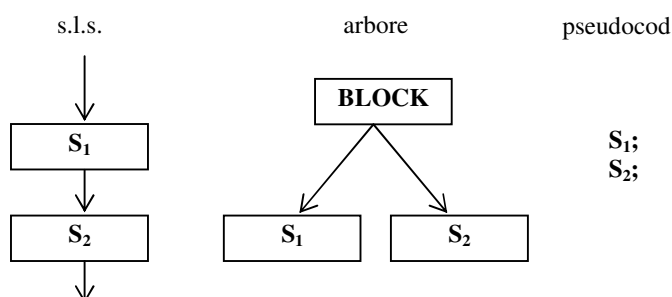
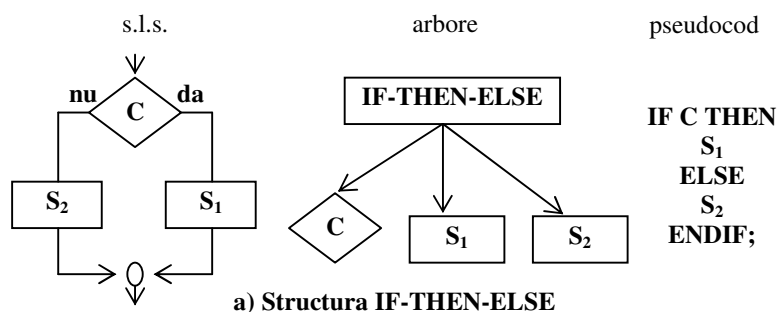
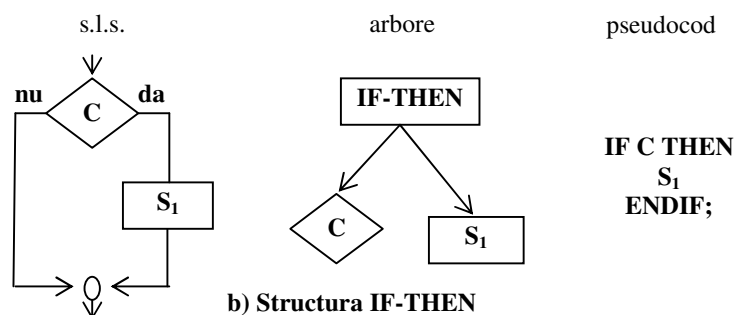


Fig. 1.2. Structura secvențială

● **Structurile alternative** sunt de mai multe tipuri:



a) Structura IF-THEN-ELSE



b) Structura IF-THEN

Fig. 1.3. Structuri alternative

✖ Structura IF-THEN-ELSE (selecția simplă) este prezentată în figura 1.3.a. și se notează analitic cu  $\text{IF-THEN-ELSE}(c, S_1, S_2)$ .

✖ Structura IF-THEN (pseudoalternativă) este prezentată în figura 1.3.b și se notează analitic cu  $\text{IF-THEN}(c, s)$ . Structura IF-THEN este o formă particulară a structurii IF-THEN-ELSE, putându-se exprima astfel:

$IF-THEN(c,s)=IF-THEN-ELSE(c,s,\Phi)$ , unde  $\Phi$  este o s.l.s. vidă.

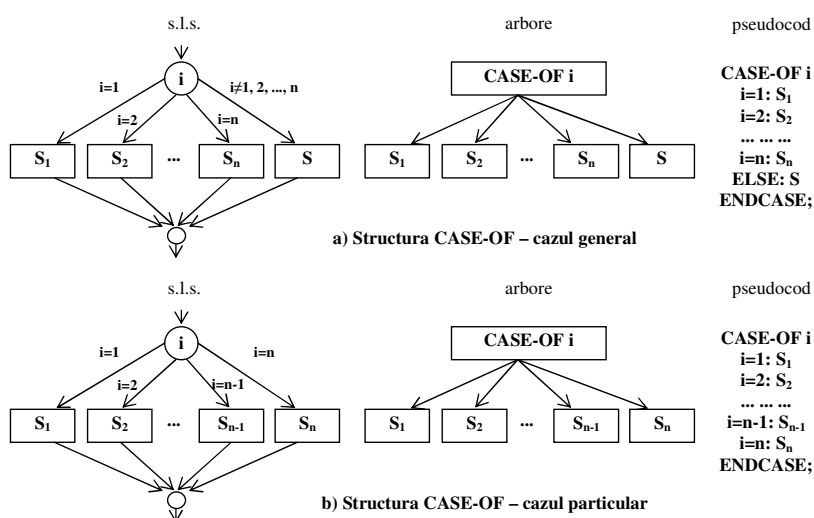
Programarea structurată acceptă și structura IF-ELSE (cu s pe ramura fals), notată  $IF-ELSE(c,s)$ , tot ca formă particulară a structurii IF-THEN-ELSE:

$$IF-ELSE(c,s)=IF-THEN-ELSE(c, \Phi, s)$$

Structurile IF-THEN și IF-ELSE se pot exprima (pot fi înlocuite în algoritm) una prin cealaltă:

$$IF-THEN(c,s) = IF-ELSE(\bar{c}, s) \text{ și } IF-ELSE(c,s) = IF-THEN(\bar{c}, s)$$

✖ Structura CASE-OF (selecția multiplă) este prezentată în figura 1.4 și se notează analitic cu  $CASE-OF(i,s_1,s_2,\dots,s_n,s)$ , în cazul general și  $CASE-OF(i,s_1,s_2,\dots,s_n)$ , în cazul particular, unde  $i$  este o variabilă selector care poate lua valori în mulțimea  $V=\{v_1, v_2, \dots, v_n\}$ .  $V$  este o mulțime discretă, finită și ordonată. Structura CASE-OF este echivalentă cu structura IF-THEN-ELSE (demonstrați acest lucru analitic, prin s.l.s. și prin structuri arborescente).



**Fig. 1.4.** Structura CASE-OF

● **Structurile repetitive** sunt de mai multe tipuri:

✖ Structura repetitivă condiționată anterior este prezentată în figura 1.5.a. și se notează analitic cu  $WHILE-DO(c,s)$ .

✖ Structura repetitivă condiționată posterior, este prezentată în figura 1.5.b. și se notează analitic cu  $DO-UNTIL(c,s)$ .

✖ Structura repetitivă cu numărător este prezentată în figura 1.5.c. și se notează analitic cu  $DO-FOR(v,v_i,v_f,v_r,s)$ , unde  $v$  este o variabilă contor (numărător), iar  $v_i, v_f, v_r$  sunt expresii cu rol de valoare inițială, valoare finală, respectiv valoare rație. Atunci când  $v_r=1$ , rația se poate omite în pseudocod.

### Observații:

1. Diferența esențială între  $WHILE-DO$  și  $DO-UNTIL$  constă în aceea că  $DO-UNTIL$  execută  $s$  cel puțin o dată, pe când  $WHILE-DO$  poate să nu execute pe  $s$ . Oricum, cele două structuri se pot transforma ușor una în alta, astfel:

$$WHILE-DO(c,s)=IF-THEN(c,DO-UNTIL(\bar{c},s))$$

$$DO-UNTIL(s,c)=BLOCK(s,WHILE-DO(\bar{c},s))$$

2. Structura  $DO-FOR$  este un caz particular al structurii  $DO-UNTIL$ , putându-se scrie:

$$DO-FOR(v,v_i,v_f,v_r,s)=BLOCK(v=v_i,WHILE-DO(v \leq v_f,BLOCK(s,v=v+v_r)))$$

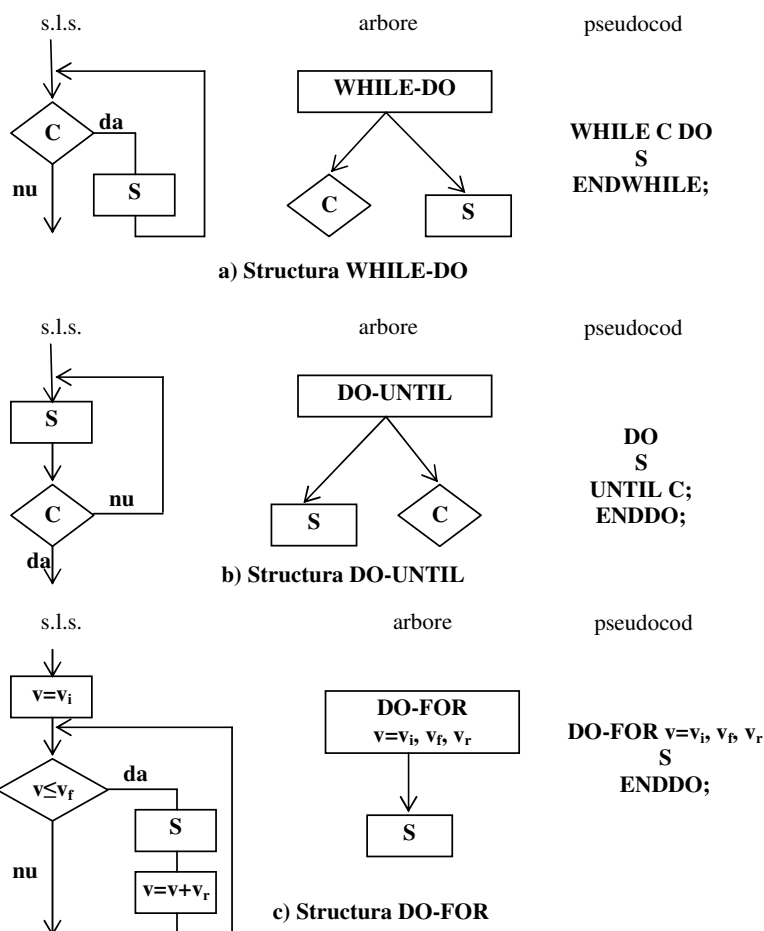


Fig. 1.5. Structurile repetitive



## Teste de autoevaluare

5. Structura  $\text{DO-FOR}(v, v_i, v_f, v_r, s)$  este echivalentă cu: a)  $\text{BLOCK}(v=v_i, \text{DO-UNTIL}(\text{BLOCK}(v=v+v_r, s), v > v_f))$ ; b)  $\text{BLOCK}(v=v_f, \text{DO-UNTIL}(\text{BLOCK}(s, v=v-v_r), v \leq v_i))$ ; c)  $\text{BLOCK}(v=v_i, \text{IF-THEN}(v \leq v_f, \text{DO-UNTIL}(\text{BLOCK}(s, v=v+v_r), v > v_f)))$ ; d)  $\text{BLOCK}(v=v_f, \text{WHILE-DO}(v > v_i, \text{BLOCK}(s, v=v-v_r)))$ ; e)  $\text{BLOCK}(v=v_i, \text{WHILE-DO}(v < v_f, \text{BLOCK}(s, v=v+v_r)))$ ;
6. Structura  $\text{WHILE-DO}(c, s)$  este echivalentă cu: a)  $\text{DO-UNTIL}(s, \bar{c})$ ; b)  $\text{BLOCK}(s, \text{DO-UNTIL}(s, \bar{c}))$ ; c)  $\text{IF-THEN}(c, \text{DO-UNTIL}(s, c))$ ; d)  $\text{BLOCK}(s, \text{IF-THEN}(c, s))$ ; e)  $\text{DO-UNTIL}(\text{IF-THEN}(c, s), \bar{c})$

### 1.3. Structurarea și proiectarea algoritmilor

Un algoritm se consideră structurat dacă și numai dacă conține structurile fundamentale prezentate anterior. Deoarece fiecare structură are o singură intrare și o singură ieșire, schema logică are o singură intrare și o singură ieșire.

Considerându-se o familie  $D'$  de structuri fundamentale (de exemplu  $D' = \{\text{BLOCK}, \text{IF-THEN-ELSE}, \text{IF-THEN}, \text{DO-UNTIL}, \text{WHILE-DO}\}$ ), un algoritm se numește  $D'$ -structurat dacă este reprezentat numai prin structuri din  $D'$ . De remarcat faptul că orice algoritm structurat poate fi realizat numai prin structurile de bază din familia  $D = \{\text{BLOCK}, \text{IF-THEN-ELSE}, \text{WHILE-DO}\}$ . Un algoritm  $P$ , structurat, este echivalent cu un algoritm pus sub una din următoarele forme:

$$P = \text{BLOCK}(s_1, s_2)$$

$$P = \text{IF-THEN-ELSE}(c, s_1, s_2)$$

$$P = \text{WHILE-DO}(c, s)$$

Dacă schema logică prin care se ilustrează algoritmul de rezolvare a unei probleme nu conține numai structurile fundamentale, atunci ea este nestructurată.

Orice schemă logică nestructurată se poate structura, conform următoarei teoreme de structură: Fie  $S$  o schemă logică nestructurată în care etichetele care sunt predicate formează o mulțime  $P$ , iar etichetele care nu sunt predicate formează mulțimea  $A$ , a acțiunilor. Se pot adăuga lui  $A$  și  $P$  alte acțiuni, respectiv alte predicate, diferite de cele din  $S$ , astfel încât să se obțină o schemă logică structurată echivalentă cu  $S$ . Dacă o schemă logică nestructurată este prea complicată, se renunță la structurare (care, prin adăugare de noi acțiuni și predicate, complică și mai mult schema) și se reproiectează.

În practică s-au impus următoarele metode de structurare.

*Metoda dublării codurilor* se folosește la structurarea alternativelor sau repetitivelor. Ea constă în dublarea, ori de câte ori este nevoie, a unui cod (a unei acțiuni sau a unui predicat), astfel încât să se obțină numai structuri fundamentale.

*Metoda introducerii unei variabile booleene* se folosește pentru structurarea repetitivelor și constă în:

- se copiază din vechea schemă logică toate blocurile până la intrarea în structura repetitivă care nu este fundamentală;
- se introduce variabila booleană, de exemplu  $VB$  ( $VB=0$  sau  $VB=1$ );
- se alege structura repetitivă fundamentală  $\text{WHILE-DO}$  sau  $\text{DO-UNTIL}$ , în care condiția se referă la variabila booleană introdusă,  $VB$ ;
- în interiorul structurii repetitive alese, variabila booleană își schimbă valoarea:  $VB=1$  (sau  $VB=0$ ) pe toate drumurile care duceau la ieșirea din vechea structură repetitivă.

#### Erorile în algoritmi

Un algoritm este eficient și devine operațional în măsura în care între resursele de calcul utilizate (timp și memorie calculator) și precizia rezultatelor se stabilește un raport acceptabil.

Cu toată precizia oferită de calculatoarele electronice, calitatea rezultatelor este influențată de mulți alți factori. Soluția unei probleme depinde de datele inițiale, acestea fiind obținute în urma unor observații, măsurători sau pe baza altor calcule prealabile. Precizia instrumentelor cu care se fac observațiile, condițiile în care au loc acestea, precizia calculelor necesare determinării unor parametri inițiali generează *erori în datele inițiale*. În general, pentru această clasă de erori se stabilesc limite de eroare.

O parte din parametrii utilizați în formulele de calcul nu au o valoare exprimabilă printr-un număr finit de zecimale (de exemplu  $\sqrt{3}$ ,  $\pi$ ,  $e$  etc). *Erorile de aproximare* a lor sunt cunoscute și

vor fi astfel alese încât să fie corelate cu precizia dorită pentru calculele în care intră acești parametri.

O altă clasă importantă de erori o constituie *erorile de rotunjire*. Ele apar ca urmare a limitării numărului de zecimale cu care se poate reprezenta un număr în calculator. Aceste erori depind de modul de reprezentare a numerelor în calculator, de sistemul de numerație folosit în calcule, precum și de conversiile dintr-un sistem de numerație în altul.

La rezolvarea, mai ales numerică, a unei probleme se folosește o metodă matematică. De multe ori, fenomenul modelat este supus unor condiții simplificatoare, în funcție de acestea alegând una sau alta din metodele de rezolvare existente. Alegerea metodei poate introduce așa numitele *erori de metodă*. Erorile introduse prin prezența în calculator a unor funcții cărora în analiza matematică le corespund serii infinite se numesc *erori reziduale*. Ele se explică prin imposibilitatea calculării unei serii infinite într-un număr finit de pași ai algoritmului.

Dacă  $x$  este valoarea exactă și  $x^*$  o valoare aproximativă a lui  $x$ , obținută ca urmare a prezenței unor erori din clasele menționate anterior, se disting următoarele situații:

$x^* > x$             ►  $x^*$  este o aproximare a lui  $x$  prin adaos;

$x^* < x$             ►  $x^*$  realizează o aproximare prin lipsă.

Diferența  $\varepsilon_{x^*} = x - x^*$  reprezintă eroarea, iar când nu interesează sensul ei se calculează  $|\varepsilon_{x^*}| = |x - x^*|$ , care poartă numele de *eroare absolută*.

Erorile pot fi acceptate sau respinse, nu numai în funcție de mărimea lor, ci și în funcție de mărimea valorilor cărora li se asociază. În acest scop se calculează raportul  $r_{x^*} = \frac{|x - x^*|}{x^*}$  care desemnează *eroarea relativă*.

În cazul operațiilor de adunare și scădere, eroarea absolută nu depășește suma erorilor absolute a celor două numere care se adună algebric.

$$\varepsilon_{x^* \pm y^*} = (x \pm y) - (x^* \pm y^*) = \varepsilon_{x^*} \pm \varepsilon_{y^*}, \text{ pentru eroarea absolută;}$$

$$r_{x^* \pm y^*} = \frac{\varepsilon_{x^*} \pm \varepsilon_{y^*}}{x^* \pm y^*} = r_{x^*} \frac{x^*}{x^* \pm y^*} \pm r_{y^*} \frac{y^*}{x^* \pm y^*}, \text{ pentru eroarea relativă.}$$

De reținut că la scăderea a două numere apropiate ca valoare, eroarea relativă își pierde din acuratețe datorită numitorului raportului care tinde spre zero. Relațiile funcționează și pentru  $n > 2$  numere. În cazul produsului a mai multor numere aproximative și nenule, eroarea relativă nu depășește suma erorilor relative ale numerelor:

$$\varepsilon_{x^* \cdot y^*} - (xy - x^* y^*) = (x^* + \varepsilon_{x^*}) \cdot (y^* + \varepsilon_{y^*}) - x^* y^* = x^* \varepsilon_{y^*} + y^* \varepsilon_{x^*} + \varepsilon_{x^*} \varepsilon_{y^*}$$

$$r_{x^* \cdot y^*} = \frac{\varepsilon_{x^* \cdot y^*}}{x^* \cdot y^*}$$

în care, logaritmând (valorile absolute permițând acest lucru, iar  $x^*$  și  $y^*$  pentru simplificare au același semn) și folosind formula de aproximare  $\ln x - \ln x^* \approx d \ln x^* = \frac{\varepsilon_{x^*}}{x^*}$ , rezultă că

$$r_{x^* \cdot y^*} \leq r_{x^*} + r_{y^*}. \text{ În cazul operației de împărțire } \varepsilon_{\left(\frac{x}{y}\right)^*} \approx \frac{y^* \varepsilon_{x^*} - x^* \varepsilon_{y^*}}{(y^*)^2}, \quad r_{\left(\frac{x}{y}\right)^*} \leq |r_{x^*}| + |r_{y^*}|, \text{ ceea ce}$$

înseamnă că eroarea relativă a câtului nu excede suma erorilor relative ale deîmpărțitului și împărțitorului. Erorile în cazul unor expresii calculabile prin operațiile elementare pot fi approximate folosind limita maximă a erorilor fiecărui termen în parte.



### Teste de autoevaluare

7. Un algoritm structurat este echivalent cu un algoritm pus sub una din formele:  
 1)BLOCK(s1,s2); 2)IF-THEN-ELSE(c,s1,s2); 3)IF-THEN(c,s); 4)CASE-OF(i,s1,s2,...,sn,s);  
 5)WHILE-DO(c,s); 6)DO-UNTIL(s,c); 7)DO-FOR(v,vi,vf,vr,s). **a) 1,2,3,4,5,6,7; b) 1,2,3,5,6;  
 c) 1,2,5,6,7; d) 1,2,5; e) 1,2,6.**
8. Teorema de structură stabilește că: **a) orice schemă logică este echivalentă cu o schemă logică structurată; b) orice schemă logică poate fi pusă sub una din formele:BLOCK(s1,s2); IF-THEN-ELSE(c,s1,s2); WHILE-DO(c,s); c) corectitudinea unei scheme logice structurate se verifică prin examinarea fiecărui nod din arborescența sa; d) o schemă logică structurată poate fi descompusă în structurile privilegiate**

### Proiectarea algoritmilor

Conceptele principale ce s-au cristalizat în domeniul programării structurate sunt: proiectarea *top-down*, proiectarea *modulară*, proiectarea *structurată*. Cele trei tipuri de proiectări nu se exclud unul pe altul, ci se intercorelează pentru desfășurarea unei activități organizate și disciplinate, concretizată în obținerea unor produse program care să reflecte clar ierarhizarea prelucrărilor și care să faciliteze testarea și documentarea lor.

● *Proiectarea top-down* presupune descompunerea, de la general la particular, a problemei date în subprobleme sau funcții de prelucrat, conducând la realizarea algoritmului în mai multe faze succesive, fiecare fază fiind o detaliere a fazei anterioare până când algoritmul este suficient de rafinat (detaliat) pentru a fi codificat. Apar astfel, în diferite faze succesive, algoritmi din ce în ce mai detaliați. În urma descompunerii se obține o structură liniară sau arborescentă.

Proiectarea top-down este însoțită de codificare (scriere a programelor) top-down și testare top-down. Codificarea top-down presupune, în principal, posibilitatea scrierii unui modul înainte de a se proiecta modulele de nivel inferior (superior), iar testarea top-down constă în realizarea ei de sus în jos: se pornește cu modulul rădăcină și cu unu sau mai multe module de ordin imediat inferior, se continuă cu atașarea unui alt nivel inferior etc., până când s-au inclus în testare modulele ultimului nivel. Testarea top-down poate lua în considerare, la un moment dat, numai legăturile unui modul cu module de pe nivelul inferior, fără testare propriu-zisă a acestora din urmă.

● *Proiectarea modularizată* presupune descompunerea problemelor în părți numite **module**, astfel încât fiecare din acestea să îndeplinească anumite funcții bine definite. Descompunerea se poate face în mai multe faze (la mai multe niveluri) prin metoda top-down. Criteriile de descompunere în module depind, în mare măsură, de experiența proiectanților (programatorilor).

Ele se referă, în principal, la: omogenizarea funcțiilor; utilizarea diverselor structuri de date; separarea funcțiilor de intrare/ieșire de funcțiile de prelucrare; utilizarea unor module deja existente; posibilitățile echipei în sarcina căreia intră realizarea modulelor; utilizarea eficientă a resurselor calculatorului (periferice, timp, memorie internă) etc.

Proiectarea modularizată presupune, pe lângă identificarea modulelor și a relațiilor dintre ele, și precizarea modului și ordinii în care modulele sunt puse în lucru. Din punct de vedere al funcțiilor pe care le conțin, se disting: *module de prelucrare* (operaționale) care conțin funcții de prelucrare (operații propriu-zise), *module de comandă* (monitor) care apelează (activează) alte module și *module mixte*, care conțin atât funcții de prelucrare cât și de comandă. În stabilirea ordinii de punere în lucru a modulelor, arborele se parcurge în preordine. După stabilirea modulelor, acestea se abordează algoritmic, independent unul față de altul.

● *Proiectarea structurată* a algoritmilor constă dintr-o mulțime de restricții și reguli de elaborare care forțează proiectantul (programatorul) să urmeze o formă strânsă de reprezentare și codificare. Într-un sens mai larg, programarea structurată - incluzând aici și elaborarea algoritmilor - este modalitatea de ordonare a activității mentale desfășurată în scopul obținerii de programe (algoritmi) constituite din structuri fundamentale cu un grad de structurare cât mai mare și în condițiile minimizării efortului de programare, dar obținerii unui produs de cea mai bună calitate.

### **Verificarea și analiza corectitudinii algoritmilor**

În procesul de elaborare a algoritmilor se pot strecura formulări imprecise sau eronate, ceea ce determină obținerea unor rezultate incomplete sau eronate. Verificarea corectitudinii ar însemna verificarea faptului că pentru orice set de date, algoritmul elaborat furnizează rezultate corecte, lucru imposibil de realizat în practică. Există încercări, cu valoare teoretică, de a elabora o metodologie de verificare a corectitudinii algoritmului. În practică se recomandă următoarele verificări ale corectitudinii algoritmilor simpli:

1. încheierea algoritmului după un număr finit de pași (în principal, modul de construire a ciclărilor);
2. modul în care au fost construite selecțiile, astfel încât variantele să fie corect definite în algoritm;
3. asigurarea valorilor (prin introducerea din exterior sau inițializare în algoritm) pentru toate variabilele referite (utilizate) în operații.

Dacă cele trei tipuri de verificări au condus la concluzia de corectitudine, se procedează la un test de birou care presupune parcurgerea atentă, operație cu operație, a algoritmului, pentru seturi de date, de obicei cazuri limită.

Analiza algoritmilor (studiul eficienței lor) constă, în principal, în:

I) Determinarea *necesarului de memorie*;

II) Determinarea *timpului necesar execuției algoritmului*. Deoarece, pentru seturi diferite de date, timpii de execuție vor fi diferiți, timpul necesar execuției algoritmului poate însemna timpul în cazul cel mai defavorabil sau timpul mediu, rezultat ca raport între suma timpului necesar pentru toate seturile de date considerate și numărul acestor seturi.

III) Determinarea *optimalității algoritmului*, care este, în general, o problemă dificilă, în care important este criteriul după care se judecă algoritmul: (I) sau (II). Cu toate progresele tehnologice din ultimul timp (calculatoare rapide, cu memorii mari), necesarul de memorie și timpul UC rămân resurse importante care trebuie bine utilizate.



### **Test de autoevaluare**

9. Care tipurile de proiectări cristalizate în domeniul programării structurate?



## Răspunsuri și comentarii la testele de autoevaluare

**1:d); 2:d); 3:b); 4:a); 5:c); 6:e); 7:d); 8:a); 9.** Tipurile de proiectări care s-au cristalizat în domeniul programării structurate sunt: proiectarea *top-down*, proiectarea *modulară*, proiectarea *structurată*.

### Rezumat

În cadrul acestei unități de învățare au fost studiate următoarele aspecte în ceea ce privește algoritmi și schemele logice:

- ➡ cunoștințe teoretice privind caracteristicile și descrierea algoritmilor;
- ➡ modalități de reprezentare a algoritmilor sub formă de scheme logice și pseudocod;
- ➡ reprezentarea structurilor fundamentale sub formă de schemă logică, arbore, pseudocod și exprimare analitică;
- ➡ proiectarea, verificarea și analiza algoritmilor.

După încheierea acestei unități de învățare, studenții au cunoștințe și abilități de rezolvare a problemelor (în special lucrul cu masive), prin reprezentarea acestora sub formă de algoritmi structurați.

### Bibliografia unității de învățare

1. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, M. Mircea, L. Bătăgan, C. Silvestru, Bazele programării calculatoarelor. Teorie și aplicații în C, Ed. ASE, București, 2006, ISBN 973-594-591-6
2. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, Programarea calculatoarelor. Știința învățării unui limbaj de programare, Teorie și aplicații, Ed. ASE, 2003
3. Roșca I. Gh., Apostol C., Ghilic-Micu B., Roșca V., *Programare sistematică în Pascal*, Ed. Didactică și Pedagogică, București 1998, ISBN 973-30-3341-3

## 2. Organizarea internă a datelor

### Cuprins

- Obiectivele unității de învățare 2
- 2.1. Informația, data și reprezentarea internă a datelor
- 2.2. Structuri de date
- Răspunsuri și comentarii la testele de autoevaluare
- Bibliografia unității de învățare

### Obiectivele unității de învățare 2

Dupa studiul acestei unitati de învățare, studenții vor avea cunoștințe teoretice și abilități practice despre:

- ➡ informații, date, cunoștințe;
- ➡ reprezentarea internă a datelor;
- ➡ structuri statice de date;
- ➡ structuri dinamice de date.



*Durata medie a unității de studiu individual - 8 ore*

### 2.1. Informația, data și reprezentarea internă a datelor

Informația este un concept de maximă generalitate, care poate fi definit prin raportarea la conceptele de materie și energie. Fondatorul ciberneticii, Norbert Wiener<sup>1</sup>, consideră informația ca a treia formă de manifestare a realității obiective, apreciind că ea nu este nici materie, nici energie, ci pur și simplu informație. Se poate remarca faptul că definiția anterioară este dată în sens negativ.

Pentru a încerca o definiție afirmativă, se cuvine mai întâi să se facă distincție între două maniere de a aborda conceptul pus în discuție: în general, ca semnele care circulă pe diferite canale între elementele lumii reale, cu forme specifice de receptare la diferitele niveluri de manifestare a materiei vii; în particular, când elementul receptor este omul, ca ființă superioară pe scara biologică. În acest ultim sens, informația trebuie considerată în raport cu *procesul de cunoaștere* și cu modul de reflectare a rezultatelor sale în conștiința ființei umane. În literatura de specialitate sunt identificate următoarele trăsături definitorii ale informației:

④ *semn cu semnificație*: este obligatoriu ca un mesaj să fie construit într-o limbă (limbaj) cunoscută de receptor;

④ *noutate*: există mesaje care, deși sunt redactate într-o limbă cunoscută, nu conțin nimic nou pentru receptor, pierzându-se calitatea de informație;

---

<sup>1</sup> Norbert Wiener – *Cybernetics or Control and Communication in the Animal and the Machine*, Herman and Cie, Paris, The MIT Press, Cambridge (Mass), Wiley and Sons, New York (1948), ediția a doua revăzută și adăugită (două capitole noi), The MIT Press, Cambridge (Mass), Wiley and Sons, New York, 1961

⊕ *utilitate*: în raport cu interesele receptorului, este posibil ca un mesaj cu caracter de noutate să fie inutil pentru activitatea sa, pierzându-și, de asemenea, calitatea de informație.

Corespunzător, se identifică următoarele niveluri la care se consideră informația: sintactic, semantic și pragmatic.

*Nivelul sintactic* este asociat sistemului de semne și regulilor utilizate pentru a le reuni în construcții sintactice folosite pentru reprezentarea informației în procesul de culegere, transmitere, înregistrare și prelucrare. Acestui nivel îi corespunde conceptul de *dată*, care, folosind notația formală **BNF** (Backus-Nour Form), poate fi definită astfel:

**<dată> ::= <identificator> <atribute> <valoare>.**

Regulile de sintaxă care formează mulțimea producțiilor pot fi specificate în mai multe moduri, dintre care se remarcă notația formală **BNF** (Backus Normal Form) și diagramele de sintaxă. În notația **BNF**, regulile sintactice (metadefinițiile) au forma:

**<parte-stânga> ::= parte-dreapta**

unde **<parte-stânga>** desemnează metavariabila (variabila neterminală) care se definește, **::=** este un metasimbol având sensul de „este prin definiție”, iar **parte-dreapta** reprezintă definiția metavariabilei.

În cadrul definiției (**parte-dreapta**) se întâlnesc următoarele elemente:

- ⊕ **<metavariabila>**, categorie folosită în definirea altei categorii sintactice;
- ⊕ **metaconstanta**, element al alfabetului terminal;
- ⊕ **|** care separă alternativele în definiție;
- ⊕ **[ ]** care indică o construcție opțională;
- ⊕ **{ }** care indică posibilitatea repetării construcției.

Alternativele se constituie prin juxtapunerea de metavariabile și/sau metaconstante.

Noțiunea de dată va conține pe cea de valoare, dar presupune, în plus, o formă de reprezentare și manipulare, adică un sistem de reguli de transformare având ca scop să se obțină date noi pornind de la cele existente. Pe lângă distincția între conceptele de *dată* și *valoare* se constată, de asemenea, diferența între *informație* și *dată*, ca între un obiect și modelul său. Se mai spune că data este o reprezentare digitală (discretă) a unei anumite cantități de informație. În concluzie, conceptele de informație și dată pot fi utilizate ca sinonime numai în măsura în care acceptăm să identificăm un obiect cu modelul său.

La *nivelul semantic*, informația poate fi caracterizată ca semnificație a datelor. Sensul informației la acest nivel este dat de corespondența între o dată și obiectul real sau situația reprezentată prin această dată.

*Nivelul pragmatic* este cel mai concret nivel de abordare a informației, fiind singurul care consideră informația în raport cu scopurile receptorului. Pornind de la scopurile receptorului pot fi definite caracteristici ca utilitatea sau importanța informației. Nivelul pragmatic permite reflectarea cea mai fidelă a procesului de cunoaștere, a cărui înțelegere completă impune utilizarea conceptului de *cunoștință*.

Procesul de cunoaștere se realizează în timp, prin acumularea progresivă de informații asupra unui obiect sau a unui sistem. La un moment dat, cunoștințele reprezintă totalitatea informațiilor deținute de un observator asupra unui obiect sau sistem. Acest ansamblu de informații se numește *tezaur de cunoștințe* și se folosește ca termen de referință pentru a evalua rezultatele oricărui proces de informare, ca parte a procesului general de cunoaștere.

La limită, rezultatele unui proces de informare pot conduce la una din următoarele situații:

✗ la o extremă, este posibil ca rezultatul să fie nul, dacă informația aparține deja tezaurului receptorului, adică ea a devenit o cunoștință;

✗ la cealaltă extremă, variația potențială a tezaurului receptorului este maximă dacă intersecția între conținutul tezaurului și acela al unui mesaj este vidă.

Referitor la al doilea caz, se constată, însă, că este obligatorie existența unei intersecții nevide între cele două mulțimi, numită *redundanță*, astfel încât recepția și înțelegerea mesajului să aibă loc.

În final putem sintetiza o serie de concluzii clarificatoare a problematicii puse în discuție.

O *informație* este un mesaj susceptibil de a aduce o cunoștință și poate fi generată numai de sisteme cu număr  $n$  finit de stări ( $n \geq 2$ ). Modelul de reprezentare a informației, „formula scrisă”, reprezintă o dată. Deosebirea dintre informație și dată este echivalentă cu deosebirea dintre obiect și modelul său. Sub aspect semantic, informația poate fi caracterizată ca semnificație a datelor. Calculatoarele actuale prelucrează date pe baza cărora se obțin rezultate (tot date) care, prin interpretare umană, capătă un sens, devenind informații.

C.E. Shannon a introdus noțiunea de *cantitate de informație*, care se poate defini astfel: fie un experiment  $X$ , având un număr finit de evenimente elementare independente  $x_1, x_2, \dots, x_n$ , care se realizează cu probabilitățile  $p_1, p_2, \dots, p_n$ :  $X = \begin{pmatrix} x_1 & x_2 & \dots & x_n \\ p_1 & p_2 & \dots & p_n \end{pmatrix}$ . Dacă  $X$  reprezintă un sistem complet de

evenimente:  $0 \leq p_k \leq 1$ , pentru orice  $k \in \{1, 2, \dots, n\}$  (1) și  $\sum_{k=1}^n p_k = 1$  (2), atunci cantitatea de informație (entropia) se măsoară după relația:

$$H(p_1, p_2, \dots, p_n) = - \sum_{k=1}^n p_k \cdot \log_2 p_k \quad (3)$$

Cantitatea de informație se exprimă, cel mai adesea, în biți. Bitul poate fi definit ca informația furnizată de un sistem cu două stări echiprobabile:

$$X = \begin{pmatrix} x_1 & x_2 \\ 1/2 & 1/2 \end{pmatrix} \quad (4)$$

Aplicând formula (3), se obține:

$$H(p_1, p_2, \dots, p_n) = - \sum_{k=1}^n p_k \cdot \log_2 p_k = -(1/2) \log_2 (1/2) - (1/2) \log_2 (1/2) = 1/2 + 1/2 = 1 \text{ bit}$$

Prin *dată* se desemnează un model de reprezentare a informației accesibil unui anumit procesor (om, unitate centrală, program etc.), model cu care se poate opera pentru a obține noi informații despre fenomenele, procesele și obiectele lumii reale.

În notație BNF, semnificația cuvintelor de definire este următoarea.

*Identificatorul* este un nume care se asociază datei, pentru a o distinge de alte date și a o putea referi în procesele de prelucrare (referire prin nume).

*Valoarea datei* se poate preciza prin enumerare sau printr-o proprietate comună. Ea poate fi număr întreg, real, complex, logic, șir de caractere etc. Data care păstrează aceeași valoare pe tot parcursul prelucrării este denumită *constantă*. În caz contrar, data se numește *variabilă*. Pentru datele constante se folosește, în general, drept identificator chiar valoarea, adică o astfel de dată se autoidentifică prin forma textuală a valorii, ceea ce justifică denumirea de *literal*. În unele limbaje de programare, între care și C, există și posibilitatea definirii unor *constante simbolice*, cărora li se asociază un identificator propriu.

*Atributele* precizează proprietăți ale datei și ele determină modul în care aceasta va fi tratată în procesul de prelucrare. Dintre atributele care se pot asocia unei date, cel mai important este tipul. El precizează mulțimea valorilor pe care le poate avea o dată, precum și mulțimea de operații care se pot efectua cu elementele mulțimii de valori ale datei. Pe lângă tip, unei date  $i$  se pot asocia și alte atribute, ca: precizia reprezentării interne, cadrarea valorilor în zona afectată, modul de alocare a memoriei pe parcursul prelucrării (static, dinamic), valoarea inițială etc.

## Reprezentarea internă a datelor

Data este elementară (scalară) dacă apare ca o entitate indivizibilă, atât în raport cu informația pe care o reprezintă, cât și în raport cu procesorul care o prelucerează. O dată scalară poate fi privită la nivelul unui procesor uman (*nivel logic*), respectiv la nivelul calculatorului, ca procesor (*nivel fizic*). La nivel fizic, unei date îi corespunde o zonă de memorie de o anumită mărime, situată la o anumită adresă, în care sunt memorate, în timp și într-o formă specifică, valorile acesteia.

Datele din memoria internă pot fi clasificate în trei mari categorii.

• **Datele alfanumerice** se reprezintă în cod ASCII extins, câte un caracter pe octet. Cele 256 de caractere codificabile ASCII se împart în afișabile și neafișabile. Caracterele afișabile sunt literele (mari și mici) ale alfabetului englez, cifrele, o serie de caractere speciale (+, \*, / etc.), spațiul, o serie de semne grafice ( $\approx$ ,  $\pm$ ,  $\parallel$ ,  $\leq$ ,  $\geq$ ), o serie de caractere grecești ( $\alpha$ ,  $\beta$ ,  $\pi$  etc.) etc. Caracterele neafișabile se utilizează, de regulă, pentru controlul transmisiei și imprimării (Line Feed, Carriage Return, Form Feed, Bell, Tab etc.). Când aceste caractere sunt folosite în alt context decât cel de control, li se pot asocia simboluri grafice nestandardizate, cum ar fi: ☺, ☹, ♂, ♀, ♠, ♣, ♪, ♫.

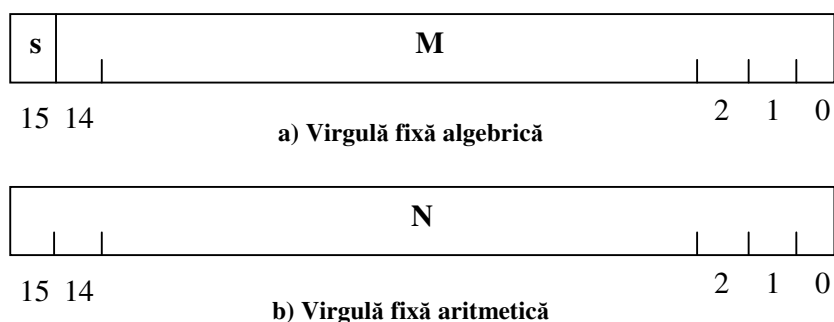
• **Datele numerice** se reprezintă intern în virgulă fixă și virgulă mobilă. UCP are dispozitiv fizic specializat în efectuarea operațiilor în virgulă fixă. Calculul în virgulă mobilă poate fi realizat când microcalculatorul are coprocesor matematic 80x87 sau când limbajele de programare dispun de biblioteci specializate pentru emularea software a operațiilor în această aritmetică.

• **Datele logice** se reprezintă intern în virgulă fixă pe un octet, prin convenție asociindu-se 1 pentru valoarea adevărat și 0 pentru fals.

Se face, de asemenea, mențiunea că unele limbaje de programare lucrează și cu un alt tip de reprezentare internă a datelor numerice, numită reprezentare zecimală.

### Reprezentarea în virgulă fixă

Reprezentarea în virgulă fixă se aplică numai numerelor întregi și se realizează pe zone de memorie standard (un cuvânt sau două cuvinte) sau pe octeți. Ea poate fi algebrică sau aritmetică. Macheta de reprezentare pe un cuvânt este prezentată în figura 2.1, unde:



**Fig. 2.1.** Reprezentarea în virgulă fixă

✖ **s** este bitul de semn (s-a adoptat convenția ca semnul "+" să fie reprezentat prin valoarea binară **0** iar semnul "-" prin valoarea binară **1**);

✖ **M** este o valoare exprimată în binar, în *cod direct* pentru numere pozitive și în *cod*

complementar, pentru numere negative.

✖  $N$  este reprezentat în cod direct. Reprezentarea aritmetică se aplică numai numerelor pozitive, bitul de semn fiind folosit ca bit de cifră.

Codul complementar se obține astfel:

- se reprezintă valoarea absolută a numărului în binar (cod direct). Fie aceasta  $N_2$ ;
- se realizează complementul față de unu (*cod invers*) al lui  $N_2$ . Complementul față de unu se obține prin inversarea fiecărui bit (**0** în **1** și **1** în **0**), inclusiv a bitului de semn. Fie numărul obținut  $N_2^*$ ;
- se adună un unu la numărul reprezentat în cod invers ( $N_2^*$ ) și astfel se obține complementul față de doi al lui  $N_2$ .

### Example:

Numărul -24 se reprezintă astfel:

- ✚ Se exprimă 24 în cod direct:

0	0	...	...	0	0	1	1	0	0	0
15	14			6	5	4	3	2	1	0

- ✚ Se realizează complementul față de unu:

1	1	...	...	1	1	0	0	1	1	1
15	14			6	5	4	3	2	1	0

sau, exprimat în octal: **177747**

- ✚ Se adună unu la numărul reprezentat în complement față de unu:

1	1	...	...	1	1	0	1	0	0	0
15	14			6	5	4	3	2	1	0

sau, exprimat în octal: **177750**.

Pentru determinarea complementului față de doi se poate proceda, astfel: din biții numărului exprimat în cod direct se formează două grupe: grupa din dreapta cuprinde toate zerourile din dreapta ultimului bit cu valoarea **1** precum și ultimul bit **1**; grupa din stânga cuprinde biții rămași; se completează față de **1** grupa din stânga, lăsându-se nemodificată grupa din dreapta.

### Example:

Pentru simplificarea se consideră un cuvânt format din patru biți:

- ✚ Complementul față de 2 pentru 0110:

	01	10	- număr pozitiv (+6)
	stânga	dreapta	
Rezultat:	10	10	- număr negativ (-6)

- ✚ Complementul față de 2 pentru 1101:

	110	1	- număr negativ (-3)
	stânga	dreapta	
Rezultat:	001	1	- număr pozitiv (+3)

Considerându-se o zonă de memorie de  $n$  biți, valoarea care poate fi reprezentată algebric aparține mulțimii  $[-2^{n-1}, 2^{n-1}-1]$  iar valoarea care poate fi reprezentată aritmetic aparține mulțimii  $[0, 2^n-1]$ .

### Reprezentarea în virgulă mobilă

Există mai multe tipuri de reprezentare virgulă mobilă. Dintre acestea, cel mai frecvent utilizat este standardul internațional IEEE (Institute for Electrical and Electronics Engineers). Conform acestui standard, datele se memorează pe 32 de biți (simplă precizie) sau pe 64 de biți (dublă precizie), după machetele prezentate în figura 2.2.

semn	caracteristică	fracție
1 bit	8 (11) biți	23 (52) biți

**Fig. 2.2.** Reprezentarea virgulă mobilă simplă (dublă) precizie

Ambele machete presupun că numărul de reprezentat are următoarea exprimare binară sub formă științifică:  $m = (-1)^s \cdot 1, \text{fracție} \cdot 2^{\text{exponent}}$ , unde  $s$  este valoarea bitului de semn (**1** pentru mantisă negativă și **0** pentru mantisă pozitivă) iar **fracție** este partea fracționară a mantisei.

Mantisa este normalizată și are întotdeauna forma **1,fracție**, ceea ce înseamnă că ea are valori în intervalul [1,2). Pentru ca mantisa să fie adusă la această formă se modifică în mod corespunzător exponentul numărului  $m$ . De exemplu, fie  $m = 101,0111$ . Scrierea lui  $m$  normalizat este  $m = 1,010111 \cdot 2^2$ . Deoarece partea întreagă a mantisei este întotdeauna **1**, aceasta nu se reprezintă intern. Frația se reprezintă intern sub forma semn-mărime (nu în cod complementar).

Pentru a nu se utiliza doi biți de semn (unul pentru mantisă și altul pentru exponent), convenția IEEE a introdus în șablon înlocuirea exponentului cu o *caracteristică*. Aceasta este o valoare în exces față de 127 pentru simplă precizie (exponent+127) sau 1023 pentru dublă precizie (exponent+1023):

#### Exemplu:

Să se reprezinte în simplă precizie numărul  $-75,375$ .

$$-75,375 = -1001011,011_{(2)} = -1,001011011 \cdot 2^6$$

$$s = 1$$

$$\text{mantisă} = 1,001011011_{(2)}$$

$$\text{fracția} = 001011011_{(2)}$$

$$\text{caracteristică} = 6 + 127 = 133 = 10000101_{(2)}$$

$$\text{Reprezentarea internă: } \boxed{1 \quad 10000101 \quad 001011011000000000000000}$$

Caracteristica are următoarele valori normale:

$$0 < \text{caracteristică} < 255 \quad \blacktriangleright \text{ pentru simplă precizie}$$

$$0 < \text{caracteristică} < 2047 \quad \blacktriangleright \text{ pentru dublă precizie.}$$

Când caracteristica are valoarea zero, numărul reprezentat intern ( $m$ ) este zero. Când caracteristica este 255 (respectiv 2047) se consideră depășire virgulă mobilă. Caracteristicile reprezentărilor interne virgulă mobilă simplă și dublă precizie sunt prezentate în tabelul 2.1.

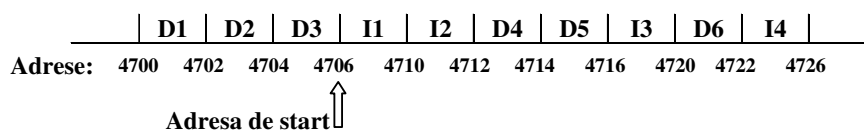
**Tabelul 2.1.** Caracteristicile datelor reprezentate virgulă mobilă

Caracteristici	Tip reprezentare	
	Simplă precizie	Dublă precizie
Număr biți pentru reprezentare caracteristică	8	11
Număr biți pentru	23	52

Caracteristici	Tip reprezentare	
	Simplă precizie	Dublă precizie
reprezentare fracție		
Valoarea minimă caracteristică	1	1
Valoarea maximă caracteristică	254	2047
Eroare maximă fracție	$2^{-24} \approx 10^{-7}$	$2^{-24} \approx 10^{-7}$
Cel mai mic număr pozitiv	$2^{1-127} \approx 10^{-38}$	$2^{1-1023} \approx 10^{-307}$
Cel mai mare număr pozitiv	$2 \cdot 2^{254-127} \approx 10^{38}$	$2 \cdot 2^{2047-1023} \approx 10^{307}$
Domeniu de reprezentare	$[-10^{38}, 10^{38}]$	$[-10^{307}, 10^{307}]$

### Prelucrarea datelor și instrucțiunilor de către unitatea centrală

Pentru înțelegerea modului în care datele și instrucțiunile sunt prelucrate de unitatea centrală, se consideră că memoria conține, din punctul de vedere al programatorului, date (**D**) și instrucțiuni (**I**), în succesiunea de cuvinte ilustrată în figura 2.3.



**Fig. 2.3.** Model de memorie

Așa după cum s-a arătat, instrucțiunile sunt prelucrate de UCC. Încărcarea instrucțiunilor în UCC se realizează automat. Pentru a lansa un program în execuție, UCC trebuie să "cunoască" adresa primei instrucțiuni de executat (adresa de start). Acesta este motivul pentru care în unele limbaje de programare este obligatorie etichetarea primei instrucțiuni executabile. În exemplul din figura 1.9. se presupune că adresa de start a programului este 4706<sub>8</sub>. Această adresă este încărcată în *contorul de adrese* (un registru special numit program counter - PC).

În UCC este încărcată întotdeauna instrucțiunea aflată la adresa precizată de PC. Registrul PC este incrementat automat, fiind astfel pregătit să adreseze următoarea instrucțiune. Deci, după ce se execută instrucțiunea **I**<sub>1</sub>, conținutul lui PC devine 4710<sub>8</sub>. Se încarcă în UCC instrucțiunea **I**<sub>2</sub>. După execuția acesteia, PC va conține adresa 4712<sub>8</sub>. Deși la această adresă este memorată o dată, ea este încărcată în UCC și, prin urmare, este tratată ca instrucțiune. Se pot ivi următoarele situații:

- conținutul lui **D**<sub>4</sub>, printr-o întâmplare, coincide cu o instrucțiune a unității centrale, caz în care este executată ca atare și conținutul lui PC este autoincrementat;
- conținutul lui **D**<sub>4</sub> nu coincide cu nici o instrucțiune acceptată de unitatea centrală, caz în care programul este abandonat (terminare anormală).

Prin acest exemplu s-a arătat cum data este tratată de calculator ca instrucțiune. Deci, prin actualizarea contorului de adrese după execuția fiecărei instrucțiuni, se asigură înlănțuirea normală a instrucțiunilor din program (derularea programului în secvență). Cum se poate proceda



atunci când nu se face parcurgerea în secvență și este nevoie să se sară peste câteva cuvinte de memorie pentru a ajunge la instrucțiunea dorită? Programatorul are la dispoziție instrucțiunile de salt, prin folosirea cărora se asigură întreruperea secvenței normale de program, prin încărcarea forțată în PC a adresei la care se găsește instrucțiunea de executat.

Se presupune că instrucțiunea **I<sub>2</sub>** este: **SALT LA 4716<sub>8</sub>**. După ce se execută instrucțiunea **I<sub>1</sub>**, contorul de locații devine 4710<sub>8</sub>. Se execută, în secvență, instrucțiunea **I<sub>2</sub>** care pune contorul de adrese pe valoarea 4716<sub>8</sub> (fiind o instrucțiune de salt).

Prin urmare, în UCC se încarcă instrucțiunea **I<sub>3</sub>** ș.a.m.d. În exemplul anterior s-a considerat că toate instrucțiunile au lungimea de un cuvânt. În secvența logică a programului, ultima instrucțiune trebuie să fie de tip **STOP**.

După extragerea instrucțiunii din memorie (prima etapă în prelucrarea unei instrucțiuni) se execută următoarele etape: calculul adresei operandului; efectuarea operației propriu-zise, precizată de instrucțiune.

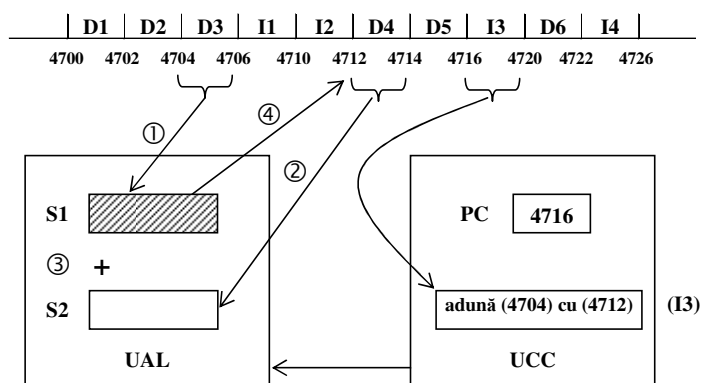
În cadrul fiecărei etape se execută un anumit număr de faze. Indiferent de modul de realizare a fazelor (prin macroinstrucțiuni sau prin funcțiuni cablate), principial, modul de prelucrare a unei instrucțiuni este asemănător. Se consideră că în UCC se găsește instrucțiunea **I<sub>3</sub>: ADUNA (4704) cu (4712)** care adună conținutul de la adresa 4704<sub>8</sub> cu conținutul de la adresa 4712<sub>8</sub> iar rezultatul se depune la adresa celui de-al doilea operand. Pentru a înțelege modul cum se execută această instrucțiune se presupune că UAL este formată din două registre (sumatoare), în care se realizează calcule. Instrucțiunea **I<sub>3</sub>** poate fi descompusă în următorii pași (figura 2.4.):

- preia conținutul operandului sursă de la adresa 4704<sub>8</sub> (**D<sub>3</sub>**) într-un sumator;
- preia conținutul operandului destinație de la adresa 4712<sub>8</sub> (**D<sub>4</sub>**) în al doilea sumator;
- efectuează calculul între cele două sumatoare, cu rezultatul în **S1**;
- depune conținutul sumatorului **S1** în memorie la adresa 4712<sub>8</sub>.

În mod similar se pot interpreta și alte operații de calcul. Toate operațiile realizate asupra datelor se efectuează în unitatea aritmetico-logică. Această afirmație este valabilă și în cazul uneia din cele mai simple operații: **MUTA (4714<sub>8</sub>) IN (4700<sub>8</sub>)**, care se descompune astfel:

- preia operandul sursă de la adresa 4714<sub>8</sub> (**D5**) în sumatorul **S1**;
- depune conținutul sumatorului **S1** în memorie la adresa 4700<sub>8</sub> (**D1** se pierde, depunându-se peste el **D5**).

Ce se va întâmpla în cazul instrucțiunii **ADUNA (4700<sub>8</sub>) cu (4710<sub>8</sub>)**? Răspuns: conținutul de la adresa 4710<sub>8</sub> (**I<sub>2</sub>**) este tratat ca dată.



**Fig. 2.4.** Prelucrarea unei instrucțiuni

Prin acest exemplu s-a văzut cum instrucțiunea este tratată ca dată și, mai mult, cum un program își poate modifica instrucțiunile proprii. Nu toate limbajele de programare oferă posibilitatea intercalării în memorie a datelor și instrucțiunilor. De fapt, această intercalare nici nu

este recomandată, având în vedere gestionarea greoaie a tipurilor de informații: date și instrucțiuni.

Instrucțiunile pot adresa operanzii direct sau indirect. La adresarea indirectă (figura 2.5), instrucțiunea memorează adresa unei zone care stochează adresa operandului.

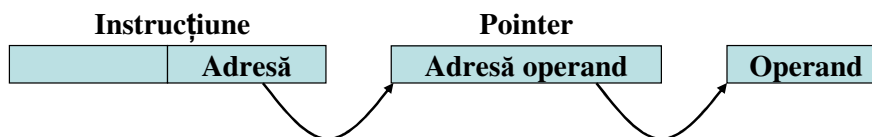


Fig. 2.5. Adresarea indirectă

Se spune că zona de memorie care stochează adresa operandului este de tip pointer. Toate limbajele de programare acceptă adresarea directă a operanzilor. Unele dintre ele acceptă lucrul cu pointeri (adresare indirectă). În acest caz, programatorul trebuie ca, înainte de adresare, să încarce în zona de tip pointer adresa operandului.

### Operații de intrare/ieșire

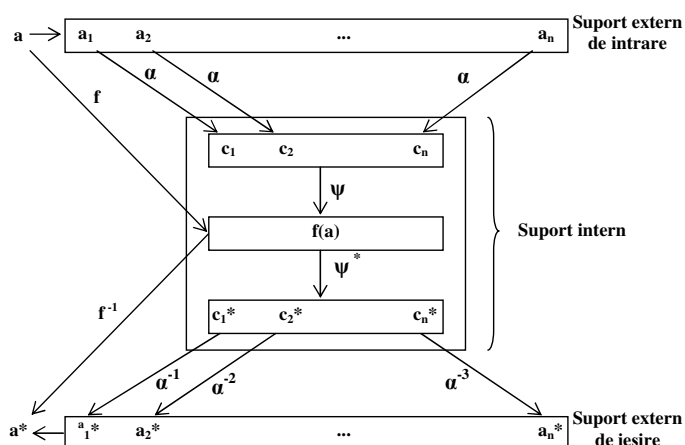
Operațiile de intrare/ieșire realizează *citirea* - introducerea datelor în memorie - și *scrierea* - extragerea datelor din memorie. Deoarece datele pot fi prelucrate doar dacă se găsesc în memoria internă, apare ca necesară operația de citire prin care se transferă datele furnizate de utilizator. Cum toate operațiile își depun rezultatul în memoria internă, punerea lor la dispoziția utilizatorilor necesită operația de scriere. Elementele care definesc o instrucțiune de I/E sunt:

- de unde (unde) sunt introduse/extrase datele;
- care este structura externă a datelor (formatul extern);
- care este adresa memoriei în/din care se introduc/extrag datele.

Reprezentarea numerelor pe suporturile externe la care are acces în mod direct operatorul uman trebuie să fie ASCII. Modul de realizare a corespondenței dintre un număr reprezentat ASCII pe un suport extern și același număr reprezentat intern virgulă fixă sau virgulă mobilă este prezentat schematic în figura 2.6.

Fie  $a \in A \subseteq \mathcal{R}$ . Aplicația  $f$  este o codificare bijectivă a mulțimii  $A$  într-o mulțime de coduri reprezentabile pe un suport intern (de exemplu, o codificare în virgulă fixă sau în virgulă mobilă). Fiecare număr din  $A$  se reprezintă pe un suport extern sub forma unui șir de caractere (caracterele aparțin unei mulțimi  $K$ ). Imaginea pe suportul extern a unui număr  $a \in A$  nu este, în general, unică (de exemplu  $a=13,9$  poate fi reprezentat în mai multe feluri:  $+13.9$ ;  $139$ ;  $+139$  etc.). Se notează cu  $I_a$  mulțimea imaginilor pe suportul extern de intrare pentru elementul  $a \in A$ . În figura 1.11 s-a presupus că imaginea numărului  $a \in A$  pe suportul extern de intrare este șirul de caractere  $a_1, a_2, \dots, a_n$ .

Aplicația  $\alpha$  este o codificare standard a mulțimii  $K$  de caractere (ASCII). Șirul  $c_1, c_2, \dots, c_n$  rezultă din șirul  $a_1, a_2, \dots, a_n$  prin aplicarea funcției  $\alpha$  fiecărui caracter  $a_i$ ,  $i=1, 2, \dots, n$ . Se notează cu  $C_a$  mulțimea șirurilor de coduri care se obține din elementele lui  $I_a$  prin aplicarea funcției  $\alpha$ , în modul descris anterior.



**Fig. 2.6.** Corespondența între reprezentarea externă și internă a datelor

Pentru orice  $a \in A$  se definește funcția  $\psi : \{C_a \mid a \in A\} \rightarrow \{f(a) \mid a \in A\}$  dată de  $\psi(C_a) = f(a)$ . Alegerea funcției  $\psi$  depinde de  $f$  și de mulțimea  $\{I_a \mid a \in A\}$ . Se observă, deci, că funcția  $f$  se realizează în calculator prin intermediul funcțiilor  $\alpha$  și  $\psi$ . Aceste funcții sunt realizate pe baza unor instrucțiuni precizate de programator. Analog se poate analiza funcția de decodificare ( $f^{-1}$ ).



### Teste de autoevaluare

1. O dată reprezentată VF algebrică pe 2o are valoarea maximă: **a)  $2^{16}$ ; b)  $2^{16}-1$ ; c)  $2^{15}-1$ ; d)  $2^{15}$ ; e)  $2^{16}+1$ .**
2. Numărul în zecimal a cărui reprezentare internă în VF algebrică este **10001111** este: **a) 143; b) -15; c) -143; d) -113; e) 113.**
3. Operația de scriere desemnează: **a) afișarea datelor pe monitor; b) scrierea datelor pe suporti magnetici; c) transferul datelor între zone de memorie principală; d) transferul datelor din memoria principală pe suporti externi; e) transferul datelor în buffer.**

## 2.2. Structuri de date

În afara datelor scalare, în aplicațiile practice se utilizează *colecții de date*. Între elementele unei colecții de date pot fi identificate sau, eventual, pot fi introduse relații care să determine pe mulțimea respectivă o anumită structură, adică un mod de ordonare, astfel încât să se faciliteze prelucrarea. O colecție de date pe care s-a definit un mecanism de selectare a elementelor (componentelor), constituie o *structură de date*. Ea este o entitate de sine stătătoare, individualizabilă prin nume, ale cărei componente își mențin proprietățile (tipul). Selectarea componentelor unei structuri se poate realiza folosind identificatorii asociați acestora (accesul prin nume) sau prin poziția pe care o ocupa în structură, în conformitate cu ordinea specificată.

● După modul de selectare a componentelor, se disting: *structuri cu acces direct*, la care o componentă poate fi selectată fără a ține seama de celelalte componente; *structuri cu acces*

*secvențial*, la care localizarea unei componente se face printr-un proces de parcurgere a mai multor componente, conform cu ordinea acestora (traversare).

- După tipul componentelor, structurile de date pot fi împărțite în *omogene*, când componentele sunt de același tip și *neomogene*, când componentele sunt de tipuri diferite.

- Componentele unei structuri de date pot să fie date elementare (scalare) sau să fie ele însele structuri de date. Dacă o structură de date se compune din (sau se poate descompune în) structuri de același tip, se spune că structura respectivă de date este *recursivă*.

- Structurile de date se pot regăsi în memoria internă (*structuri interne*) sau pe un purtător extern (bandă magnetică, disc magnetic, dischetă etc.), caz în care se numesc *structuri externe* (fișiere).

- Dacă în reprezentarea structurii de date pe suportul de memorie se înregistrează, împreună cu componentele acesteia, și date care materializează relațiile de ordonare, se spune că aceasta este o *structură explicită* (altfel, este *implicită*), iar datele suplimentare, de obicei adrese, se numesc *referințe* sau *pointeri*. Necesitatea acestor date suplimentare apare în cazul reprezentării dispersate a structurilor de date, în care componentele nu ocupă zone adiacente pe suportul de memorare. Reprezentarea componentelor în zone adiacente corespunde structurilor dense. Toate structurile de date care au aceeași organizare și sunt supuse acelorași operații formează un anumit *tip de structură de date*, extinzând noțiunea de tip de dată și asupra mulțimilor ordonate de date.

- În funcție de modul de alocare a zonelor de memorie se disting date de tip *static* sau *dinamic*. Pentru datele de tip static sunt alocate zone de memorie bine definite, în momentul compilării. Pentru datele de tip dinamic, zonele de memorie sunt alocate în momentul execuției, în funcție de necesități, fiind posibilă chiar eliberarea memoriei ocupate de unele date. Tipul dinamic corespunde tipului referință (pointer).

### Structuri statice de date

Pe lângă datele elementare (scalare), fie ele constante sau variabile, frecvent este necesară introducerea în memoria internă a unor colecții de date între elementele cărora există anumite relații. Practica programării a impus, ca structuri de date frecvent utilizate, *masivul* (tabloul) și *articolul*. Aceste structuri își dovedesc utilitatea în aproape toate aplicațiile practice, ceea ce explică implementarea lor - cu unele diferențe și particularități - în majoritatea limbajelor de programare.

- ✚ *Masivul* este o structură de date omogenă, cu una, două sau mai multe dimensiuni. Pentru masivele uni, bi și tridimensionale se utilizează denumirile de vector, matrice, respectiv plane de matrice. Structura este cu acces direct, referirea unui element fiind realizată printr-o construcție sintactică numită *variabilă indexată*, formată din numele masivului și un număr de expresii indiciale (indici) corespunzând dimensiunilor acestuia.

Diversele limbaje de programare implementează structura de masiv în modalități diferite. Limbajul C, care face obiectul acestei lucrări, le implementează necontiguu, bazându-se pe adresarea prin pointeri.

- ✚ *Articolul* este o structură de date eterogenă, de tip arborescent (figura 2.2.). Componentele structurii arborescente, numite câmpuri, sunt individualizate prin nume. Relația de ordine ierarhică dintre câmpuri se precizează fie prin numere de nivel, fie prin incluziuni de articole în articole.

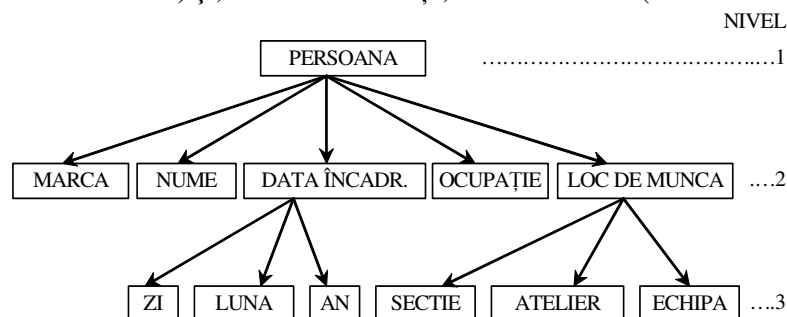
Articolul este o structură cu acces direct, adică fiecare câmp (elementar sau grupat) poate fi referit direct prin numele său, fără a face referiri la celelalte elemente din structura înregistrării. Articolul este o structură recursivă, deoarece datele de grup din interiorul ei corespunzând unor

subarbori, conservă aceleași proprietăți ca și structura în ansamblul său. Una din principalele utilizări ale acestei structuri de date apare în legătură cu fișierele.

Elementele din structura înregistrării sunt:

- *câmpuri elementare* corespunzând nodurilor terminale de pe fiecare ramură a arborelui (în exemplul din figura 2.7: MARCA, NUME, ZI, LUNA, AN, OCUPATIE, SECTIE, ATELIER, ECHIPA);

- *câmpuri grupate* (date de grup), corespunzând nodurilor neterminale (DATA-INCADRARI, LOC-MUNCA) și, în ultimă instanță, chiar rădăcinii (PERSOANA).



**Fig.2.7.** Exemplu de articol

Descrierea unei structuri de tip articol se face prin parcurgerea arborelui în preordine, astfel: se parcurge arborele de sus în jos și de la stânga la dreapta, cu condiția descrierii complete a fiecărui subarbor.

### Structurile dinamice de date

Pentru anumite clase de probleme structurile statice de date nu numai că nu sunt suficiente dar se dovedesc chiar imposibil de folosit datorită limitărilor la care sunt supuse. În primul rând, spațiul de memorie aferent unor astfel de date se definește și se rezervă în momentul compilării programului (rezervare statică), la o dimensiune maximă (cuprinzătoare). Spațiul nu poate fi disponibilizat și nici împărțit cu alte date, chiar dacă nu este în întregime utilizat în anumite momente ale execuției programului. În al doilea rând, componentele structurilor statice ocupă locuri prestabilite în spațiul rezervat, determinate de relația de ordonare specifică fiecărei structuri. În al treilea rând, limbajele de programare definesc operațiile admise cu valorile componentelor, potrivit tipului de bază al structurii, astfel încât numărul maxim și ordinea componentelor structurii nu pot fi modificate.

#### 2.4.1. Grafuri

Un *graf* (sau un *graf neorientat*) este o structură  $G = (V, E)$ , unde  $V$  este o mulțime nevidă iar  $E$  este o submulțime (posibil vidă) a mulțimii perechilor neordonate cu componente distincte din  $V$ . Obiectele mulțimii  $V$  se numesc *vârfuri*, iar obiectele mulțimii  $E$  se numesc *muchii*. Dacă  $e = (u, v) \in E$  se spune că muchia  $e$  are ca extremități  $u$  și  $v$  (este determinată de vârfurile  $u$  și  $v$ ). Graful  $G = (V, E)$  este finit dacă  $V$  este o mulțime finită. În continuare vor fi considerate în exclusivitate grafurile finite, chiar dacă acest lucru nu va fi precizat în mod explicit.

Fie grafurile  $G_i = (V_i, E_i)$ ,  $i = 1, 2$ .  $G_2$  este un *subgraf* al grafului  $G_1$  dacă  $V_2 \subseteq V_1$  și  $E_2 \subseteq E_1$ . Dacă  $G_2$  este un subgraf al lui  $G_1$ ,  $G_2$  este un *graf parțial* al lui  $G_1$  dacă  $V_2 = V_1$ .

Un *graf orientat* (*digraf*) este o structură  $D = (V, E)$ , unde  $V$  este o mulțime nevidă de obiecte numite convențional vârfuri, iar  $E$  este o mulțime (posibil vidă) de perechi ordonate cu componente elemente distincte din  $V$ . Convențional, elementele mulțimii  $E$  sunt numite *arce* sau *muchii ordonate*. Terminologia utilizată relativ la digrafuri este similară celei corespunzătoare grafurilor.

Se numește *graf ponderat* o structură  $(V, E, W)$ , unde  $G=(V, E)$  este un graf, iar  $W$  o funcție,  $W : E \rightarrow (0, \infty)$ . Funcția  $W$  este numită *pondere* și asociază fiecărei muchii a grafului un cost/câștig al parcurgerii ei.

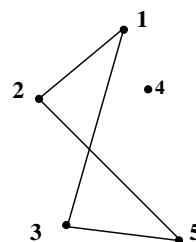
Fie  $G=(V, E)$  un graf,  $u, v \in V$ . Secvența de vârfuri  $\Gamma: u_0, u_1, \dots, u_n$  este un *u-v drum* dacă  $u_0=u$ ,  $u_n=v$ ,  $u_i u_{i+1} \in E$  pentru  $i \in [0, n]$ . Lungimea drumului, notată  $l(\Gamma)$  este egală cu  $n$ . Convențional, se numește drum trivial, un drum  $\Gamma$  cu  $l(\Gamma)=0$ .

Cea mai simplă reprezentare a unui graf este cea intuitivă, *grafică*: fiecare vârf este figurat printr-un punct, respectiv muchiile sunt reprezentate prin segmente de dreaptă orientate (în cazul digrafurilor) sau nu și etichetate (în cazul grafurilor ponderate) sau nu, având ca extremități punctele corespunzătoare vârfurilor care le determină.



Exemple:

1. Fie  $G = (V, E)$  un graf cu  $V = \{1, 2, 3, 4, 5\}$ ,  $E = \{(1,2), (1,3), (2,5), (3,5)\}$ .  
O posibilă reprezentare grafică este:



Deși acest mod de reprezentare este foarte comod și sugestiv în special în cazul grafurilor cu număr mic de vârfuri, pentru prelucrări cu ajutorul calculatorului sunt necesare reprezentări prin intermediul structurilor de date.

O modalitate de reprezentare este cea prin *matrice de adiacență*. Dacă  $G=(V, E)$  este un graf (sau digraf) cu  $|V|=n$ , atunci matricea de adiacență  $A \in M_{n \times n}(\{0,1\})$  are componentele

$$a_{ij} = \begin{cases} 1, & \text{dacă } (v_i, v_j) \in E \\ 0, & \text{altfel} \end{cases}, \text{ unde } v_i, v_j \text{ reprezintă cel de-al } i\text{-lea, respectiv cel de-al } j\text{-lea nod din } V.$$

Se observă că în cazul unui graf neorientat matricea de adiacență este simetrică –  $(\forall) i, j = \overline{1, n}, a_{ij} = a_{ji}$  (perechile de vârfuri care caracterizează muchiile sunt neordonate, deci dacă  $uv \in E$ , atunci și  $vu \in E$ ), în timp ce, în cazul unui digraf, este posibil ca  $(v_i, v_j) \in E$ ,  $(v_j, v_i) \notin E$ , deci  $a_{ij} \neq a_{ji}$ .

În cazul grafurilor ponderate, reprezentarea matriceală este asemănătoare celei prezentate anterior. *Matricea ponderilor* unui graf ponderat  $G=(V, E, W)$ ,  $|V|=n$ ,  $W \in M_{n \times n}((0, \infty))$  are componentele:

$$w_{i,j} = \begin{cases} W(v_i, v_j), & \text{dacă } (v_i, v_j) \in E \\ \alpha, & \text{altfel} \end{cases},$$

unde  $v_i, v_j$  reprezintă cel de-al  $i$ -lea, respectiv cel de-al  $j$ -lea nod din  $V$ ,  $\alpha = 0$  dacă ponderea are semnificația de câștig, respectiv  $\alpha = \infty$  dacă are semnificația de pierdere, în cazul în care se dorește reprezentarea costurilor ca ponderi ale grafului.

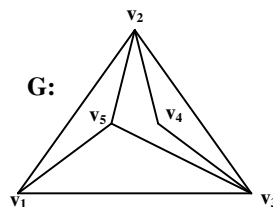
Una dintre cele mai importante proprietăți ale grafurilor o constituie posibilitatea de accesare, prin intermediul unei secvențe de muchii (arce), a oricărui vârf al grafului plecând dintr-un vârf dat, proprietate cunoscută sub numele de *conexitate* sau *conexiune*.

Fie  $\Gamma: u_0, u_1, \dots, u_n$  un drum în graful  $G=(V, E)$ .  $\Gamma$  este un drum închis dacă  $u_0=u_n$ . În caz contrar,  $\Gamma$  se numește drum deschis. Drumul  $\Gamma$  este *elementar* dacă oricare două vârfuri din  $\Gamma$

sunt distincte, cu excepția, eventual, a extremităților. Drumul  $\Gamma$  este *proces* dacă, pentru orice  $0 \leq i \neq j \leq n-1$ ,  $u_i u_{i+1} \neq u_j u_{j+1}$ . Orice drum elementar este un proces.

### Exemplu:

3. Pentru graful  $G$ ,  $\Gamma_1: v_1, v_2, v_3, v_2, v_5, v_3, v_4$  este un  $v_1$ - $v_4$  drum care nu este proces;  $\Gamma_2: v_1, v_2, v_5, v_1, v_3, v_4$  este un  $v_1$ - $v_4$  proces care nu este drum elementar;  $\Gamma_3: v_1, v_3, v_4$  este un  $v_1$ - $v_4$  drum elementar.



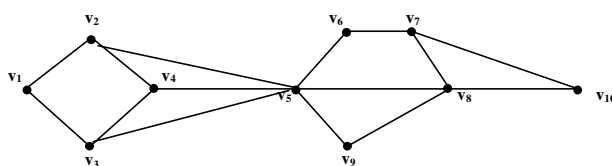
Fie  $\Gamma: u_0, u_1, \dots, u_n$  un drum în graful  $G=(V,E)$ .  $\Gamma': v_0, v_1, \dots, v_m$  este un subdrum al lui  $\Gamma$  dacă  $\Gamma'$  este un drum și pentru orice  $j$ ,  $0 \leq j \leq m$ , există  $i$ ,  $0 \leq i \leq n$  astfel încât  $u_i = v_j$ . Orice drum cu lungime cel puțin 1 conține cel puțin un drum elementar cu aceleași extremități. Într-adevăr, dacă  $\Gamma: u_0, u_1, \dots, u_n$  nu este elementar, atunci există  $0 \leq i < j \leq n$  și  $i \neq 0$  sau  $j \neq n$  astfel încât  $u_i = u_j$ .

Atunci drumul  $\Gamma': \begin{cases} u_j u_{j+1} \dots u_n, & \text{dacă } i = 0 \\ u_0 u_1 \dots u_i, & \text{dacă } j = 0 \\ u_0 u_1 \dots u_i u_{j+1} \dots u_n, & \text{dacă } i \neq 0, j \neq n \end{cases}$  este de asemenea un  $u_0$ - $u_n$  drum.

Aplicând în continuare eliminarea duplicatelor vârfurilor în modul descris, rezultă în final un  $u_0$ - $u_n$  drum elementar.

### Exemplu:

4. În graful



dacă  $\Gamma: v_1, v_2, v_4, v_5, v_3, v_1, v_2, v_5, v_6, v_7, v_8, v_9, v_5, v_9, v_8, v_{10}$ , atunci  $\Gamma_1: v_1, v_2, v_5, v_9, v_8, v_{10}$ ,  $\Gamma_2: v_1, v_2, v_4, v_5, v_9, v_8, v_{10}$  sunt  $v_1$ - $v_{10}$  subdrumuri elementare.

Fie  $G=(V,E)$  un graf,  $|V|=n$ . Dacă  $A$  este matricea de adiacență asociată grafului atunci, pentru orice  $p \geq 1$ ,  $a_{ij}^{(p)}$  este numărul  $v_i$ - $v_j$  drumurilor distincte de lungime  $p$  din graful  $G$ , unde  $A^p = (a_{ij}^{(p)})$ .

Fie  $M_n(\{0,1\})$  mulțimea matricelor de dimensiuni  $n \times n$ , componentele fiind elemente din mulțimea  $\{0,1\}$ . Pe  $M_n(\{0,1\})$  se definesc operațiile binare, notate  $\oplus$  și  $\otimes$ , astfel: pentru orice  $A=(a_{ij})$ ,  $B=(b_{ij})$  din  $M_n(\{0,1\})$ ,  $A \oplus B=(c_{ij})$ ,  $A \otimes B=(d_{ij})$ , unde  $1 \leq i, j \leq n$ ,  $c_{ij} = \max\{a_{ij}, b_{ij}\}$  și  $d_{ij} = \max\{\min\{a_{ik}, b_{kj}\}, 1 \leq k \leq n\}$ . Dacă  $A=(a_{ij}) \in M_n(\{0,1\})$ , se notează  $\{\bar{A}^k = (\bar{a}_{ij}^{(k)}); k \geq 1\}$  secvența de matrice definită prin:

$$\bar{A}^{(1)} = A, \bar{A}^k = A \otimes \bar{A}^{(k-1)}, (\forall) k \geq 2.$$

Dacă  $A$  este matricea de adiacență a unui graf  $G=(V,E)$ , atunci pentru fiecare  $k$ ,  $1 \leq k \leq n-1$ ,

$$\bar{a}_{ij}^{(k)} = \begin{cases} 1, & \text{dacă există drum de lungime } k \text{ de la } i \text{ la } j \\ 0, & \text{altfel} \end{cases}.$$

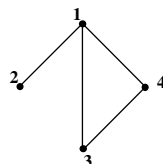
Matricea  $M = \overline{A}^{(1)} \oplus \overline{A}^{(2)} \oplus \dots \oplus \overline{A}^{(n-1)}$  se numește *matricea existenței drumurilor* în graful  $G$ . Semnificația componentelor matricei  $M$  este:

$$(\forall) 1 \leq i, j \leq n, \quad m_{ij} = \begin{cases} 0, & \text{dacă nu există un } v_i - v_j \text{ drum în } G \\ 1, & \text{altfel} \end{cases}$$



**Exemplu:**

5. Pentru graful:



$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}, \quad \overline{A}^2 = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \quad \overline{A}^3 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \quad M = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Calculul matricei existenței drumurilor permite verificarea dacă un graf dat este conex. Un graf este conex dacă și numai dacă toate componentele matricei  $M$  sunt egale cu 1.

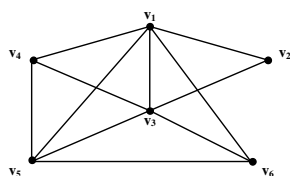
Fie  $G=(V,E)$  un graf netrivial,  $u,v \in V$  și  $\Gamma$  un  $u-v$  drum în  $G$ .  $\Gamma$  se numește proces dacă toate muchiile drumului  $\Gamma$  sunt distincte. Drumul  $\Gamma$  este trivial dacă  $\Gamma: u,u$ . Drumul  $\Gamma$  este un circuit dacă  $\Gamma$  este un proces netrivial închis. Circuitul  $\Gamma: v_1, v_2, \dots, v_n, v_1$  cu  $n \geq 3$  este un ciclu al grafului dacă, pentru orice  $i, j$ , cu  $1 \leq i, j \leq n, i \neq j$ , rezultă  $v_i \neq v_j$ . Orice ciclu este un drum elementar închis. Graful  $G$  este *aciclic* dacă nu există cicluri în  $G$ .

Într-un digraf  $D$  noțiunile de proces, circuit, ciclu sunt definite ca și în cazul grafurilor.



**Exemple:**

6. În graful



$\Gamma_1: v_1, v_2, v_3, v_6, v_5$  este un proces;

$\Gamma_2: v_1, v_2, v_3, v_6, v_5, v_3, v_4, v_1$  este un circuit și nu este ciclu;

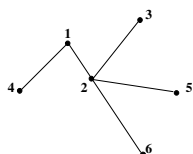
$\Gamma_3: v_1, v_3, v_5, v_4, v_1$  este un ciclu.

### 2.4.2. Arbori

În clasa grafurilor conexe, structurile cele mai simple, dar care apar cel mai frecvent în aplicații sunt cele arborescente. Graful  $G$  este *arbore* dacă  $G$  este aciclic și conex. Fie  $G=(V,E)$  graf arbore. Subgraful  $H=(V_1, E_1)$  al lui  $G$  este un subarbore al lui  $G$  dacă  $H$  este graf arbore.

Exemple:

Graful



este arbore, deoarece pentru orice pereche de vârfuri  $i, j$ ,  $1 \leq i, j \leq 6, i \neq j$ , există un  $i-j$  drum și graful nu conține cicluri.



Verificarea proprietății unui graf de a fi arbore poate fi efectuată pe baza unor algoritmi care să verifice calitățile de conexitate și aciclicitate. Aceeași verificare poate fi realizată și pe baza proprietăților care urmează.

**Proprietatea 1:** Un graf  $G=(V,E)$ , cu  $|V|=n$ ,  $|E|=m$  este graf arbore dacă și numai dacă  $G$  este aciclic și  $n=m+1$ . Cu alte cuvinte, problema verificării dacă un graf este arbore revine la verificarea aciclicității grafului și a relației existente între numărul vârfurilor și numărul muchiilor grafului.

**Proprietatea 2:** Un graf  $G=(V,E)$ , cu  $|V|=n$ ,  $|E|=m$  este graf arbore dacă și numai dacă  $G$  este conex și  $n=m+1$ .

*Notă:* Fie  $G=(V,E)$  un graf. Următoarele afirmații sunt echivalente:

1.  $G$  este graf arbore
2.  $G$  este graf conex minimal (oricare ar fi  $e \in E$ , prin eliminarea muchiei  $e$  graful rezultat nu este conex)
3.  $G$  este graf aciclic maximal (prin adăugarea unei noi muchii în graf rezultă cel puțin un ciclu).

Un graf orientat  $D=(V,E)$  cu proprietatea că pentru orice  $u, v \in V$  dacă  $uv \in E$ , atunci  $vu \notin E$  se numește *graf asimetric*. Digraful  $D$  este *simetric* dacă  $(\forall) u, v \in V$ ,  $uv \in E$ , dacă și numai dacă  $vu \in E$ .

Fie  $D=(V,E)$  digraf netrivial. Graful  $G=(V,E')$ , unde  $E'=\{uv \mid uv \in E \text{ sau } vu \in E\}$  se numește *graf suport* al digrafului  $D$ .

Un arbore orientat este un arbore direcționat cu rădăcină. Deoarece un arbore orientat este un caz particular de digraf, pentru reprezentarea lui poate fi utilizată oricare din modalitățile de reprezentare a grafulor. În plus există și posibilitatea obținerii unor reprezentări mai eficiente pentru acest tip de graf. Una dintre modalități este reprezentarea FIU-FRATE, care constă în numerotarea convențională a vârfurilor grafului și în reținerea, pentru fiecare vârf  $i$  al arborelui, a următoarelor informații:

- FIU( $i$ ), care reprezintă numărul atașat primului descendent al vârfului  $i$ ;
- FRATE( $i$ ), care reprezintă numărul atașat vârfului descendent al tatălui vârfului  $i$  și care urmează imediat lui  $i$ ;
- INF( $i$ ), care reprezintă informația atașată vârfului  $i$  (de obicei valoarea  $i$ ).

Pentru reprezentarea arborelui se reține rădăcina și numărul nodurilor. Absența “fiului”, respectiv a “fratelui” unui vârf este marcată printr-o valoare diferită de numerele atașate vârfurilor (de obicei valoarea 0).

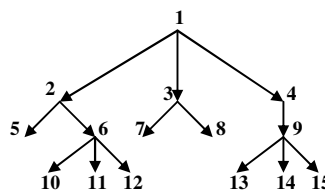
Exemplu:

Următorul arbore orientat este reprezentat astfel:

$N=15$  (numărul de noduri ale arborelui);

$R=1$  (rădăcina); FIU=(2,5,7,9,0,10,0,0,13,0,0,0,0,0,0), “fiul” lui 1 este 2, iar vârful 9 are “fiul”

13; FRATE=(0,3,4,0,6,0,8,0,0,11,12,0,14,15,0), vârful 1 nu are frate, iar vârful 14 are fratele 15.



O parcurgere revine la aplicarea sistematică a unei reguli de vizitare a vârfurilor grafului. Cele mai uzuale reguli de parcurgere a arborilor orientați sunt prezentate în continuare.

A. *Parcurgerea în A-preordine* presupune inițial vârful curent ca fiind rădăcina arborelui. Se vizitează vârful curent și sunt identificați descendenții lui. Se aplică aceeași regulă de vizitare pentru arborii având ca rădăcini descendenții vârfului curent, arborii fiind vizitați în ordinea dată de numerele atașate vârfurilor rădăcină corespunzătoare.

B. *Parcurgerea A-postordine* diferă de parcurgerea în A-preordine numai prin faptul că rădăcina fiecărui arbore este vizitată după ce au fost vizitate toate celelalte vârfuri ale arborelui.

*Notă:* Parcurgerile în A-preordine și A-postordine sunt variante de parcurgeri în adâncime, în cazul ambelor metode fiind prioritare vârfurile aflate la distanță maximă față de rădăcina arborelui inițial.

C. *Parcurgerea pe niveluri*. Un vârf  $v$  al unui arbore orientat cu rădăcină  $r$  se află pe *nivelul*  $i$  al arborelui, dacă distanța de la vârf la rădăcină (lungimea  $r$ - $v$  drumului) este egală cu  $i$ . Rădăcina arborelui este de nivel 0. Parcurgerea pe niveluri a unui arbore orientat constă în vizitarea vârfurilor sale în ordinea crescătoare a distanțelor față de rădăcină.

Un *arbore binar* este un arbore orientat cu proprietatea că orice vârf  $v$  are maxim doi descendenți ( $\text{od}(v) \leq 2$ ). În cazul  $\text{od}(v)=2$ , cei doi descendenți sunt desemnați ca descendent stâng (fiu stânga), respectiv descendent drept (fiu dreapta). Pentru vârfurile cu  $\text{od}(v)=1$ , unicul descendent este specificat fie ca fiu stânga, fie ca fiu dreapta.

Se numește *nod terminal* (*frunză*) orice vârf  $v$  al arborelui cu  $\text{od}(v)=0$ . Nodul  $v$  este *neterminal* dacă  $\text{od}(v)>0$ . Reprezentarea unui arbore binar poate fi realizată prin reținerea, pentru fiecare nod, a legăturilor către descendenții lui, absența unui descendent putând fi reprezentată prin valoarea nulă (nil).

identificator nod	legătură fiu stânga	legătură fiu dreapta
-------------------	---------------------	----------------------

Datorită particularității lor, arborii binari oferă posibilitatea aplicării de noi metode de parcurgere (pe lângă cele aplicabile pentru arborii generali): parcurgerile în *preordine* (RSD), *inordine* (SRD) și *postordine* (SDR). Regula de vizitare pentru aceste tipuri de parcurgere revine la parcurgerea subarborilor stâng și drept corespunzători vârfului curent, care la momentul inițial este chiar rădăcina arborelui. Diferența între aceste trei tipuri de parcurgere este dată de momentul în care devine vizitat fiecare vârf al arborelui. În parcurgerea RSD (rădăcină-subarbore stâng-subarbore drept), fiecare vârf al arborelui este vizitat în momentul în care devine vârf curent. În parcurgerea SRD (subarbore stâng-rădăcină-subarbore drept), vizitarea vârfului este efectuată după ce a fost parcurs subarborele stâng. În parcurgerea SDR (subarbore stâng-subarbore drept-rădăcină) vizitarea fiecărui vârf este efectuată după ce au fost parcurși subarborii aferenți lui.



### Exemplu:

Pentru arborele de la exemplul anterior, secvențele de vârfuri rezultate prin aplicarea parcurgerilor RSD, SRD, SDR sunt:

- ✖ preordine: 1,2,4,7,5,3,6,8,9
- ✖ inordine: 4,7,2,5,1,8,6,9,3
- ✖ postordine: 7,4,5,2,8,9,6,3,1.

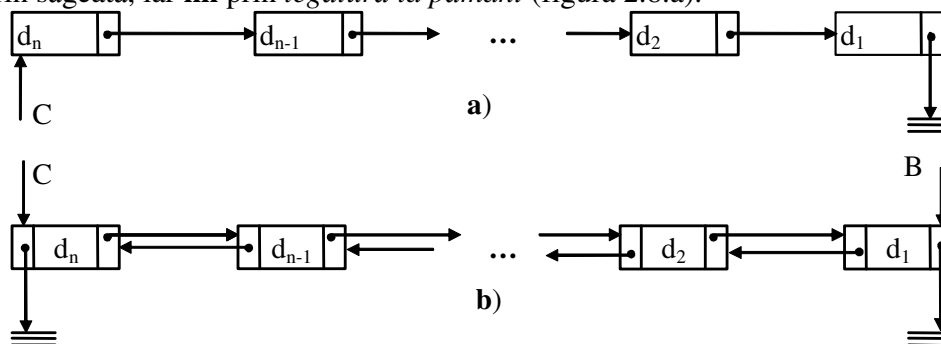
### 2.4.3. Liste

Lista este o structură dinamică de date formată din mai multe *noduri*. Un nod este o structură de date care conține două părți distincte: o parte de informație utilă (memorată în nodul respectiv) și o parte de informație de legătură (folosită pentru a identifica în memorie următorul nod din listă).

Grafurile pot fi reprezentate prin intermediul *listelor*, permițând utilizarea economică a spațiului de memorare și, în anumite cazuri, implementări mai eficiente pentru anumite clase de algoritmi. Vârfurile grafului se memorează într-o listă, fiecare celulă a listei având o legătură către lista vecinilor acelui vârf (vârfurile din graf adiacente cu vârful corespunzător acelei celule și indicat ca informație utilă).

În cele ce urmează se consideră un spațiu oarecare de memorie adresabilă. Se notează mulțimea referințelor din acest spațiu cu  $\mathbf{P}$  și se adăugă la ea o valoare referință specială, notată **nil**. Se notează cu  $\mathbf{D}$  mulțimea informațiilor care vor fi conținute de nodurile unei liste. În aceste condiții, un nod poate fi desemnat prin perechea  $(d_i, p_i)$  cu  $d_i \in \mathbf{D}$  și  $p_i \in \mathbf{P}$ .

● O mulțime  $L_a = \{(d_i, s_i) \mid d_i \in \mathbf{D}, s_i \in \mathbf{P}\}$  este o *listă simplu înlănțuită* (sau *asimetrică*) dacă pe  $L_a$  s-a definit o structură liniară și  $s_1 = \mathbf{nil}$ , iar  $s_i$  ( $i > 1$ ) este referință spre succesorul direct al lui  $d_i$ . Pentru reprezentarea unei liste asimetrice, informațiile de legătură (pointerii) vor fi marcate prin săgeată, iar **nil** prin *legătură la pământ* (figura 2.8.a).



**Fig. 2.8.** Liste simplu (a) și dublu (b) înlănțuite

● O mulțime  $L_s = \{(p_i, d_i, s_i) \mid d_i \in \mathbf{D}, p_i, s_i \in \mathbf{P}\}$  este o *listă dublu înlănțuită* (sau *simetrică*) dacă pe  $L_s$  s-a definit atât o structură liniară directă, cât și inversa sa și  $s_1 = p_n = \mathbf{nil}$ , iar  $s_i$ ,  $i \neq 1$  și  $p_i$ ,  $i \neq n$  sunt referințe spre succesorii direcți în ordinea specifică (figura 2.3.b).

În cazul listelor dublu înlănțuite, un nod cuprinde două referințe, fiind posibilă cunoașterea atât a predecesorului, cât și a succesului imediat. Se remarcă, din definiție, că nodurile unei liste pot apărea dispersate în spațiul de memorare, iar referințele creează mecanismul care le leagă. De aici derivă și atributul de *înlănțuită*, care o deosebește de cazul particular, denumit *listă densă*. La o listă densă nodurile sunt dispuse adiacent în spațiul de memorare și informațiile de legătură, adică referințele, devin de prisos (ca în cazul vectorilor).

Pentru listele înlănțuite, nodurile pot fi plasate în memoria calculatorului în zone diferite, informația de legătură din fiecare nod asigurând regăsirea nodului următor (pentru liste asimetrice) respectiv a nodului următor și succesori (pentru liste simetrice). Pentru a putea lucra cu aceste liste este nevoie să se cunoască o singură informație: adresa primului nod din listă, numit în continuare *capul listei*. În acest sens se asociază listei o referință **C**, care conține fie adresa acestui nod, fie **nil**, dacă lista este vidă. Spunem că o listă este complet identificată dacă se cunoaște adresa primului nod al său și structura unui nod.

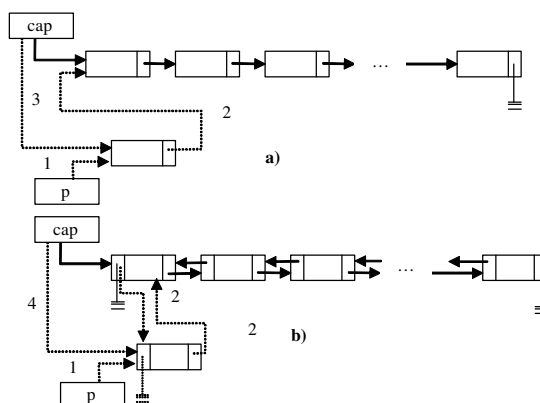
Asupra unei liste se pot realiza o multitudine de operații: traversare, căutare nod, numărare noduri etc. Sunt însă tipice operațiile de *inserare* (adăugare) și *ștergere* de noduri, deoarece dau însuși caracterul dinamic al acestor structuri. Prin inserări și ștergeri o listă „crește” și „descrește” în timp (ca număr de noduri), solicitând spațiu pentru noile componente și

eliberând spațiul ocupat de componentele șterse. Astfel, structura și numărul de noduri ale listei suferă o permanentă schimbare.

1. *Traversarea.* Localizarea unui nod prin traversare este o operație de căutare. Scopul este de a găsi un nod care să conțină în partea de informație utilă anumite date. Operația de căutare se poate încheia cu succes, adică cu localizarea unui astfel de nod sau fără succes, în cazul în care lista este vidă sau nu există un nod care conține informațiile cerute.

2. *Inserarea.* Inserarea unui nod într-o listă înlănțuită se poate realiza cu verificarea existenței anterioare a nodului în listă, caz în care nu se mai face inserarea, sau fără nici un fel de verificare, caz în care se procedează direct la inserare (se permit noduri duplicate). În ambele cazuri inserarea se poate face în mai multe locuri: la începutul listei, la sfârșitul listei, după sau înaintea unui anumit nod identificat printr-o cheie.

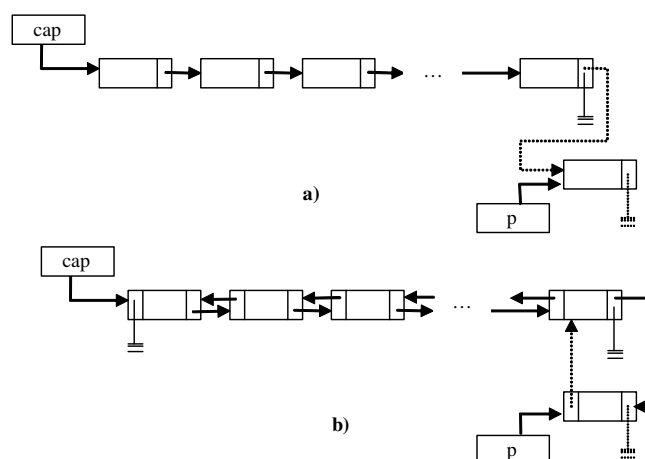
➤ Inserarea unui nod la începutul listei. Pentru inserarea într-o listă simetrică este necesară crearea legăturilor între noduri în ambele sensuri. Singura diferență față de inserarea într-o listă asimetrică este înscrierea informației despre nodul precedent. Deoarece inserarea are loc la începutul listei, nu va exista un nod precedent, deci informația respectivă va primi valoarea *nil*. Primul nod din lista inițială va avea acum ca precedent nodul nou creat *p* (figura 2.9). Se observă că întotdeauna se modifică capul listei, nodul inserat devenind noul cap al listei.



**Fig. 2.9.** Inserarea la începutul listelor asimetrice (a) și simetrice (b)

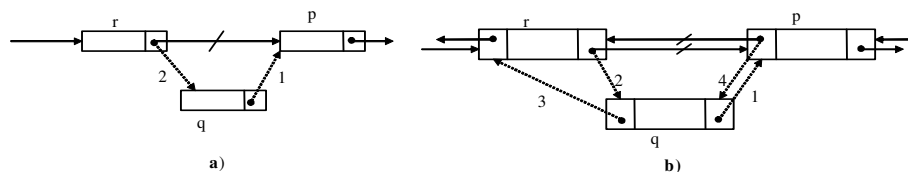
➤ Inserarea unui nod la sfârșitul listei. Pentru adăugarea unui nod la sfârșitul listei se creează întâi un nod nou cu informația utilă care trebuie inserată. Deoarece va fi ultimul nod în listă, nu există un nod următor deci informația de legătură spre nodul următor va avea valoarea **nil**.

Dacă lista este vidă atunci capul listei ia ca valoare adresa noului nod creat (va fi singurul nod din listă). În caz contrar trebuie parcursă lista pentru a ajunge la ultimul nod, informația de legătură din acesta primind ca valoare adresa noului nod creat. În cazul unei liste simetrice trebuie creată și legătura în sens invers (figura 2.10).



**Fig. 2.10.** Inserarea la sfârșitul listelor asimetrice (a) și simetrice (b)

➤ Inserarea după un anumit nod al listei. Prima operație necesară inserării este localizarea nodului după care se face inserarea. Acesta poate fi localizat în mai multe feluri: în mod exact, prin furnizarea unei chei a cărei valoare trebuie să se regăsească în informația utilă a nodului căutat, sau relativ, prin stabilirea unei condiții care trebuie să fie îndeplinită de nod. În continuare vom considera cazul căutării după cheie. Dacă nu se găsește nici un nod care să corespundă criteriului de căutare (cazul căutării cu eșec) sunt posibile mai multe decizii: nu se inserează noul nod, se inserează la începutul listei sau se inserează la sfârșitul listei. Inserarea înseamnă „ruperea” unei legături între două noduri și crearea unor noi legături astfel ca noul nod să se interpună în listă între cele două noduri a căror legătură a fost „ruptă”, conform figurii 2.11.



**Fig. 2.11.** Inserare în listă asimetrică (a) și simetrică (b)

3. *Ștergere*. La fel ca și inserarea, ștergerea se realizează mai eficient și mai ușor din punct de vedere algoritmic dacă ea privește nodul cap de listă, deoarece se reduce, în principiu, la actualizarea referinței de cap. Dacă trebuie șters un nod interior listei, atunci ștergerea propriu-zisă trebuie precedată de o căutare după conținutul informațional al nodurilor. Dacă listele simplu înlanțuite pot fi traversate într-un singur sens, în momentul în care s-a localizat nodul **p** de șters nu se mai cunoaște precedentul său **q** care, pentru eliminarea nodului **p**, trebuie să fie legat de succesorul acestuia (figura 2.12.a). Datorită acestei situații, este necesară introducerea unei referințe suplimentare **q** care, în procesul de traversare, să fie în urma lui **p** cu un nod. De aceea, **p** se numește *referință de urmărire*. Inițial, când **p=nil**, se atribuie aceeași valoare lui **q**. Se precede atribuirea de valoare pentru **p** care schimbă nodul curent, de atribuirea **q=p**.

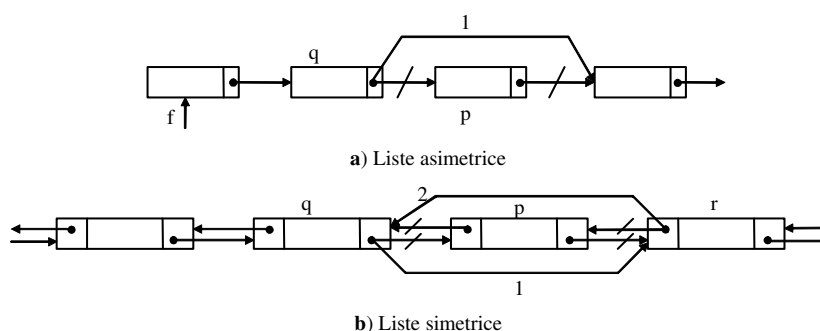


Fig. 2.12. Ștergere în liste înlănțuite

#### 2.4.4. Stive și cozi

Stivele și cozile sunt liste particulare din punctul de vedere al operațiilor de ștergere și inserare ale nodurilor din listă.

- **Stiva (stack)** este o listă asimetrică (figura 2.13.a) la care operațiile de inserare, ștergere și citire a informațiilor se fac numai în capul listei (**top**). Stivele se mai numesc și liste LIFO (**Last-In-First-Out** - ultimul intrat, primul ieșit), având în vedere că orice acces se poate face numai la ultimul nod inserat. Acesta poate fi citit, șters sau în fața lui se poate insera un nou nod care devine cap de stivă. Un astfel de comportament este bine sugerat de modul în care se gasează și se scot vagoanele într-o linie moartă (închisă) sau de modul în care se pot stivui și utiliza scândurile într-un depozit etc. Stiva este o structură frecvent folosită în gestionarea dinamică a spațiului de memorie și reprezintă un mecanism curent în procesul transferului de parametri între un apelator și un subprogram etc.

- **Coadă (queue)** este o listă asimetrică la care operația de inserare se face la un capăt, denumit spatele cozii, iar ștergerea și citirea se fac de la celălalt capăt, denumit fața cozii (figura 2.13.b). Coadă este denumită și listă FIFO (**First-In-First-Out** - primul intrat, primul ieșit) având în vedere că numai nodul din față poate fi citit sau șters, adică primul nod al listei. Comportamentul unei astfel de structuri este de fapt un model abstract al cozilor întâlnite în diferite situații: cozi de persoane, cozi de automobile etc.

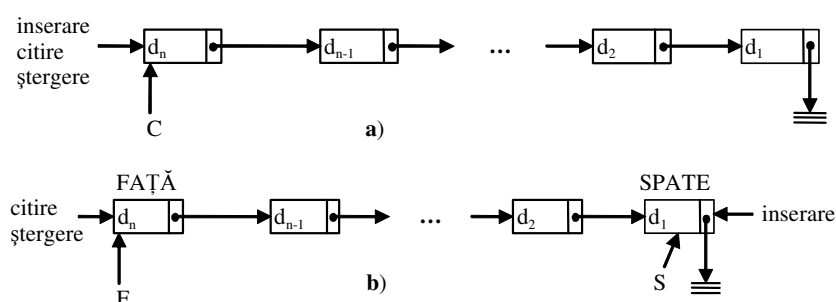


Fig. 2.13. Stivă și coadă

În timp ce pentru a trata corect o stivă este suficientă o referință spre **top**, pentru coadă trebuie menținute două referințe: una pentru fața (**f**) și alta pentru spatele cozii (**s**). O coadă este vidă atunci când **f=s=nil**.



### Teste de autoevaluare

4. Structura de date se definește ca: a) o colecție de date pe care s-a definit un mecanism de selectare a componentelor; b) o colecție de date la care o componentă este independentă de celelalte; c) o colecție de date compusă din subcolecții de același tip; d) o colecție de date compusă din subcolecții de tipuri diferite; e) o colecție recursivă de date.
5. Masivul este o structură: a) recursivă; b) omogenă cu acces secvențial; c) omogenă cu acces direct; d) eterogenă cu acces secvențial; e) eterogenă cu acces direct.
6. Articolul este o structură: a) dinamică; b) omogenă cu acces secvențial; c) omogenă cu acces direct; d) eterogenă cu acces secvențial; e) eterogenă cu acces direct.
7. Stiva este o listă la care: a) inserarea și ștergerea se fac la capul listei și citirea se face la baza listei; b) inserarea, ștergerea și citirea se fac la capul listei; c) inserarea, ștergerea și citirea se fac la baza listei; d) inserarea se face la capul listei, iar ștergerea și citirea se fac la baza listei; e) inserarea și ștergerea se fac la baza listei și citirea se face la capul listei.

### Răspunsuri și comentarii la testele de autoevaluare

1:c); 2:d); 3:d); 4:a); 5:c); 6:e); 7:b).

### Rezumat

În cadrul acestei unități de învățare au fost studiate următoarele aspecte în ceea ce privește datele și structurile de date:

- ➡ cunoștințe teoretice privind conceptele de informație, dată, cunoștință;
- ➡ modalități de reprezentare internă a datelor;
- ➡ structuri statice și dinamice de date.

După încheierea acestei unități de învățare, studenții au cunoștințe și abilități de reprezentare a datelor și a structurilor de date.

### Bibliografia unității de învățare

1. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, M. Mircea, L. Bătăgan, C. Silvestru, Bazele programării calculatoarelor. Teorie și aplicații în C, Ed. ASE, București, 2006, ISBN 973-594-591-6
2. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, Programarea calculatoarelor. Știința învățării unui limbaj de programare, Teorie și aplicații, Ed. ASE, 2003

### 3. Etapele rezolvării problemelor cu calculatorul

#### Cuprins

- Obiectivele unității de învățare 3
- 3.1. Caracteristici generale ale PPAD
- 3.2. Fazele dezvoltării programelor
- Răspunsuri și comentarii la testele de autoevaluare
- Bibliografia unității de învățare

#### Obiectivele unității de învățare 3

Dupa studiul acestei unitati de învățare, studenții vor avea cunoștințe teoretice și abilități practice despre:

- ➡ caracteristici generale ale PPAD;
- ➡ organizarea procesului de rezolvare a PPAD;
- ➡ fazele dezvoltării programelor.



*Durata medie a unității de studiu individual - 2 ore*

#### 3.1. Caracteristici generale ale PPAD

„Probleme de prelucrare automată a datelor” (PPAD) este o denumire generică pentru aplicațiile practice ale informaticii în economie sau în alte domenii de activitate. Având în vedere marea diversitate a PPAD, în cele ce urmează se propune o structurare pe clase a acestora și se analizează tipurile de soluții între care se poate alege în procesul de informatizare. Aplicațiile informatice acoperă un evantai foarte larg de situații, ale cărui extreme pot fi caracterizate astfel:

a) existența unui volum relativ redus de date de intrare și de ieșire, dublată de calcule laborioase (aplicații tehnico-inginerești, care conduc la soluții bazate pe aproximări succesive, exprimate matematic prin ecuații algebrice și transcendente, sisteme de ecuații liniare, ecuații diferențiale, integrale, adică, într-o formulare globală, aplicații bazate pe metode de calcul numeric);

b) existența unui volum mare de date de intrare și de ieșire, asociată uzual cu calcule de complexitate redusă (evidența materialelor, evidența personalului și calculul salariilor, evidența mijloacelor fixe, urmărirea realizării producției, a contractelor de aprovizionare/desfacere etc., adică, în general, aplicații de gestiune economică).

Între aceste extreme există, evident, o diversitate de aplicații, dintre care unele presupun atât volume mari de date, cât și modele economico-matematice bazate pe calcule laborioase: gestiunea stocurilor, programarea producției, probleme de transport, croire, alocare a resurselor etc. În contextul prezentului capitol se consideră suficientă și relevantă doar investigarea caracteristicilor celor două mari clase de aplicații evidențiate, realizată succint în cele ce urmează.

● PPAD din categoria a) permit, în general, soluții simple de organizare a datelor care, frecvent, se reduc la preluarea integrală în memoria internă a calculatorului, cu constituirea lor în



structuri de tip masiv (vectori, matrice, plane de matrice etc.). În cazul unor volume mai mari de date se poate folosi și memoria externă, văzută ca o prelungire a celei interne, cu organizarea datelor în fișiere (uzual, fișiere relative). În majoritatea cazurilor, chiar când datele sunt organizate în fișiere, nu se ajunge la păstrarea lor pe perioade mari de timp, adică nu apare necesitatea actualizării datelor (adăugări, modificări, ștergeri). Singura problemă în accesul la datele organizate în fișiere o reprezintă trecerea de la memorarea liniară a elementelor în fișier, la caracterul lor de elemente ale unor masive (tablouri) cu două sau mai multe dimensiuni.

Pe exemplul matricelor, alegând liniarizarea pe linii (ordinea lexicografică), în cazul unei matrice  $A_{m \times n}$ , poziția în fișierul relativ a unui element  $a_{ij}$  poate fi determinată printr-un calcul simplu, folosind funcția rang  $r(i,j)=n(i-1)+j$ ,  $i \leq m$ ,  $j \leq n$ . Această tehnică se poate folosi la memorarea datelor în fișier (crearea și popularea fișierului), permițând chiar furnizarea elementelor într-o ordine oarecare, cu condiția precizării coordonatelor. La regăsirea elementelor masivului (consultarea fișierului), pornind de la poziția în fișier, pot fi determinate coordonatele elementului, astfel:

- dacă  $r$  modulo  $j = 0$ , atunci  $i=[r/n]$  și  $j=n$ ;
- dacă  $r$  modulo  $j <> 0$ , atunci  $i=[r/n]+1$  și  $j=r$  modulo  $n$ .

În concluzie, la această clasă de **PPAD** atenția principală se deplasează către algoritmul de calcul, bazat, uzual, pe aproximări succesive, ce pot conduce la apariția unor tipuri specifice de erori (erori de trunchiere și erori de rotunjire), care uneori se compensează, dar frecvent se cumulează (se propagă) pe parcursul prelucrării, afectând corectitudinea rezultatului final.

❖ Datorită volumelor mari de date, la problemele din categoria **b)** accentul principal cade pe organizarea și întreținerea colecțiilor de date memorate pe purtători externi. La limită, se poate alege între două soluții: organizarea datelor în fișiere (secvențiale, relative sau indexate) sau organizarea datelor în baze de date (cu structuri ierarhice, de tip rețea sau relaționale).

Indiferent de nivelul de organizare ales, tehnologia de prelucrare conduce la lanțuri (sisteme) de programe, în general de două tipuri: programe de creare și actualizare (întreținere) a colecțiilor de date; programe de consultare (interogare), pentru satisfacerea cererilor de informare. În algoritmi, principalele tipuri de operații sunt cele de prelucrare a colecțiilor de date organizate pe purtători externi: citire, scriere, rescriere, ștergere, precum și cele de verificare a corectitudinii datelor (operații de validare). Complexitatea calculelor este redusă, adesea nedepășind nivelul celor patru operații aritmetice.

În general, în rezolvarea PPAD se poate distinge între următoarele variante de soluții: soluții proprii; utilizarea pachetelor de programe aplicative; soluții mixte. Alegerea între cele trei variante depinde de mai mulți factori, dintre care o importanță aparte are aria (dimensiunile) problemei: un domeniu de activitate strict delimitat (*aplicație informatică*), respectiv ansamblul activităților unui organism economico-social (*sistem informatic*).

❷ În general, *soluțiile proprii* sunt posibile pentru orice PPAD. Din perspectiva costurilor implicate de diversele faze ale realizării unei soluții informatice, această variantă este, frecvent, cea mai neeconomică. Totuși, adesea aceasta este singura variantă accesibilă, după cum se poate constata și din prezentarea următoarelor cazuri:

- la nivel de aplicație informatică, atunci când domeniul de activitate are o serie de particularități care nu permit utilizarea unor eventuale pachete de programe aplicative și, evident, când nu există asemenea pachete de programe pentru domeniul respectiv;
- la nivel de sistem informatic, datorită mării diversități a organismelor economico-sociale, atunci când nu se poate pune problema unor soluții tip, cu posibilitatea ca unele subsisteme (aplicații) să fie realizate prin soluții mixte.

❸ *Utilizarea unor pachete de programe aplicative* este posibilă pentru activitățile cu un anumit grad de generalitate, care se regăsesc în forme identice (sau cu diferențe nesemnificative)

la mai multe organisme economico-sociale. Printre acestea pot fi menționate aplicațiile bazate pe modele de programare liniară, gestiunea stocurilor, optimizarea transporturilor, optimizarea croirii, gestiunea financiar-contabilă etc.

Adoptarea unei asemenea soluții elimină costurile de proiectare și realizare a aplicațiilor (sistemelor) informatice, dar poate conduce la unele necesități de adaptare a sistemului informațional propriu, în funcție de datele de intrare și modul de organizare a acestora, impuse pachetelor de programe. Pe un plan mai general, în această categorie pot fi incluse și produse care, fără a furniza direct soluții, simplifică substanțial efortul de realizare a lor, pentru domenii specifice. De exemplu, pentru prelucrări de serii de date statistice, ca și pentru orice alte date ce se organizează în tabele, pot fi utilizate produsele din categoria *spread-sheet* (foaie de calcul), cum este EXCEL.

❷ *Soluțiile mixte* presupun încadrarea în soluții proprii a unor pachete de programe aplicative, destinate unora dintre aplicațiile unui sistem informatic. Această variantă reduce, de asemenea, costurile de realizare, dar necesită elaborarea unor interfețe între intrările/ieșirile pachetelor de programe și celelalte componente ale sistemului (aplicației) pentru care se elaborează soluții proprii.

Realizarea sistemelor (aplicațiilor) informatice este un proces complex, structurat în mai multe etape, cu obiective specifice. În modul de realizare a acestor etape, în resursele implicate și în durata lor totală vor exista o serie de diferențe, impuse, pe de o parte, de varianta de soluție aleasă și, pe de altă parte, de nivelul de abordare (aplicație, sistem informatic). Cu aceste observații, într-o abordare schematică, redusă la enumerarea și prezentarea obiectivelor principale, etapele de realizare a sistemelor informatice sunt următoarele:

- ❖ *studiul și analiza sistemului informațional actual*, care are ca obiectiv principal formularea cerințelor și a restricțiilor pentru sistemul informatic (caietul de sarcini);

- ❖ *proiectarea de ansamblu*, care presupune elaborarea modelului de ansamblu al sistemului informatic și planificarea realizării sale pe părți componente (subsisteme, aplicații);

- ❖ *proiectarea de detaliu*, care are ca obiective elaborarea modelului de detaliu al sistemului informatic și stabilirea soluțiilor tehnice de realizare (corespunzător, se face distincție între *proiectarea logică de detaliu* și *proiectarea fizică de detaliu*);

- ❖ *elaborarea programelor*, în care se realizează, conform priorităților stabilite în etapa anterioară și pe baza documentațiilor reunite în proiectul logic de detaliu și proiectul tehnic de detaliu, programele incluse în fluxurile tehnologice specifice diverselor tipuri de prelucrări;

- ❖ *implementarea și exploatarea sistemului*, care presupune darea în funcțiune a sistemului și utilizarea curentă a acestuia.

Pe baza diferențierilor evidențiate, în cele ce urmează se abordează numai cazul unor aplicații informatice de mai mică amploare, pentru care se menține în continuare denumirea de PPAD.

### **Organizarea procesului de rezolvare a PPAD**

După complexitatea problemelor abordate în vederea rezolvării cu calculatorul electronic, din perspectiva programelor ce vor fi elaborate, soluțiile proprii pot conduce la una din următoarele situații: *sisteme de programe* realizate la nivelul unor aplicații informatice, abordate independent sau ca parte a unor sisteme informatice; *unul sau câteva programe*, în cazul unor probleme de amploare redusă.

În primul caz, realizarea programelor este doar una dintre etapele procesului de studiu, analiză, proiectare și implementare a sistemelor sau aplicațiilor informatice. Depinzând de tipul aplicației, acuratețea și generalitatea abordării, este posibil ca soluțiile de acest tip să evolueze către realizarea unor pachete de programe aplicative. În al doilea caz, se ajunge frecvent la

realizarea tuturor etapelor, de la punerea problemei până la rezolvarea ei, de către o singură persoană. În general, pot fi abordate în această variantă unele probleme tehnico-științifice, care nu conduc la cerințe deosebite privind volumul și organizarea datelor și sunt, uzual, probleme relativ independente.

Există, de asemenea, unele probleme economico-sociale sau administrative în care, datorită numărului mic de informații utilizate, modificărilor reduse, precum și cerințelor simple de prelucrare să nu fie necesare soluții complexe, bazate pe realizarea unor sisteme de programe. Etapele de rezolvare a PPAD sunt:

■ *Formularea problemei.* În această etapă se precizează rezultatele care trebuie să se obțină și forma lor de prezentare, datele de intrare și modul lor de organizare, precum și transformările și prelucrările care se aplică asupra datelor de intrare pentru a se obține rezultatele. Formularea clară, corectă și completă a problemei reprezintă o cerință importantă pentru finalizarea cu bune rezultate a activității de proiectare și realizare a programelor. În cazul aplicațiilor de mai mari dimensiuni, a căror rezolvare se abordează de către o echipă de analiză-proiectare-programare, formularea problemei poate fi primită de programator de la analistul-proiectant, sub forma unor specificații de problemă.

■ *Formalizarea matematică și alegerea metodei numerice.* În această etapă se exprimă matematic toate transformările și prelucrările la care sunt supuse datele de intrare pentru a se obține rezultatele, se pun în evidență condițiile inițiale și restricțiile referitoare la soluție. La alegerea metodei numerice de rezolvare trebuie analizate cu atenție stabilitatea și convergența metodei, tipul, mărimea și propagarea erorilor, precizia rezultatelor. De asemenea, este necesar să se estimeze necesarul de memorie, timpul de prelucrare, deoarece unele dintre metode nu pot fi aplicate tocmai datorită solicitării acestor resurse peste limitele admisibile. Se recomandă ca în etapa formalizării matematice să se stabilească și ordinea de efectuare a prelucrărilor, să se rețină informațiile care sunt necesare următoarei etape (variația indicilor, momentul introducerii sau extragerii datelor, inițializarea variabilelor și masivelor etc.). Este necesară precizarea că această etapă este cu deosebire importantă în cazul problemelor tehnico-inginerești sau economice bazate pe folosirea modelelor matematice.

■ *Elaborarea (proiectarea) algoritmilor.* Este o etapă complexă care presupune analiza riguroasă a problemei și a formalizării ei matematice. Una dintre metodele uzuale de elaborare a algoritmilor este proiectarea structurată, ale cărei reguli se vor aplica, de asemenea, în faza de scriere a programelor. Uzual, această etapă se încheie cu reprezentarea algoritmului într-o anumită formă (schemă logică structurată, pseudocod etc.), simplificând substanțial eforturile depuse în etapele de scriere, testare și definitivare a programelor. În același sens, este foarte importantă faza testării algoritmilor.

■ *Stabilirea resurselor.* Pe baza analizei algoritmilor se precizează, pentru întregul algoritm și pe module, echipamentele periferice necesare, în ce limbaj urmează să fie scris fiecare modul, care sunt seturile de date ce vor fi folosite la testare, se aproximează necesarul de memorie internă, se stabilește dacă este necesară segmentarea programului pentru reacoperire etc.

■ *Scrierea programelor.* Pe baza algoritmului elaborat se realizează codificarea modulelor, eventual în paralel dacă se lucrează în echipă. Scrierea programului trebuie să se facă respectând strict algoritmul (eventual se pot face modificări și detalieri ale acestuia) urmărindu-se, în același timp, optimizarea utilizării resurselor calculatorului (timp unitate centrală, memorie internă și echipamente periferice). În cazul programelor interactive este necesară realizarea unor interfețe prietenoase, care să lanseze solicitări clare, precise și complete.

■ *Testarea și definitivarea programelor.* Programul scris într-un limbaj sursă va fi compilat și consolidat. În procesul compilării programele sunt testate sintactic, iar în procesul consolidării (editării de legături) se validează utilizarea resursei memorie, modul în care au fost

apelate, implicit sau explicit, procedurile utilizatorului sau ale sistemului, modul în care au fost concepute operațiile de intrare/ieșire sau, în general, cum au fost gestionate resursele de intrare/ieșire etc.

După parcurgerea acestor faze programul se testează în execuție, pe baza unor seturi stabilite de date de control. În această fază programul se consideră corect dacă rezultatele obținute sunt cele scontate (pentru datele de test, programatorii trebuie să determine dinainte rezultatele - valorile și forma lor de prezentare). Se mai face mențiunea că, într-un lanț de programe dintr-un sistem, trebuie testat întreg lanțul, în intercorelările lui interne. Programele considerate corecte se memorează pe medii magnetice, eventual constituite în biblioteci.

■ **Definitivarea documentației programelor.** Deși mai puțin spectaculoasă, chiar „birocratică”, această etapă este obligatorie pentru exploatarea, dezvoltarea și adaptarea programelor, mai ales dacă fac parte dintr-un sistem. Documentarea corectă a programului face posibilă, de asemenea, folosirea lui de către alți utilizatori. S-a arătat că, pe măsura desfășurării precedentelor etape, s-au realizat unele documente necesare exploatării corecte a programelor. Acestea se completează, redactează și ordonează, constituindu-se într-un *dosar de prezentare și exploatare*, care cuprinde:

- ✖ descrierea problemei (rezultate, date de intrare, procesul de prelucrare, formalizarea matematică și modelul matematic, precizia algoritmului etc.);
- ✖ descrierea resurselor necesare și a restricțiilor de utilizare;
- ✖ organigrama modulelor;
- ✖ schemele logice structurate (sau alte forme de reprezentare a algoritmului) pentru fiecare modul;
- ✖ modul de apel al procedurilor și funcțiilor și structura datelor și parametrilor care se transferă între apelator și apelat; instrucțiuni de utilizare (procedura de punere în lucru, lista și forma întrebărilor și răspunsurilor în conversația cu utilizatorii, lista programelor care trebuie executate în amonte etc.);
- ✖ exemple de utilizare.

■ **Exploatarea curentă.** După punerea la punct, programele intră în utilizare, fiind urmărite în vederea depistării unor erori care nu au putut fi sesizate anterior. Această fază asigură menținerea în funcțiune și dezvoltarea programelor, proces care se poate continua pe tot parcursul utilizării lor.



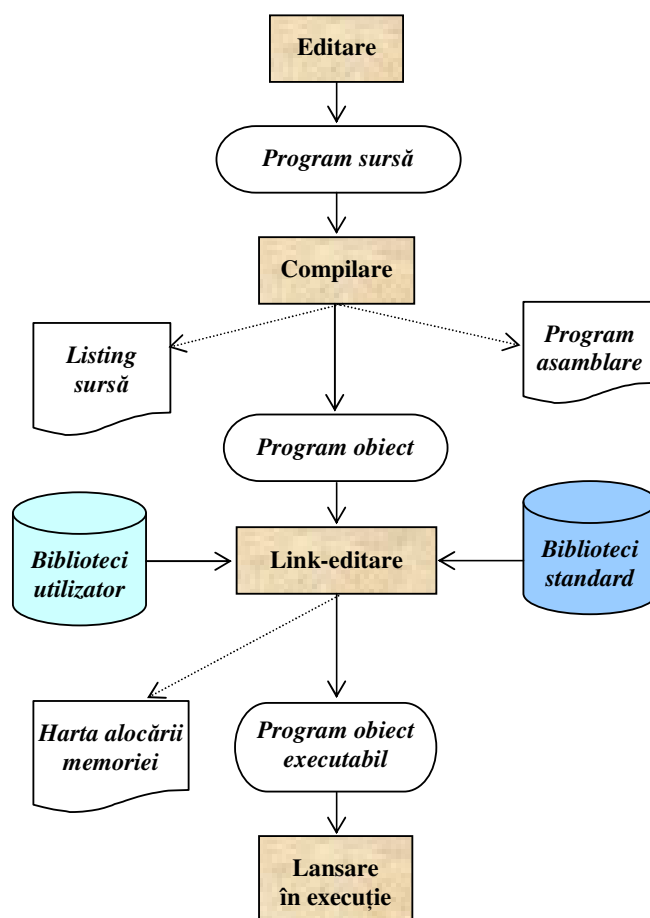
### Test de autoevaluare

1. Într-o abordare schematică, enumerați etapele de realizare a sistemelor informatice.

## 3.2. Fazele dezvoltării programelor

Utilizarea calculatoarelor necesită elaborarea programelor pe baza cărora se obțin rezultate dorite, prin aplicarea unui algoritm asupra datelor de intrare. Un *program* reprezintă o mulțime ordonată de instrucțiuni, asociată unui algoritm de rezolvare a problemei, care comandă operațiile de prelucrare a datelor. *Instrucțiunea* reprezintă exprimarea într-o formă riguroasă - conform regulilor de sintaxă ale unui limbaj artificial (limbaj de programare) a unei operații și precizează funcția și adresele operanzilor. Prin *limbaj de programare* se înțelege o mulțime de

cuvinte (dicționar) împreună cu regulile de utilizare a lor (gramatică). Regulile de utilizare sunt de sintaxă (formă) și de semantică (sens).



**Fig. 3.1.** Etapele dezvoltării unui program

Unitatea centrală a unui calculator recunoaște un singur limbaj, numit limbaj mașină (sau limbaj cod obiect executabil). Scrierea programelor într-un astfel de limbaj este greoaie și ineficientă. El este apropiat de mașină, necesită coduri numerice pentru operații și adrese absolute pentru operanzi.

În practică, utilizatorii folosesc *limbaje evolute* (universale, algoritmice) apropiate de factorul uman, cum ar fi FORTRAN, COBOL, BASIC, C, PASCAL etc. Programele scrise într-un astfel de limbaj se numesc *programe sursă*, care din punct de vedere al utilizatorului sunt fișiere de tip text, create prin editare specifică limbajului sau de utilizare generală. Transformarea programelor sursă în programe obiect executabile se realizează, de obicei, în mai multe faze (figura 3.1).

*Compilarea* este operația de traducere a programului sursă în program cod obiect. Ea este realizată de componenta software numită compilator. Lucrul cu un anumit limbaj evoluat presupune existența compilatorului pentru acel limbaj. În funcție de posibilitățile compilatorului, în urma compilării se mai pot obține fișierul cu listingul programului sursă și fișierul cu programul sursă tradus în limbaj de asamblare. Unele limbaje, printre care și C, impun ca programele să fie supuse unei prelucrări anterioare, numită *preprocesare*.

*Editarea de legături* (link-editarea) este operația de transformare a programului cod obiect, rezultat din compilare în program cod obiect executabil. Ea este realizată de componenta software numită link-editor. Link-editorul încorporează în programul obiect atât module

executabile din biblioteca specifică limbajului, invocate (apelate) implicit sau explicit de programul sursă al utilizatorului, cât și module executabile din bibliotecile realizate de utilizator. Programul obiect executabil se lansează în execuție cu comenzi ale sistemului de operare.



### Test de autoevaluare

2. Fazele dezvoltării programelor sunt: 1) editare; 2) verificare sintaxă; 3) compilare; 4) editare legături; 5) lansare în execuție; 6) testare. **a) toate; b) 1,2,3,4 și 5; c) 1,3,4,5 și 6; d) 1,2,3 și 4; e) 1,3,4 și 5.**

### Răspunsuri și comentarii la testele de autoevaluare

1. Într-o abordare schematică, etapele de realizare a sistemelor informatice sunt următoarele:

- ✘ *studiul și analiza sistemului informațional actual;*
- ✘ *proiectarea de ansamblu;*
- ✘ *proiectarea de detaliu;*
- ✘ *elaborarea programelor;*
- ✘ *implementarea și exploatarea sistemului.*

2:e).

### Rezumat

În cadrul acestei unități de învățare au fost studiate următoarele aspecte în ceea ce privește datele și structurile de date:

- ➡ caracteristici generale ale PPAD;
- ➡ organizarea procesului de rezolvare a PPAD;
- ➡ fazele dezvoltării programelor.

După încheierea acestei unități de învățare, studenții au cunoștințe și abilități de organizare a procesului de rezolvare a problemelor și a etapelor de dezvoltare a programelor.

### Bibliografia unității de învățare

1. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, M. Mircea, L. Bătăgan, C. Silvestru, Bazele programării calculatoarelor. Teorie și aplicații în C, Ed. ASE, București, 2006, ISBN 973-594-591-6

## 4. Caracteristicile limbajului C

### Cuprins

Obiectivele unității de învățare 4

4.1 Elementele de bază ale limbajului C

4.2 Tipurile de date în C

4.3 Expresii

4.4. Realizarea structurilor fundamentale de control

Răspunsuri și comentarii la testele de autoevaluare

Bibliografia unității de învățare

### Obiectivele unității de învățare 4

După studiul acestei unitati de învățare, studenții vor avea cunoștințe teoretice și abilități practice despre:

- elementele de bază ale limbajului C;
- tipuri de date în C;
- expresii;
- realizarea structurilor fundamentale de control.



*Durata medie a unității de studiu individual - 10 ore*

### 4.1 Elementele de bază ale limbajului C

Principalele construcții sintactice ale limbajului C, într-o ordine relativă a procesului de agregare sunt:

- *identificatorii*, care sunt denumirile (adresele simbolice) pe baza cărora sunt referite în program datele, tipurile, funcțiile, fișierele etc.;
- *comentariile*, care sunt texte prezente în programul sursă, dar ignorate de compilator;
- *expresiile*, care sunt construcții formate din operanzi (date numerice, logice, de tip caracter etc.) și operatori (aritmetici, relaționali, logici etc.) și a căror evaluare produce o valoare de un anumit tip;
- *declarațiile*, care sunt construcții pentru definirea și descrierea datelor (constante și/sau variabile, date scalare și/sau structuri de date etc.);
- *instrucțiunile*, pentru descrierea acțiunilor realizate asupra datelor;
- *funcțiile*, care pe baza unor date de intrare furnizează unul sau mai multe rezultate de ieșire;
- *programul*, reprezentând construcția sintactică de cel mai înalt nivel, care poate face obiectul prelucrării și execuției de un sistem de calcul.



❶ **Identificatorii** sunt denumiri asociate entităților referite în program. Ele sunt succesiuni de *litere* (mici și/sau mari, din alfabetul latin), *cifre zecimale* (de la 0 la 9) și *\_* (liniuța de subliniere). Primul caracter trebuie să fie o literă sau linieuța de subliniere. Nu se recomandă însă ca primul caracter să fie „\_” pentru a nu se face confuzii nedorite cu identificatorii rezervați folosiți de diferite compilatoare. Literele mari **nu** sunt identice cu cele mici (C face parte din categoria limbajelor *case sensitive*).

**Example:**

a, x, X, abc, Abc, x1, x\_1, alfa, pret\_unitar, student, \_, \_a, \_\_\_\_.  
CodProdus nu are aceeași semnificație cu codprodus.

La scrierea identificatorilor se utilizează frecvent literele mici, dar pentru denumirile compuse se recomandă ca fiecare cuvânt să înceapă cu literă mare, pentru creșterea lizibilității programului sursă.

**Example:**

PretUnitar, MatrUnitate, SumaElemVector, MedAritmPond.

Lungimea unui identificator variază de la un singur caracter până la oricâte, dar se iau în considerare numai primele 32. Dacă doi identificatori au primele 32 de caractere identice, atunci ei vor fi considerați identici de compilator. Unele compilatoare permit modificarea acestei limite de lungime.

Anumiți identificatori sunt predefiniți în limbaj, au destinație impusă și nu pot fi utilizați în alte moduri sau scop decât cele pentru care au fost definiți. Aceștia sunt numiți *cuvinte rezervate* (sau cuvinte cheie) și vor fi introduse pe măsură ce sunt prezentate construcțiile sintactice ale limbajului.

**Example:**

for, do, if, else, char, long, float, while, return etc. (din *K&R C*);  
const, enum, signed, void etc. (din *ANSI C*);  
\_AL, \_AX, \_DX, \_BX, \_FLAGS (din *Turbo C*).

❶ **Comentariile** sunt secvențe de text compuse din orice caractere admise în setul limbajului, care au format liber și care nu se compilează. În general, comentariile sunt necesare pentru explicații suplimentare ale programatorului, în vederea înțelegerii ulterioare a logicii sale. Comentariile sunt delimitate de perechile de caractere */\** și *\*/* și se pot întinde pe mai multe linii sursă.

Orice caracter aflat între delimitatori este ignorat de compilator. Nu se admit comentarii imbricate (incluse unul în altul). La compilatoarele C++ a fost adăugată posibilitatea de a scrie comentarii pe o singură linie sursă. Începutul unui astfel de comentariu este marcat de perechea *//* iar sfârșitul său este marcat de sfârșitul liniei (nu există marcator special). Orice caracter aflat după *//* este ignorat, până la trecerea pe o nouă linie sursă.

❶ **Instrucțiunile** sunt construcții sintactice ale limbajului, terminate de caracterul *;* (punct și virgulă). Ele indică operațiile (acțiunile) care se aplică datelor în vederea obținerii rezultatelor scontate prin algoritm. Instrucțiunile pot fi clasificate în: simple, structurate, compuse.

❶ **Funcțiile** sunt entități (subprograme) care pot fi proiectate și realizate independent, dar care nu se execută decât împreună cu o altă entitate de program, numită apelatoare. Conceptul a fost definit de realizatorii limbajului Fortran și s-a impus în practica programării din



următoarele motive: evitarea scrierii repetate în program a unor secvențe de instrucțiuni aplicate de mai multe ori pe seturi diferite de date; creșterea eficienței activității de programare prin crearea unor biblioteci de subprograme des utilizate, care pentru fiecare din aplicațiile concrete doar se apelează, fără să fie proiectate și realizate de fiecare dată; necesitatea modularizării problemelor cu grad înalt de complexitate.

Funcția are un nume și un tip. Ea se definește printr-un antet și un corp (bloc) astfel:

```
antet
{
    corp
}
```

Antetul are forma:

```
tip nume(lista_parametrilor_formali)
```

unde:

- *tip* poate fi un tip simplu de dată. Dacă lipsește, este considerat tipul implicit (*int* pentru unele compilatoare, *void* pentru altele);
- *nume* este un identificator care reprezintă numele funcției;
- *lista-parametrilor-formali* conține parametrii formali sub forma:

```
[tip1 identificator1[,tip2 identificator[,tip3 identificator ...]]]
```

Parametrii sunt separați prin virgulă. La limită, lista poate fi vidă. Pentru fiecare parametru trebuie specificat tipul, chiar dacă mai mulți parametri sînt de același tip.

Funcțiile returnează o valoare „prin nume”, dar pot returna valori și prin parametri. Unele funcții (de exemplu, cele care citesc sau scriu date) execută „servicii” și apoi returnează, prin nume, valori care pot sau nu să fie utilizate în apelator. Funcțiilor care nu trebuie să întoarcă valori prin nume li se poate asocia tipul *void* (care, la unele compilatoare, este implicit).

Corpul este o instrucțiune compusă: conține declarațiile locale și instrucțiunile executabile care implementează algoritmul. Corpul funcției se execută pînă la executarea ultimei instrucțiuni sau pînă la executarea instrucțiunii *return*. Forma ei generală este:

```
return(expresie); sau
return expresie; sau
return;
```

Prima și a doua formă sînt folosite în cazul funcțiilor care returnează o valoare prin numele lor. Prin executarea acestei instrucțiuni se evaluează expresia, valoarea sa este atribuită funcției și se încheie execuția funcției. A treia formă este folosită în cazul funcțiilor care nu returnează nici o valoare prin numele lor (poate chiar să lipsească). Dacă este prezentă, efectul ei este încheierea execuției funcției. Tipul expresiei din instrucțiunea *return* trebuie să coincidă cu tipul funcției.

Apelul funcțiilor se face prin referirea lor ca operanzi în expresii, sub forma:

```
nume(lista_parametrilor_realii)
```

La apel, parametrii reali se pun în corespondență cu cei formali, de la stînga la dreapta. Pentru o corectă utilizare a datelor, tipurile parametrilor reali trebuie să fie aceleași ca ale celor formali corespunzători.

**Programul** este construcția sintactică de cel mai înalt nivel. El codifică algoritmul, fiind constituit din declarații și instrucțiuni executabile. Dat fiind faptul că limbajul C este puternic orientat spre funcții, programul principal este el însuși o funcție (care poate avea parametri și poate întoarce un rezultat de tip întreg): *main* (cuvînt rezervat). La modul general, un program este o înșiruire de funcții și declarații, printre care trebuie să existe o funcție denumită *main*,

lansată în execuție la rularea programului. Pentru funcția *main* sunt permise mai multe forme ale antetului:

```
main()
int main()
void main()
main(void)
void main(void)
int main(void)
```

Orice program este format dintr-o *parte de declarații* (formată din instrucțiuni neexecutabile, cu rol doar în faza de compilare) și o *parte executabilă* (formată din instrucțiuni executabile). Unele limbaje impun o separare clară a acestora (de exemplu Pascal, COBOL, FORTRAN), în care partea de declarații precede partea executabilă. Alte limbaje (de exemplu C, C++) sunt mai flexibile, permițând ca declarațiile să alterneze cu instrucțiunile executabile, păstrând însă regula că *orice entitate referită trebuie definită anterior*. Pentru a evita confuziile se recomandă ca declarațiile să fie reunite la începutul blocului de instrucțiuni pentru care sunt vizibile.

● **Directivele de preprocesare.** Înaintea compilării, în C se desfășoară etapa de *preprocesare*, în cadrul căreia se efectuează substituirii asupra textului sursă scris de programator. Prin preprocesare se rezolvă cerințele compilării condiționate, se asigură inserarea unor fișiere în textul sursă și se expandează macrodefinițiile. Preprocesarea este controlată prin *directive*, a căror sintaxă (în format BNF) este:

$$\langle \text{directiva} \rangle ::= \# \langle \text{cuvant\_rezervat} \rangle [ \langle \text{parametri} \rangle ]$$

⊞ Directiva `#include` este folosită pentru includerea de fișiere cu text sursă într-un program și are următoarea sintaxă:

```
#include<specificator_de_fisier>
```

sau

```
#include"specificator_de_fisier"
```

Prima formă caută fișierul specificat între fișierele standard ale limbajului, folosindu-se calea descrisă în mediul de programare. A doua formă caută fișierul specificat în calea curentă (de obicei este folosită pentru a include fișiere scrise de utilizator). Dacă fișierul căutat nu este găsit se produce o eroare de compilare. Dacă fișierul este găsit, conținutul lui este inserat în program, în locul directivei care l-a invocat, aceasta fiind ștearsă (la limită poate fi considerată o substituie de text).

Pentru ca textul inclus să fie vizibil din tot programul, se recomandă ca directivele `#include` să fie scrise la începutul programului. Un text inclus poate să conțină la rândul lui directive care determină noi includeri de text. Textul inclus va fi compilat ca parte componentă a programului.

Pentru a putea utiliza funcțiile care realizează operațiile de intrare/ieșire trebuie inclus fișierul `stdio.h`.

❗ **Exemplu:** `#include<stdio.h>`

⊞ Directiva `#define` este folosită pentru a substitui unele secvențe de caractere cu altele și are forma generală:

```
#define sir1 sir2
```

unde atât *sir1* cât și *sir2* sunt șiruri de caractere. La preprocesare, se șterge directiva din program și se înlocuiește secvența de caractere *sir1* cu secvența de caractere *sir2* peste tot unde apare în programul sursă. Secvența *sir1* este denumită *nume* iar *sir2* este denumită *descriere*. Nu se face înlocuirea dacă *sir1* apare în interiorul unui literal de tip șir de caractere sau în interiorul unui comentariu. *sir2* poate să fie descris pe mai multe rânduri, dacă la sfârșitul fiecărui rând (în afară de ultimul) se scrie caracterul \ (backslash). Rolul acestuia va fi discutat în alt capitol. *sir2* poate să conțină șiruri de caractere pentru care au fost definite anterior. Acestea vor fi substituite conform definițiilor lor.

✚ Substituirea poate fi dezactivată din punctul în care apare directiva `#undef` până la sfârșitul programului sau până la redefinirea lui *sir1*. Directiva are forma generală:

`#undef sir1`

Între cele mai frecvente utilizări ale directivei `#define` sunt definirea de *constante simbolice* (literali cărora li se asociază identificatori) și definirea de *macrodefiniții* (necesare simplificării scrierii).

Din punct de vedere al textului sursă, un macro se folosește la fel ca orice funcție: se apelează prin numele simbolic urmat de lista parametrilor reali. Din punct de vedere al compilării există o diferență fundamentală: macro-urile sunt tratate la *preprocesare*. Preprocesorul șterge din textul sursă apelul macro-ului și îl înlocuiește chiar cu secvența de definire. Parametrii formali ai macro-ului sunt înlocuiți cu parametrii reali, prin substituție de text (corespondența se face conform regulilor de punere în corespondență a parametrilor reali cu cei formali: unu-la-unu, de la stânga la dreapta).



### Test de autoevaluare

1. Specificați cum va arăta secvența de cod următoare, după preprocesare:

```
#define N 10
#define M 10
#define MAX (M+N)
#define DIM(a,b) (a)*(b)
char v[N], v1[10+DIM(5+M, 6)];
char v1[10*MAX];
char m[M][N];
```

## 4.2 Tipurile de date în C

Limbajul oferă posibilități multiple în declararea și utilizarea tipurilor de date. În general, tipurile de date se clasifică în *stactice* și *dinamice*, împărțite la rândul lor în *simple* și *structurate*. Tipurile de date admise de limbajul C sunt prezentate în tabelul 4.1.

**Tabelul 4.1.** Clasificarea tipurilor de date în C

După modul de alocare a memoriei	După numărul de valori memorate	Tipuri existente
Stactice	Simple	Întregi
		Reale
		Caracter

	Structurate	Masiv
		Articol
		Fișier
Dinamice	Simple	Pointer
		Referință

#### 4.2.1. Tipurile simple de date

În limbajul C există o serie de tipuri simple de date predefinite (tabelul 4.2), a căror lungime poate să difere de la un sistem de calcul la altul și de la o implementare la alta (standardul C este mai elastic decât altele). De exemplu, pentru tipul *char* este definită lungimea 1 sau cel mult lungimea tipului *int*.

\* Tabelul 4.2. Tipuri simple de date în C

Grup a de dată	Tipul	Lungime (octeți)	Domeniu de valori	Mod de reprezentare
Întreg	unsigned char	1	$0..255 (0..2^8-1)$	Codul ASCII al caracterului.
	[signed] char	1	$-128..127 (-2^7..2^7-1)$	Poate fi prelucrat ca un caracter sau ca un întreg cu/fără semn.
	unsigned [int]	2	0..65535	Virgulă fixă aritmetică
	[signed] [int]	2	-32768..32767	Virgulă fixă algebrică
	unsigned long	4	$0..2^{32}-1$	Virgulă fixă aritmetică
	[signed] long [int]	4	$-2^{31}..2^{31}-1$	Virgulă fixă algebrică
Real	float	4	$3.4 \cdot 10^{-38}..3.4 \cdot 10^{38}$	Virgulă mobilă simplă precizie
	double	8	$1.7 \cdot 10^{-308}..1.7 \cdot 10^{308}$	Virgulă mobilă dublă precizie
	long double	10	$3.4 \cdot 10^{-4932}..3.4 \cdot 10^{4932}$	Virgulă mobilă extra precizie

#### Variabilele

Datele variabile își modifică valoarea pe parcursul execuției programului. De aceea, ele se asociază unor zone de memorie. Identificatorii acestora se declară și apoi se referă în operații. Operațiile afectează conținutul zonelor de memorie, care este interpretat ca fiind de un anumit tip. Declararea variabilelor se realizează prin listă de identificatori (separați prin virgulă) pentru care se specifică tipul.

tip lista\_variabile;

Declararea se poate face oriunde în program, cu următoarea condiție: variabilele trebuie declarate înainte de a fi folosite (referite). Domeniul de valabilitate este limitat la blocul în care s-a făcut declarația.

#### Example:

unsigned x,y,z;

```
float a,b,c;
char k;
```

Variabilele pot fi inițializate la momentul compilării, dacă se utilizează următoarea declarație:

```
tip nume_variabila=valoare;
```



### Example:

```
int dim_vector=100;
dim_vector=50; /*nu este eroare*/
```

**Definirea de noi tipuri de date.** Utilizatorul poate defini noi tipuri de date sau poate atribui un alt nume unui tip predefinit sau definit anterior. În acest scop se folosește cuvântul rezervat `typedef`, astfel:

```
typedef descriere_tip nume_utilizator;
```



### Example:

```
typedef int INTREG;
typedef float REAL;
```

Particularitățile unor tipuri simple de date. **Tipul caracter memorează caractere ale codului ASCII, reprezentate pe un octet. Variabilele de tip caracter pot fi utilizate și ca valori numerice (modul de utilizare se alege automat, în funcție de expresia din care face parte operandul respectiv).**

Valoarea numerică folosită depinde de modul de declarare a caracterului: cu sau fără semn. Pentru caracterele cu coduri mai mici decât 128 nu se sesizează nici o diferență (se obține aceeași valoare și pentru interpretarea ca virgulă fixă aritmetică și pentru interpretarea ca virgulă fixă algebrică). Pentru caracterele cu coduri mai mari de 127, valoarea obținută este diferită.



### Example:

Declararea și utilizarea variabilelor de tip caracter.

```
unsigned char a,b;
.....
a=100; /* corect */
b='Q'; /* corect */
b=81; /* corect, echivalent cu precedentul */
```

## 4.2.2. Constantele

Datele constante sunt predefinite la momentul scrierii programului iar valoarea lor este generată la momentul compilării. Datele constante pot fi *literal* (constante propriu-zise), care se autoidentifică prin valoare, sau *constante simbolice*, care sunt identificatori (denumiri) asociați altor constante (literal sau constante simbolice definite anterior). În C literalii sunt întregi, reali, caracter și șir de caractere.

**Literalii întregi** sunt reprezentați intern în virgulă fixă. Ei pot fi exprimați în bazele de numerație 10 (forma implicită), 8 (folosind prefixul *0* – zero) sau 16 (folosind prefixul *0x* sau *0X*). În funcție de mărimea lor, se asociază implicit un tip întreg (și implicit un mod de reprezentare). Se încearcă întotdeauna întâi reprezentarea pe 16 biți, conform tipului *int*; dacă nu este posibilă, atunci se folosește reprezentarea pe 32 de biți, conform tipului *long*.

Dacă se dorește forțarea reprezentării pe 32 de biți (pentru o valoare din domeniul tipului *int*), se adaugă sufixul *l* sau *L*. Pentru a forța tipul *fără semn* (*unsigned int* sau *unsigned long*) se folosește sufixul *u* sau *U*. Cele două sufixe se pot folosi împreună, în orice ordine.

● **Literalii reali** sunt reprezentați intern în virgulă mobilă. Ei se pot exprima sub formă matematică ( $\pm$ întreg.fracție) sau științifică ( $\pm$ întreg.fracțieE $\pm$ exponent). Semnul + poate lipsi (este implicit), iar e este echivalent, ca valoare, cu E. Din exprimare poate să lipsească fie partea fracționară, fie partea întreagă, fie partea întreagă și exponentul (inclusiv litera e).

Literalii reali se reprezintă intern pe 64 de biți (tipul *double*). Pentru a forța reprezentarea în simple precizie (tipul *float*) se adaugă sufixul *f* sau *F*. Pentru a forța reprezentarea pe 80 de biți (tipul *long double*) se folosește sufixul *l* sau *L*. Cele două sufixe nu se pot folosi împreună.

● Literalii caracter *se reprezintă intern prin codul ASCII al caracterului respectiv, pe un octet. Exprimarea externă depinde de caracterul respectiv. Literalii de tip caracter pot participa în expresii cu valoarea lor numerică, așa cum s-a arătat anterior.*

Exprimarea externă a unui caracter direct imprimabil (existent pe tastatură, cu coduri ASCII cuprinse între 32 și 127) se face prin caracterul respectiv inclus între apostrofuri. Excepție fac caracterele cu semnificație specială în C: ' (apostrof), " (ghilimele) și \ (backslash), care se exprimă printr-o succesiune de două caractere, fiind precedate de caracterul \.

➤ **Exemple:** 'B', 'b', '7', ' ' (spațiu), '\*', '\ (caracterul backslash, cod ASCII 92), '\'' (caracterul apostrof, cod ASCII 39), '\"' (caracterul ghilimele, cod ASCII 34).

Reprezentarea folosind caracterul *backslash* este numită *secvență escape*. Secvențele escape sunt folosite și pentru a reprezenta caracterele de control (coduri ASCII între 0 și 31). Pentru a reprezenta caracterele grafice ale codului ASCII extins (coduri între 128 și 255) se pot folosi numai secvențele escape construite astfel: '\ddd', unde d este o cifră din sistemul de numerație octal (0÷7). Construcția ddd este considerată implicit ca fiind codul ASCII al caracterului, reprezentat în baza 8. Nu este nevoie ca ea să fie precedată de un zero nesemnificativ. Această construcție poate fi folosită pentru a reprezenta orice caracter al setului ASCII.

➤ **Exemple:** '\a' și '\7' reprezintă caracterul BEL, '\b' și '\10' reprezintă caracterul BS, '\"' și '\42' reprezintă caracterul ghilimele, '\377' reprezintă caracterul cu codul ASCII 255.

La inițializarea unei variabile de tip caracter se poate folosi oricare din variantele de reprezentare descrise anterior sau orice valoare numerică (întreagă sau reală). În acest ultim caz, din reprezentarea internă a literalului numeric se iau în considerare primii 8 biți care sunt interpretați ca un cod ASCII, obținându-se valoarea care se atribuie.

● Literalii de tip șir de caractere. *Un literal de tip șir de caractere este un șir de zero sau mai multe caractere, delimitate prin ghilimele (ghilimelele nu fac parte din șir). Pentru a reprezenta diferite caractere, în interiorul șirului se pot folosi secvențe escape. Un literal de tip șir de caractere poate fi scris pe mai multe rânduri. Pentru a semnala că un literal continuă pe rândul următor se scrie caracterul \ la sfârșitul rândului curent.*

Un șir de caractere este reprezentat intern prin codurile ASCII ale caracterelor, câte unul pe fiecare octet, la sfârșit adăugându-se caracterul *nul* (cod ASCII 0 – '\0'). Caracterul *nul* nu poate să facă parte dintr-un șir, el având rolul de terminator de șir. În reprezentarea internă, un șir de caractere ocupă cu un octet mai mult decât numărul de caractere din componența sa.

➤ **Exemple:** literalul  
 "Acesta este un literal \ "șir de caractere" \  
 scris pe doua randuri."

reprezintă șirul Acesta este un literal "șir de caractere" scris pe doua randuri.

- " " - șir de lungime 1 (caracterul spațiu)
- "" - șirul de lungime 0 (șirul vid)

• **Constantele simbolice** sunt literalii cărora li se asociază identificatori. În limbajul C constantele simbolice se construiesc folosind directiva:

```
#define nume_constanta valoare
```

Utilizarea constantelor simbolice în cadrul programelor are următoarele avantaje:

- literalii primesc nume sugestive, mai ușor de reținut și de utilizat, mai ales dacă sunt formați dintr-un număr mare de cifre;
- pentru rularea succesivă a programului cu valori diferite ale unui literal utilizat des este suficientă modificarea valorii constantei simbolice în definire (referirile nefiind modificate).



#### Exemple:

```
#define pi 3.141592653589
#define raspuns "D"
```

• **Constantele obiect** sunt variabile inițializate la declarare, pentru care se rezervă memorie, dar conținutul lor nu poate fi modificat pe parcursul execuției programului. Ele se declară folosind modificatorul *const*:

```
const tip nume_constanta=valoare;
```



#### Exemplu:

```
const int dim_vector=10;
```

Dacă se încearcă o atribuire (de exemplu `dim_vector=7`) se generează eroare.

### 4.2.3. Tipurile structurate de date

Tipurile structurate au la bază mulțimi (colecții) de date de tipuri simple sau structurate, constituite după anumite reguli ierarhice și de dependență bine stabilite. Nici o dată structurată nu poate ocupa în memorie mai mult de 65520 de octeți (lungimea unui segment de memorie).

#### • Tipul masiv

Tipul *masiv* este structurat și desemnează o mulțime finită de elemente omogene constituită ca un tablou cu una, două sau mai multe dimensiuni. Mulțimea are un singur identificator și oferă posibilitatea referirii elementelor în acces direct prin poziție, determinată printr-un număr de expresii indiciale corespunzând dimensiunilor masivului. Masivul se declară folosind o construcție de forma:

```
tip nume[dim1][dim2]...[dimn];
```

unde *tip* este tipul comun elementelor masivului, iar *dim1*, *dim2*, ..., *dimn* sunt expresii constante (evaluate la compilare), care indică numărul de elemente de pe fiecare dimensiune. Se acceptă oricâte dimensiuni, cu condiția ca structura în ansamblul ei să nu depășească zona de memorie maximă permisă pentru structurile de date.



#### Exemple:

```
int x[10]; /* vector cu 10 componente intregi */
float a[10][20]; /* matrice cu 10 linii si 20 coloane
de elemente reale */
```

La declararea masivelor se poate face și inițializarea lexicografică a acestora, astfel (pentru masiv bidimensional):

```
tip nume[dim1][dim2]=
```

```
{{lista_const1}{lista_const2}...{lista_constn}};
```

**Exemple:**

```
int b[10]={2,3,0,5,6,7,3,6,8,5};
/* vectorul contine valorile 2 3 0 5 6 7 3 6 8 5 */
float a[5][3]={{1,2,3},{1,2,3},{1,2,3}};
/* matricea contine valorile 1 2 3 1 2 3 1 2 3 0 0 0 0 0 0 */
```

Masivul poate fi referit *global* prin numele său. Elementele unui masiv se referă *direct*, prin numele masivului urmat de indicele (indicii) corespunzători. Indicii sunt expresii incluse între paranteze drepte (se mai numesc și *variabile indexate*): [indice]. Pentru fiecare dimensiune a masivului trebuie să existe câte o referire de indice. Indicele de pe fiecare dimensiune are valoarea minimă 0 și valoarea maximă egală cu dimensiunea din declarare minus o unitate. Compilatorul nu verifică corectitudinea indicilor (în sensul de depășire a dimensiunilor masivului la referirea elementelor).

### • Tipul articol

Articolul este o structură de date eterogenă, cu acces direct la elementele sale, între care există o relație de ordine ierarhică. Articolul poate fi reprezentat sub formă de arbore, ale cărui noduri sunt asociate componentelor structurii. Componentele de pe ultimul nivel sunt scalare și se numesc *date elementare* sau *câmpuri*. Datele de pe celelalte niveluri, denumite *date de grup*, se constituie prin agregarea datelor de pe nivelurile inferioare. Data de grup de cel mai înalt nivel (rădăcina arborelui) corespunde articolului în ansamblu. Conceptual, datele de grup de pe diverse niveluri au aceleași proprietăți ca și articolul, ceea ce permite ca această structură să fie construită recursiv, prin descompunerea în structuri cu aceleași proprietăți (figura 4.1).

Declararea împreună a tipului articol și a variabilelor de acest tip se realizează conform sintaxei:

```
struct tip_articol{lista_campuri} var1,var2,...,varn;
```

unde *tip\_articol* este identificatorul asociat tipului articol, iar *var1*, *var2*,..., *varn* sunt identificatorii asociați variabilelor de tipul articol declarat.

Parametrii declarației pot lipsi (dar nu toți deodată). Dacă lipsesc parametrii *var1*, *var2*,..., *varn*, atunci *tip\_articol* trebuie să fie prezent, fiind numai o declarare explicită de tip nou, utilizabil ulterior la alte declarații. Dacă lipsește *tip\_articol*, atunci trebuie să fie prezentă lista de variabile (nevidă), caz în care este vorba de o declarare de variabile de tip articol, fără însă a declara și un tip utilizator nou. În continuare, *tip\_articol* este un tip nou de date, iar *var1*, *var2*,..., *varn* sunt variabile de tipul *tip\_articol*. Variabilele pot fi declarate și ca masive, ale căror elemente sunt de tip articol: *var1*[dim1][dim2]...[dimn].



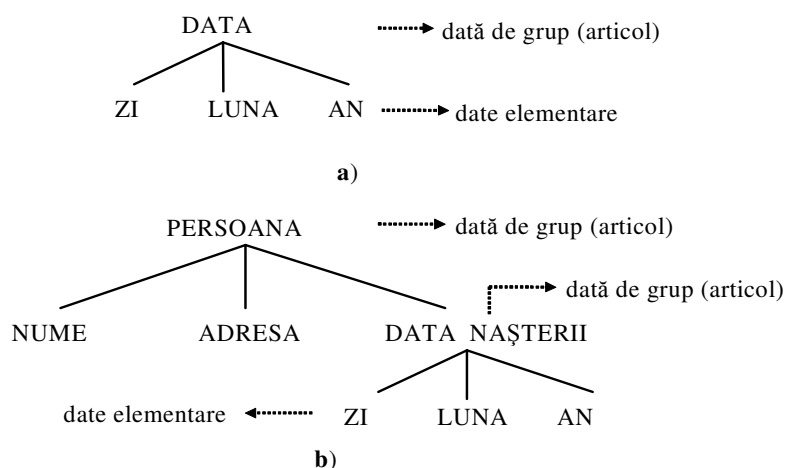


Fig. 4.1. Exemple de structuri de articole

O variabilă de tip articol poate fi declarată și ulterior definirii tipului:

```
struct tip_articol var1;
```

Descrierea constituie o definire implicită de un nou tip de dată. Este posibilă definirea explicită a unui nou tip de dată, adăugând cuvântul rezervat *typedef* în fața declarării (în acest caz nu mai pot fi declarate simultan și variabile).

*Lista\_campuri* este o înșiruire de declarații de câmpuri separate prin punct și virgulă, asemănătoare declarațiilor de variabile, de forma *tip\_camp nume\_camp*. Câmpurile unei structuri pot fi variabile simple, masive sau alte articole. Lista câmpurilor nu poate fi vidă.

**Exemplu:** pentru exemplele din figura 6.1, declararea poate fi realizată prin definire recursivă, astfel:

```
struct tip_data {
    unsigned zi;
    char luna[3];
    int an; };
struct persoana {
    char nume[30];
    char adresa[50];
    struct tip_data data_nasterii;
} angajat;
```

Dacă nu ar fi existat declarația tipului articol *tip\_data*, atunci tipul *persoana* putea fi scris astfel:

```
struct persoana {
    char nume[30];
    char adresa[50];
    struct {
        unsigned zi;
        char luna[3];
        int an;
    } data_nasterii;
} angajat;
```

Variabilele de tip articol se reprezintă intern ca succesiuni de câmpuri elementare, cu reprezentarea internă și lungimea fizică specifice tipurilor lor. Lungimea zonei de memorie rezervată pentru variabila de tip articol rezultă din însumarea lungimilor câmpurilor. Aceasta nu poate depăși 65520 octeți (ca orice variabilă de tip structurat). Pentru structura unui articol își dovedește utilitatea operatorul *sizeof*, care asigură determinarea lungimii zonei de memorie asociate unei variabile sau unui tip de date.

**Exemplu:**

Considerând declarațiile anterioare, expresia `sizeof(data_nasterii)` are valoarea 8, iar `sizeof(angajat)` are valoarea 90.

Datele de tip *articol* pot fi referite în două moduri: global sau pe componente. *Referirea globală* este permisă numai în operația de atribuire, cu condiția ca ambele variabile (sursă și destinație) să fie articole de același tip.

*Referirea pe componente* (prin numele lor) este o reflectare a faptului că articolul este o structură cu acces direct. Referirea unor componente de tip articol din structura altui articol este posibilă numai în operația de atribuire, în condițiile precizate anterior la referirea globală. În cele ce urmează se are în vedere numai referirea componentelor de tip dată elementară, situate pe ultimul nivel al structurii.

*Referirea câmpurilor* unei structuri se face prin *calificare*, folosind operatorul `.` (punct). În referirea prin calificare, asigurarea identificării unice a câmpurilor se realizează prin asocierea numelui acestora cu numele articolului care le conține. Construcția rămâne la această formă în cazul în care structura are numai două niveluri: articolul și câmpurile elementare ale acestuia.

În articolele cu structură recursivă se realizează calificarea progresivă cu articolele de pe nivelurile superioare, primul calificator fiind numele articolului rădăcină. În lanțul de calificări, numele articolului rădăcină este *nume de variabilă*, celelalte fiind nume de câmpuri ale articolului. Dacă anumite componente sunt structuri de date de alte tipuri (de exemplu masive sau șiruri de caractere), în referirea elementelor lor se aplică, pe lângă calificare, regulile specifice acestor structuri.



### Exemplu:

Referirea prin calificare a câmpurilor articolului `angajat` de tipul `persoana` (vezi exemplele anterioare) se realizează astfel:

```
angajat.nume;
angajat.adresa;
angajat.data_nasterii.zi;
angajat.data_nasterii.luna;
angajat.data_nasterii.an
```

În aceste referiri, `angajat` este identificatorul variabilei articol, celelalte elemente sunt identificatori de câmpuri. Construcțiile `angajat.nume` și `angajat.adresa` corespund referirii globale a câmpurilor respective, care sunt șiruri de caractere. Pentru a referi, de exemplu, primul caracter din șir, se scrie: `angajat.nume[0]`.



### Exemplu:

```
#include <stdio.h>
void main()
{
    struct persoana {
        char nume[40];
        char adresa[30];
        struct {
            int zi, luna, an; } datan;
    };
    //Inițializarea articolului din exemplul 1:
    struct persoana p={"Popescu Ion", "Bucuresti, Magheru 14", 2, 4, 1960};
    //sau cu evidentierea structurii data nasterii:
    struct persoana p1={"Popescu Ion", "Bucuresti, Magheru 14", {2, 4, 1960}};
    printf("\n%i",p1.datan.an);
}
```

În activitatea de programare pot fi întâlnite aplicații care reclamă utilizarea articolelor cu structură variabilă. La inițializarea câmpurilor unui astfel de articol, constanta de tip articol se asociază unei singure structuri, deoarece zona de memorie rezervată pentru articol este unică. Pentru acest tip de articol, limbajul pune la dispoziția utilizatorilor tipul predefinit *reuniune* (***union***), care se comportă ca și tipul *struct* cu o singură diferență: la un moment dat al execuției

programului, în zona de memorie rezervată articolului nu este memorat decât unul dintre câmpurile acestuia.

Declararea tipului reuniune se realizează astfel:

```
union nume_tip { tip_camp1 camp1;
                  tip_camp2 camp2;
                  .....
                  tip_campn campn;};
```

Lungimea zonei de memorie rezervate pentru o variabilă de tip reuniune va fi egală cu maximul dintre lungimile câmpurilor componente. Gestiunea conținutului respectivei zone de memorie va trebui realizată de către programator.

### ● Lucrul cu șiruri de caractere

Șirurile de caractere se reprezintă convențional ca masive unidimensionale (vectori) cu elemente de tipul *char*. Totuși există o diferență semnificativă între un vector de numere și un vector de caractere. Vectorul de caractere (șirurile de caractere) se reprezintă intern printr-o succesiune de octeți în care sunt memorate codurile ASCII ale caracterelor șirului. Ultimul octet conține caracterul NULL (cod ASCII 0 – sau ASCIIZ), cu rol de marcator de sfârșit al șirului. Marcatorul de sfârșit de șir este gestionat automat de către sistem. Astfel, pentru memorarea unui șir de  $n$  caractere sunt necesari  $n+1$  octeți. Marcatorul de sfârșit nu face parte din șir și este tratat ca atare de funcțiile care lucrează cu șiruri de caractere.



#### Exemplu:

1. `char s[10]="Limbaul C";`

0	1	2	3	4	5	6	7	8	9	10
L	i	m	b	a	j	u	l		C	0x00

Pentru prelucrarea șirurilor de caractere, limbajul C pune la dispoziția utilizatorului o serie de funcții specifice, definite în biblioteca standard *string.h*. Limbajul C dispune și de alte funcții care lucrează cu șiruri de caractere (funcții de conversie), definite în biblioteca standard *stdlib.h*: *atof* (care necesită și includerea bibliotecii standard *math.h*), *atoi*, *atol*, *itoa*, *ltoa*. De asemenea, în biblioteca *stdio.h* există definite funcții de intrare/ieșire pentru șirurile de caractere.



### Teste de autoevaluare

2. . Se presupune un articol cu următoarea structură:

Cod magazin	Vânzări lunare			
	Luna 1	Luna 2	...	Luna 12
întreg	real	real	...	real
2	4	4	...	4

Scrieți modul de declarare a articolului și lungimea sa în număr de octeți. Exemplificați referirea câmpurilor.

3. Se presupune un articol cu următoarea structură:

nume	data nașterii	an de studiu	forma de învățământ			
			zi		id	
			bursa	valoare	loc de muncă	data angajării
char[40]	z i n a	int	char	float	char[30]	z i n ă n

Specificați cum se realizează declararea și inițializarea câmpurilor unui student la zi pentru structura articolului prezentat.

## 4.3 Expresii

Asupra datelor pot fi aplicați operatori din diverse clase, rezultând construcții sintactice numite expresii. În forma lor cea mai generală, expresiile sunt alcătuite din operanzi și operatori. Evaluarea expresiilor produce ca rezultat o valoare de un anumit tip. În metalimbajul BNF expresia poate fi definită astfel:

$$\begin{aligned} \langle \text{expresie} \rangle ::= & \langle \text{operand} \rangle | \langle \text{operator\_unar} \rangle \langle \text{expresie} \rangle | \\ & \langle \text{expresie} \rangle \langle \text{operator\_binar} \rangle \langle \text{expresie} \rangle \end{aligned}$$

Prima variantă din definiție corespunde celei mai simple forme de expresie, redusă la o variabilă sau o constantă de un anumit tip. A doua și a treia variantă, prin aplicare repetată, conduc la recursivitate în construirea expresiilor, proces evidențiat în exemplul următor:

$$\begin{array}{ccccccc} a & + & (-b) & - & c & + & 15 \\ \underbrace{\quad} & & \underbrace{\quad} & & \underbrace{\quad} & & \underbrace{\quad} \\ \text{expresie} & & \text{expresie} & & \text{expresie} & & \text{expresie} \\ \underbrace{\quad \quad \quad} & & & & & & \\ \text{expresie} & & & & & & \\ \underbrace{\quad \quad \quad \quad \quad} & & & & & & \\ \text{expresie} & & & & & & \\ \underbrace{\quad \quad \quad \quad \quad \quad \quad} & & & & & & \\ \text{expresie} & & & & & & \end{array}$$

Ordinea de aplicare a operatorilor din expresii poate fi, total sau parțial, impusă prin folosirea parantezelor. Acestea sunt evaluate cu prioritate, iar dacă sunt mai multe niveluri de paranteze, evaluarea se realizează din interior spre exterior. În absența parantezelor, respectiv în interiorul acestora, evaluarea se realizează în funcție de ordinul de precedență a operatorilor.

### 4.3.1 Operanzi și operatori

Un operand poate fi una din următoarele construcții:

- o constantă simbolică;
- un literal;
- o variabilă simplă;
- numele unui masiv;
- numele unui tip de dată;
- numele unei funcții;
- referirea unui element de masiv;
- referirea unui câmp de articol;
- apelul unei funcții;
- o expresie.

Din ultima posibilitate rezultă că expresia este o construcție recursivă. Un operand are un tip și o valoare. Valoarea se determină fie la compilare fie la execuție. Nu pot fi operanzi șirurile de caractere.

Operatorii se clasifică după diferite criterii, precum numărul operanzilor asupra cărora se aplică și tipul de operație pe care o realizează. Majoritatea operatorilor sunt *unari* sau *binari*. Limbajul utilizează și un operator *ternar* (care se aplică asupra a trei operanzi). După tipul operației realizate, operatorii sunt aritmetici, logici, relaționali etc.

Într-o expresie apar, de obicei, operatori din aceeași clasă, dar pot fi și din clase diferite. Se pot scrie expresii complexe care să conțină operatori din toate clasele. La evaluarea acestor expresii se ține cont de *prioritățile* operatorilor (numite și clase de precedență), de *asociativitatea* lor și *regula conversiilor implicite*.

Atunci când un operator binar se aplică la doi operanzi de tipuri diferite, înainte de a efectua operația, cei doi operanzi sunt aduși la un tip comun. În general, operandul de tip *inferior* se convertește către tipul operandului de tip *superior*. În primul rând se convertesc operanzii de tip *char* către tipul *int*. Dacă operatorul se aplică la doi operanzi cu același tip nu se face nici o conversie, rezultatul având același tip cu cei doi operanzi. Dacă valoarea lui depășește domeniul asociat tipului de dată, rezultatul este eronat (eroarea de depășire de domeniu). Dacă operatorul se aplică la operanzi de tipuri diferite, conversia se face astfel:

1. dacă un operand este de tipul *long double*, celălalt se convertește la acest tip, iar rezultatul va fi de tip *long double*;
2. dacă un operand este de tip *double*, celălalt se convertește la acest tip, iar rezultatul va fi de tip *double*;
3. dacă un operand este de tip *float*, atunci celălalt se convertește la acest tip, iar rezultatul va fi de tip *float*;
4. dacă un operand este de tip *unsigned long*, atunci celălalt se convertește la acest tip, iar rezultatul va fi de tip *unsigned long*;
5. dacă un operand este de tip *long*, atunci celălalt se convertește la acest tip, iar rezultatul va fi de tip *long*;
6. dacă unul din operanzi este de tip *unsigned* iar celălalt de tip *int* acesta se convertește la tipul *unsigned*, iar rezultatul va fi de tip *unsigned*.

Regula se aplică pe rând fiecărui operator din cadrul unei expresii, obținând în final tipul expresiei.

#### 4.3.2. Operatorii de atribuire

Atribuirea este o expresie cu forma generală:  $v = \text{expresie}$

unde  $v$  este o variabilă simplă, un element de masiv, un câmp al unui articol sau o expresie în urma evaluării căreia se obține o adresă. Entitatea care se poate afla în stânga operatorului de atribuire se numește *left value*.

Pentru a se putea efectua atribuirea, tipul expresiei și tipul entității din stânga operatorului de atribuire trebuie să fie compatibile. Dacă sunt incompatibile se produce eroare la compilare (deoarece compilatorul nu poate insera în codul obiect apelurile pentru operațiile de conversie). Efectele atribuirii constau în:

- evaluarea expresiei din dreapta operatorului de atribuire, cu determinarea unei valori și a unui tip;

- memorarea valorii expresiei din dreapta în variabila din stânga;
- întreaga expresie de atribuire are valoarea și tipul variabilei din stânga.

În practica programării, se utilizează frecvent expresiile de forma  $v = v \text{ op } (\text{expresie})$  (de exemplu  $a = a + b - c$ ). Se observă redundanța de exprimare prin specificarea variabilei  $v$  atât în membrul stâng cât și în cel drept. Pentru eliminarea acestei redundanțe, s-au introdus operatorii combinați, cu forma generală

$$\text{op} =$$

unde  $\text{op}$  este un operator binar, aritmetic sau logic pe biți ( $/, \%, *, -, +, <<, >>, \&, \wedge, |$ ). O expresie de forma

$$v \text{ op} = \text{expresie}$$

este echivalentă cu

$$v = v \text{ op } (\text{expresie})$$


#### Example:

Expresia  $a = a + b - c$  este echivalentă cu  $a += b - c$ .

Expresia  $i *= 5$  este echivalentă cu  $i = i * 5$ .

### 4.3.3. Operatorii aritmetici

În ordinea priorității lor, operatorii aritmetici sunt:

- operatori unari  $+, -, ++, --$
- operatori binari multiplicativi  $*, /, \%$
- operatori binari aditivi  $+, -$

Operațiile efectuate de acești operatori sunt descrise în tabelul 4.3.

**Tabelul 4.3.** Operatorii aritmetici

Semnificație operație	Operator
Schimbare semn	-
Păstrare semn (nici un efect, nu este folosit)	+
Decrementare (post sau pre)	--
Incrementare (post sau pre)	++
Adunare	+
Scădere	-
Înmulțire	*
Împărțire	/
Împărțire întreagă (câtul)	/

Împărțire întreagă (restul)

%

**Observație:** Cu excepția operatorului %, care admite numai operanzi întregi, ceilalți admit toate tipurile numerice.

Operatorii ++ și -- au efect diferit, depinzând de poziția față de operand, astfel:

- a. dacă apar înaintea operandului, valoarea acestuia este modificată înainte de a fi utilizată la evaluarea expresiei din care face parte: ++var (preincrementare) sau --var (predecrementare);
- b. dacă apar după operand, valoarea acestuia este folosită la evaluarea expresiei din care face parte, apoi este modificată: var++ (postincrementare) sau var-- (postdecrementare).

Incrementare/decrementarea au ca efect modificarea valorii operandului cu o unitate.

Semnificația unei unități depinde de tipul operandului asupra căruia se aplică. Pentru operanzi numerici, o unitate înseamnă unu.

Operatorul % are ca rol obținerea restului unei împărțiri întregi. Operatorul / are efect diferit, în funcție de tipul operanzilor:

- a. dacă cel puțin un operand este de tip real, se execută împărțire reală;
- b. dacă ambii operanzi sunt întregi, se execută împărțire întreagă și se obține câtul.

Calculul câtului și restului unei împărțiri întregi se poate realiza și prin intermediul funcției *div*. Rezultatul funcției are tipul *div\_t*, definit în biblioteca *stdlib.h* astfel:

```
typedef struct {long int quot; //cât
               long int rem;  //rest
               } div_t;
```



#### Exemplu:

Determinarea câtului (*cat*) și restului (*rest*) împărțirii numărului *m* la numărul *n* se realizează astfel:

```
div_t x;
int m,n,cat,rest;
x=div(m,n);
cat=x.quot;
rest=x.rem;
```

#### 4.3.4. Operatorii logici și relaționali

Expresiile logice sunt cele care, în urma evaluării produc, în interpretarea programatorului, valorile *adevărat* sau *fals*. Astfel de valori sunt produse de două categorii de operatori: operatori logici propriu-ziși și operatori relaționali.

Operatorii logici sunt prezentați în tabelul 4.4. (în ordinea priorității) iar operatorii relaționali în tabelul 4.5.

Tabelul 4.4. Operatorii logici

Semnificație operație	Operator
Negare	!
Și logic	&&
Sau logic	
Sau exclusiv logic	Nu există

În limbajul C nu există tipul de dată logic. Operanzii asupra cărora se aplică operatorii logici sunt convertiți în valori numerice și interpretați conform convenției: *adevărat* pentru valoare nenulă și *fals* pentru zero. Rezultatul unei operații logice este de tip *int*, conform convenției: pentru *adevărat* valoarea 1 iar pentru *fals* valoarea 0.

La evaluarea expresiilor logice se aplică principiul evaluării parțiale: dacă expresia este de tip aditiv (operandi legați prin operatorul *sau*), în momentul în care s-a stabilit că un operand are valoarea *adevărat* toată expresia va fi *adevărată* și nu se mai evaluează restul operandilor. Asemănător, pentru o expresie multiplicativă (operandi legați prin operatorul *și*), dacă un operand are valoarea *fals*, expresia are valoarea *fals* și nu se mai evaluează restul operandilor.

Operația *sau exclusiv* între doi operandi *a* și *b* se efectuează cu expresia  $!a \& \& b || !b \& \& a$ , conform tabelului următor:

a	b	!a	!b	!a&&b	!b&&a	!a&&b  !b&&a
0	0	1	1	0	0	0
≠ 0	≠ 0	0	0	0	0	0
≠ 0	0	0	1	0	1	1
0	≠ 0	1	0	1	0	1

Operatorii relaționali sunt prezentați în tabelul 4.5. Operatorii relaționali au același nivel de prioritate.

Rezultatul unei operații relaționale este de tip `int`, conform convenției: pentru adevărat valoarea 1, iar pentru fals valoarea 0.

Pentru a nu se greși la evaluarea unor expresii complexe, este indicat să se facă raționamentul bazat pe logica booleană (cu valori de adevărat și fals).

Tabelul 4.5. Operatorii relaționali

Semnificație operație	Operator
Mai mare	>
Mai mare sau egal	>=
Mai mic	<
Mai mic sau egal	<=
Egal	==
Diferit	!=

#### 4.3.7. Operatorii la nivel de bit

Limbajul permite operații pe biți, ai căror operatori sunt prezentați în tabelul 4.6.

Tabelul 4.6. Operații pe biți

Semnificație operație	Operator
Și logic pe biți	&
Sau logic pe biți	
Sau exclusiv logic pe biți	^
Negare (complement față de 1)	~
Deplasare la dreapta	>>
Deplasare la stânga	<<



Operandii pot fi de orice tip întreg. Dacă este nevoie, operandii sunt extinși la reprezentare pe 16 biți. Execuția are loc bit cu bit. Pentru o pereche de biți (x,y), valorile posibile rezultate în urma aplicării operatorilor logici la nivel de bit sunt prezentate în tabelul 4.7.

Tabelul 4.7 Rezultatele operațiilor logice pe biți

x	y	x&y	x y	x^y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Tot în categoria operațiilor la nivel de bit se încadrează și deplasările binare la stânga sau la dreapta a conținutului unei variabile întregi:

operand<<număr\_poziții și operand>>număr\_poziții

#### Exemplu:

1. Următoarea secvență afișează, în ordine inversă, biții unui octet (dată de tip *char*).

```
char c;
scanf("%d",&c);
for(int i=0;i<8;i++,c=c>>1)
printf("%d",c&1);
```

#### 4.3.6. Operatorul virgulă (,)

Operatorul virgulă este specific limbajului C. Pe lângă rolul de separator într-o listă, virgula este considerată și operator într-o secvență de forma:

expresie\_1, expresie\_2, ..., expresie\_n

Operatorul *virgulă* (,) induce evaluarea succesivă a expresiilor, de la stânga la dreapta, întreaga secvență fiind tratată ca o expresie căreia i se atribuie în final valoarea și tipul corespunzătoare ultimei expresii evaluate (*expresie\_n*). Operatorul se utilizează acolo unde este legal să apară o expresie în program și se dorește realizarea unui calcul complex, exprimat prin mai multe expresii.

#### Example:

```
int a=10, b=3, c, d;
d=(c=a+b, b+=c, a/2);
```

Valorile variabilelor după evaluarea expresiei sunt: a=10, c=13, b=16, d=5 (câtu împărțirii lui 10 la 2).

#### 4.3.7. Operatorul de conversie explicită

Limbajul oferă posibilitatea conversiei explicite a tipului unei expresii către un tip dorit de utilizator (dar compatibil). Expresia de conversie are forma generală:

(tip)operand

Se observă că operatorul nu are un simbol explicit. El este format din numele unui tip de dată inclus între paranteze. Construcția este numită *expresie cast* (conversia explicită se numește *typecast*). Trebuie reținut că nu se schimbă efectiv tipul operandului, doar se folosește în expresie valoarea operandului convertită la un alt tip. Operația de conversie de tip nu are efect permanent.

#### Exemplu:

```
int a=7;
float b=(float)a;
```

După execuția secvenței, *a* are valoarea 7 (nu 7.0) nefiind afectată în nici un fel de operația de conversie de tip, iar *b* are valoarea 7.0 (obținută prin conversia valorii lui *a* către tipul *float*).

#### 4.3.8. Operatorul dimensiune

Operatorul dimensiune este folosit pentru a afla dimensiunea în octeți a spațiului de memorie ocupat de o variabilă de sau definit de un tip. Operatorul are mnemonica *sizeof* și poate fi folosit în una din formele:

`sizeof var sau sizeof(var) sau sizeof(tip)`

unde *tip* este numele unui tip de dată iar *var* poate fi numele unei variabile simple, numele unui masiv, numele unui articol, un element al unui masiv sau un câmp al unui articol.

În primele două forme, rezultatul va fi numărul de octeți alocați entității respective. Pentru ultima formă, rezultatul este numărul de octeți pe care îi va ocupa o variabilă de tipul *tip*.

#### 4.3.9. Operatorii paranteze

Parantezele au rolul de a include o expresie sau lista de parametri ai funcțiilor. Prin includerea subexpresiilor între paranteze se modifică ordinea de evaluare a unei expresii. Asupra unei expresii între paranteze nu se pot aplica orice operatori (de exemplu nu se pot aplica operatorii de incrementare sau decrementare).

La apelul unei funcții, parantezele rotunde „()” sunt numite *operatori de apel de funcție*, ele delimitând lista parametrilor reali. Parantezele pătrate „[]” au rolul de a include expresii care reprezintă indici pentru accesarea elementelor unui masiv. Ele se numesc *operatori de indexare*.

#### 4.3.10. Operatorul condițional

Operatorul condițional este singurul care are trei operanzi și este specific limbajului C. Forma generală este:

`expresie_1?expresie_2:expresie_3`

unde *expresie\_1*, *expresie\_2* și *expresie\_3* sunt expresii. Operatorul condițional are simbolul `?:`. Cele două caractere care compun simbolul apar intercalate între cei trei operanzi, conform formei generale. Construcția se numește *expresie condițională*. Valoarea și tipul acestei expresii sunt identice cu cele ale *expresie\_2* (dacă *expresie\_1* este adevărată) sau cu cele ale *expresie\_3* (dacă *expresie\_1* este falsă).

#### 4.3.11. Alți operatori

Din categoria operatorilor limbajului fac parte și următorii:

- a. operatorul de calificare, cu simbolul `.` (caracterul punct), folosit pentru a accesa un câmp al unui articol;
- b. operatorul de calificare (cu simbolul `->`) folosit pentru a accesa un câmp atunci când se știe adresa unei structuri;
- c. operatorul de referențiere (cu simbolul `&`) folosit pentru a extrage adresa unei variabile;
- d. operatorul de referențiere (cu simbolul `*`) folosit pentru a defini un pointer;
- e. operatorul de dereferențiere (cu simbolul `*`) folosit pentru a extrage valoarea de la o anumită adresă.

Ultimii patru operatori sunt specifici lucrului cu adrese (care nu face obiectul prezentei lucrări).

#### 4.3.12. Evaluarea expresiilor

Expresiile sunt evaluate pe baza următoarelor reguli:

- *Precedența*: determină ordinea de efectuare a operațiilor într-o expresie în care intervin mai mulți operatori (gradul de prioritate);


- *Asociativitatea*: indică ordinea de efectuare a operațiilor în cazul unui set de operatori cu aceeași precedență;

- *Regulile de conversie de tip*: asigură stabilirea unui tip comun pentru ambii operanzi, pentru fiecare operație care solicită acest lucru și în care tipurile diferă.

Asociativitatea și precedența operatorilor (începând cu prioritatea maximă) sunt redată în tabelul 4.8.

În concluzie, expresiile care pot fi construite în limbajul C se încadrează în următoarele categorii: expresii de atribuire, expresii aritmetice, expresii logice, expresii relaționale.

**Tabelul 4.8.** Precedența operatorilor

Operatori	Asociativitate	Grad de prioritate
() [] . ->	de la stânga la dreapta	maxim
+ - & * (unari) ++ -- (tip) sizeof ! ~	de la dreapta la stânga	
* (binar) / %	de la stânga la dreapta	
+ - (binari)		
<< >>		
< <= > >=		
== !=		
& (binar)		
^		
&&		
?:		
= < <= > >= += -= *= /= %= &= ^=  =	de la dreapta la stânga	
,	de la stânga la dreapta	minim



### Teste de autoevaluare

4. Specificați care va fi valoarea variabilei c.

```
int a=7, b=9, c;  
c = (a > b) ? a : b;
```

5. Scrieți secvențele echivalente pentru următoarele exemple:

```
y = x++;  
y = --x;
```

## 4.4. Realizarea structurilor fundamentale de control

### 4.4.1. Tipurile de instrucțiuni

Instrucțiunile unui program C se grupează într-un bloc delimitat de o pereche de acolade `{ }`. Instrucțiunile se încheie cu caracterul `;` (punct și virgulă), care este terminator de instrucțiune și este obligatoriu pentru a marca sfârșitul fiecăreia.

Un program trebuie să execute cel puțin o instrucțiune, chiar dacă, la limită, aceasta este vidă. În limbajul C, un program se regăsește sub forma unei funcții rădăcină, care, la limită, poate avea un corp vid:

```
void main() { }
```

După modul de realizare a construcțiilor sintactice și al numărului de acțiuni descrise, se disting instrucțiuni *simple* și *structurate*. De asemenea, pot fi create blocuri de instrucțiuni executabile, denumite instrucțiuni *compuse*.

● O instrucțiune compusă este o secvență de instrucțiuni (simple, structurate sau compuse) delimitată de perechea de acolade `{ }`. Ea implementează natural structura secvențială din programarea structurată. Mulțimea instrucțiunilor executabile ale unui program este, la limită, o instrucțiune compusă. Orice instrucțiune (simplă sau structurată) poate fi transformată în instrucțiune compusă.


● O instrucțiune este simplă dacă descrie o singură acțiune, unic determinată și care nu provoacă condiționări. Din categoria instrucțiunilor simple fac parte, de exemplu: instrucțiunea vidă, instrucțiunea de evaluare a unei expresii, `goto`, `break`, `continue`, `return`.

● Instrucțiunile structurate sunt construcții care conțin alte instrucțiuni (simple, compuse sau structurate) care vor fi executate alternativ sau repetitiv. Corespunzător, prin instrucțiunile structurate se codifică structurile fundamentale alternative sau repetitive din algoritm.

Pe lângă instrucțiunile care implementează conceptele programării structurate, C conține și instrucțiuni care contravin acestora, datorită orientării limbajului spre compactarea textului sursă și spre neconformism în stilul de programare impus de autori.

### 4.4.2. Instrucțiunile simple

Instrucțiunea vidă descrie acțiunea vidă. În C nu are o mnemonică explicită, fiind dedusă din contextul unor construcții sintactice. Ea se folosește acolo unde trebuie să apară o instrucțiune, dar care nu trebuie să execute nimic. Situația este întâlnită de obicei în cazul instrucțiunilor structurate.

 **Exemple:**

```

if (c==1) ; else c=2;
↑
instrucțiune vidă
↑
if (c==1) ; else ;
↑      ↑
instrucțiuni vide
↑      ↑
{;}
↑
instrucțiune vidă
```

*Instrucțiunea de tip expresie* evaluează o expresie care, în cele mai dese cazuri, este de atribuire sau de apel al unei funcții (vezi și capitolul *Operatori și expresii*). Instrucțiunea de tip expresie se obține scriind terminatorul de instrucțiune după o expresie (acolo unde este legal să apară o instrucțiune în program). Forma generală este:

```
expresie;
```

### 4.4.3. Instrucțiunea compusă

Instrucțiunea compusă este o succesiune de instrucțiuni și declarații, cuprinse între o pereche de acolade. Se preferă ca declarațiile să fie plasate înaintea instrucțiunilor. Forma generală este:

```
{declaratii
  instructiuni}
```

Declarațiile sunt valabile în interiorul instrucțiunii compuse. Instrucțiunea compusă se utilizează acolo unde este nevoie să se execute mai multe acțiuni, dar sintaxa impune prezența unei singure instrucțiuni: mai multe instrucțiuni sunt „transformate” într-o singură instrucțiune (compusă). Situația este întâlnită în cadrul instrucțiunilor *if*, *while*, *do*, *for*, care precizează, în sintaxa lor o singură instrucțiune.

#### 4.4.4. Instrucțiunile structurate

În continuare, prin instrucțiune se înțelege o instrucțiune simplă, compusă sau structurată.

##### Realizarea structurilor alternative

a) *Structura alternativă simplă* (figura 4.2) permite realizarea unei ramificări logice binare, în funcție de valoarea unei condiții (expresie cu rezultat logic). Instrucțiunea care realizează această structură este *if*:

```
if(expresie)instructiune_1;
[else instructiune_2];
```

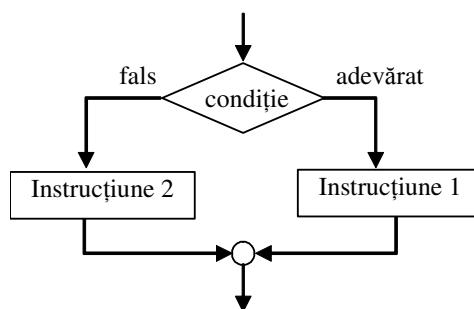


Fig. 4.2. Structura alternativă simplă

Expresia poate fi de orice tip. Dacă valoarea expresiei este diferită de zero (valoare asociată din punct de vedere logic cu *adevărat*) se execută *instrucțiune\_1*; în caz contrar se execută *instrucțiune\_2* sau se iese din structură (când construcția *else* lipsește, caz în care instrucțiunea *if* realizează structura pseudoalternativă).



##### Exemple:

1)     if(a>b) a=c;  
       else a=b;

2. Rezolvarea ecuației de gradul I,  $ax+b=0$ :

```
a ? printf("Solutia este %10.5f", (float)-b/a) : b ? printf("Ecuația nu are soluții") : printf("Soluții: orice x real");
```

b) *Structura alternativă multiplă* permite alegerea unei acțiuni dintr-un grup, în funcție de valorile pe care le poate lua un selector (figura 9.2). În limbajul C structura se simulează cu instrucțiunea *switch*, a cărei sintaxă este:

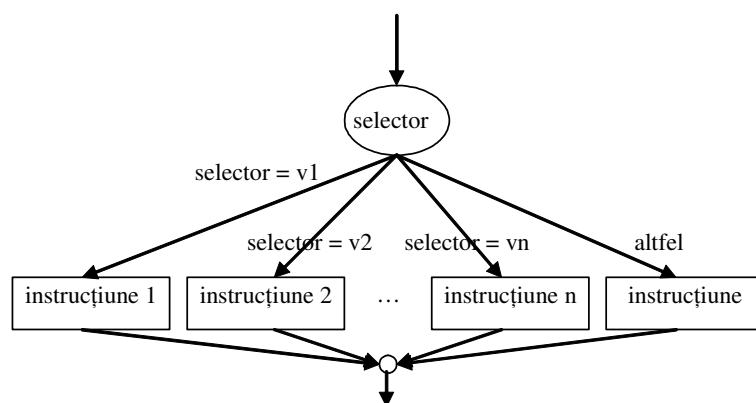
```
switch(expresie)
{case c_1: instructiuni_1;
 case c_2: instructiuni_2;
 .....
 case c_n: instructiuni_n;
 [default: instructiuni;]}
```

unde: *expresie* este de tip întreg;  $c_1, c_2, \dots, c_n$  sunt expresii constante, de tip *int*, unice (o valoare nu poate să apară de două sau mai multe ori); *instrucțiuni\_1, instrucțiuni\_2, ..., instrucțiuni\_n, instrucțiuni* sunt simple, compuse sau structurate. Dacă pe o ramură sunt mai multe instrucțiuni, nu este nevoie să fie incluse între acolade (să fie instrucțiune compusă).

Instrucțiunea *switch* evaluează expresia dintre paranteze, după care caută în lista de expresii constanta cu valoarea obținută. Dacă este găsită, se execută instrucțiunile asociate valorii respective și, secvențial, toate instrucțiunile care urmează, până la terminarea structurii de selecție. Dacă valoarea căutată nu este găsită în listă și ramura *default* este prezentă, se execută instrucțiunile asociate acesteia. Dacă ramura *default* lipsește nu se execută nimic. Pentru a limita acțiunea strict la execuția instrucțiunilor asociate unei valori, trebuie inclusă instrucțiunea *break*, care determină ieșirea din structura *switch*.

Pentru a simula structura din figura 4.3 se scrie:

```
switch(expresie)
{case c_1: instrucțiuni_1; break;
 case c_2: instrucțiuni_2; break;
 .....
 case c_n: instrucțiuni_n; break;
 [default: instrucțiuni;]}
```



**Fig. 4.3.** Structura alternativă multiplă

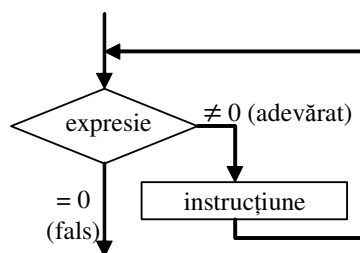
### Realizarea structurilor repetitive

a) *Structura repetitivă condiționată anterior* este implementată prin instrucțiunea *while* cu forma generală:

```
while (expresie) instrucțiune;
```

Instrucțiunea *while* se execută astfel: se evaluează expresia și dacă este diferită de zero (adevărată) se execută instrucțiunea (simplă, compusă sau structurată), apoi se reia procesul până când la evaluarea expresiei se obține valoarea zero. În acest moment se încheie execuția instrucțiunii *while*. Pentru a asigura ieșirea din ciclu, instrucțiunea trebuie să modifice valoarea expresiei.

Dacă la prima evaluare a expresiei se obține valoarea zero, atunci nu se execută nimic. Instrucțiunea *while* implementează natural structura corespunzătoare din teoria programării structurate. Mecanismul de execuție este redat în figura 4.4.



**Fig. 4.4.** Mecanismul de execuție a instrucțiunii *while*

**Example:**

1. `while (a>b)`  
`a=a+1;`
2. `i=0;`

```

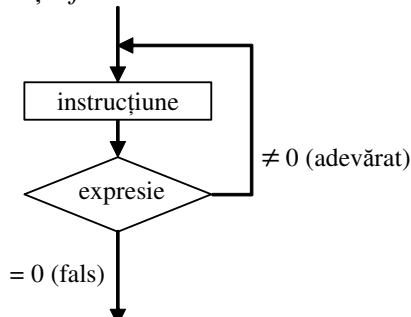
while (i<n)
{ printf("\n %4.2f",v[i]);
  i=i+1;}
  
```

b) *Structura repetitivă condiționată posterior* este implementată (cu unele deosebiri față de teoria programării structurate), prin intermediul instrucțiunii *do-while*. Forma generală este:

```

do instrucțiune
while (expresie);
  
```

Instrucțiunea *do-while* se execută astfel: se execută *instrucțiune* apoi se evaluează expresia; dacă expresia este nenulă (adevărată), se repetă procesul, altfel se încheie execuția. Mecanismul de execuție este redat în figura 4.5. Se observă că instrucțiunea *do-while* se abate de la teoria programării structurate, realizând repetiția pe condiția *adevărat*, în timp ce structura fundamentală o realizează pe condiția *fals*.



**Fig. 4.5.** Mecanismul de execuție a instrucțiunii *do-while*

Instrucțiunea *do-while* este echivalentă cu secvența:

```

instrucțiune;
while(expresie) instrucțiune;
  
```

**Example:**

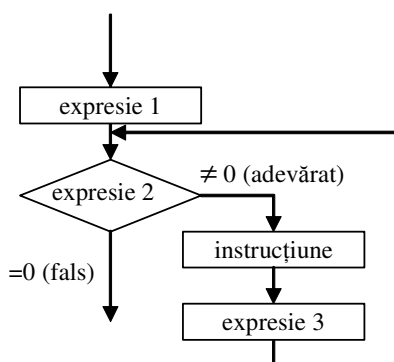
1. `do a=a+1; while (a<=100);`
2. `#include <stdio.h>`  
`main()`  
`{ unsigned i;`  
`i=1;`  
`do`  
`{ printf("+");`  
`printf("-");`  
`i++;}`  
`while(i<=80);}`

c) *Structura repetitivă cu numărător* nu este implementată în C. Ea poate fi simulată prin instrucțiunea *for*, care, datorită facilităților deosebite pe care le oferă, poate fi considerată ca o instrucțiune repetitivă cu totul particulară, nedefinită în teoria programării structurate. Ea este mai apropiată de structura *while-do*. Instrucțiunea *for* din C are următoarea formă generală:

`for(expresie_1; expresie_2; expresie_3)instrucțiune;`

Instrucțiunea *for* se execută astfel: se evaluează *expresie\_1*; se evaluează *expresie\_2* și dacă este nulă (fals), se încheie execuția lui *for*, altfel se execută *instrucțiune* apoi se evaluează *expresie\_3*. Se revine la evaluarea lui *expresie\_2* ș.a.m.d. Instrucțiunea *for* realizează structura repetitivă din figura 4.6 și poate fi înlocuită cu secvența:

```
expresie1;
while (expresie2)
{ instrucțiune;
  expresie3;}
```



**Fig. 4.6.** Mecanismul de execuție a instrucțiunii *for*

Pentru a simula structura repetitivă cu numărător (do-for) se folosesc forme particulare pentru cele trei expresii:

- ✦ *expresie\_1* va fi expresia de inițializare: *contor=valoare inițială*;
- ✦ *expresie\_2* controlează terminarea ciclului: *contor<valoare finală*;
- ✦ *expresie\_3* realizează avansul contorului: *contor=contor+pas*.



#### Exemplu:

```
/*citirea elementelor unui vector*/
#include <stdio.h>
void main()
{ float v[20];
  int n,i;
  printf("\n Nr. de elemente:");
  scanf("%i",&n);
  for(i=0;i<n;i++)
  { printf("\n v[%i]=",i+1);
    scanf("%f",&v[i]);}
}
```

#### 4.4.5. Instrucțiunile de salt necondiționat și ieșire forțată din structuri

Instrucțiunile de salt necondiționat și ieșire forțată din structuri contravin principiilor programării structurate, dar pot fi utilizate în măsura în care, în aplicații complexe, ușurează munca programatorului.

✶ Prin instrucțiunea ***continue***, care se poate folosi numai în interiorul unei structuri repetitive, se abandonează iterația curentă și se trece la următoarea. Forma ei este:

`continue;`



Instrucțiunea are următoarele efecte:

- în interiorul instrucțiunilor *while* și *do-while*: se abandonează iterația curentă și se trece la evaluarea expresiei care controlează terminarea buclei.
- în interiorul instrucțiunii *for*: se abandonează iterația curentă și se trece la evaluarea parametrului *expresie\_3*.

● Instrucțiunea ***break*** este folosită numai în interiorul unei instrucțiuni structurate și are ca efect terminarea imediată a execuției instrucțiunii respective.

● Instrucțiunea ***goto*** realizează salt necondiționat și este moșnită din primele limbaje de programare, care nu foloseau principiile programării structurate. Forma generală este:

*goto* etichetă;

O etichetă este un identificator urmat de caracterul **:** (două puncte), după care urmează o instrucțiune. Rolul unei etichete este de a defini un punct de salt, către care se poate face trimitere prin instrucțiunea *goto*. Etichetele au valabilitate locală, în corpul funcției în care sunt definite. Din acest motiv, nu se poate face salt din corpul unei funcții la o instrucțiune aflată în altă funcție.



### Test de autoevaluare

6. Care din următoarele secvențe **nu** realizează suma a **n** elemente ale unui vector:  
**a) `s=0; for(i=0; i<n; i++) s+=x[i];` b) `s=0; for(i=n-1; i>=0; i--) s+=x[i];` c) `s=0; i=0; while (i<n) {s+=x[i]; i++;}` ; d) `s=0; i=n-1; while (i>0) {s+=x[i]; i--;}` ; e) `s=0; i=0; do { s+=x[i]; i++; } while(i<n);`**
7. Secvența: `for(i=0; i<n-1; i++) {z=x[i]; p=i; for(j=i+1; j<n; j++) if(x[j]<z) {z=x[j]; p=j; } a=x[i]; x[i]=z; x[p]=a; }` realizează:  
**a) minimul dintr-un vector cu reținerea poziției primei apariții; b) minimul dintr-un vector cu reținerea poziției ultimei apariții; c) sortarea unui vector prin metoda bulelor; d) sortarea unui vector prin metoda selecției; e) căutarea unei valori date într-un vector.**
8. Triunghiul de sub diagonală secundară (inclusiv diagonală) unei matrice pătrate se poate parcurge numai cu secvențele: 1. `for(i=0; i<n; i++) for(j=n-i-1; j<n; j++) ...;` 2. `for(i=0; i<n; i++) for(j=n-1; j>=n-i-1; j--) ...;` 3. `for(i=n-1; i>=0; i--) for(j=n-i-1; j<n; j++) ...;` 4. `for(i=n-1; i>=0; i--) for(j=n-1; j>=n-i-1; j--) ...;` 5. `for(j=0; j<n; j++) for(i=n-j-1; i<n; i++) ...;` 6. `for(j=0; j<n; j++) for(i=n-1; i>=n-j-1; i--) ...;` 7. `for(j=n-1; j>=0; j--) for(i=n-j-1; i<n; i++) ...;` 8. `for(j=n-1; j>=0; j--) for(i=n-1; i>=n-j-1; i--) do ...`  
**a) 1,2,5 și 6; b) 3,4,7 și 8; c) 1,2,3 și 4; d) 5,6,7 și 8; e) toate.**
9. Următoarele secvențe descriu algoritmi recursivi: 1) `s=0; for(i=n-1; i>=0; i--) s+=x[i];` 2) `for(i=0; i<n; i++) y[i]=x[i];` 3) `nr=0; i=0; while(i<n) {if(x[i]>0) nr+=1; i++;}` ; 4) `for(i=0; i<n; i++) z[i]=x[i]*y[i];` 5) `i=0; z=0; do {z+=x[i]*y[i]; i++;} while(i<n);` 6) `s=1; for(i=0; i<n; i++) s*=i;`  
**a) toate; b) 1,3,5 și 6; c) 2,4 și 6; d) 3 și 5; e) niciunul.**

## Răspunsuri și comentarii la testele de autoevaluare

1. După preprocesare, secvența de cod va deveni:

```
char v[10],v1[10+(5+10)*(6)];
char v1[10*(10+10)];
char m[10][10];
```

2. Articolul se declară astfel:

```
struct magazin {int cod_magazin;
                float vanzari_lunare[12];
            } articol;
```

Articolul are 50 de octeți, iar referirea câmpurilor se realizează astfel:

articol.cod\_magazin; articol.vanzari\_lunare[i], cu  $i=0,1,\dots,11$ .

```
#include <stdio.h>
```

```
void main()
```

```
{ struct magazin {
    int cod_magazin;
    float vanzari_lunare[12];
};
```

**//Inițializarea articolului;**

```
struct magazin gigel_srl={200, 1,2,3,4,5,6,7,8,9,10,11,12};
```

**//sau cu evidențierea structurii de masiv:**

```
struct magazin gigel_srl1={200, {1,2,3,4,5,6,7,8,9,10,11,12}};
printf("\n%6.2f",gigel_srl1.vanzari_lunare[10]);
```

3. Declararea și inițializarea câmpurilor unui student la zi pentru structura prezentată se realizează astfel:

```
#include <stdio.h>
```

```
void main()
```

**{ //Declararea articolului cu structura variabila:**

```
struct articol { char nume[40];
                struct { int zi, luna, an;} datan;
                int an_st; char forma_inv;
                union { struct {char bursa; float valoare;} zi;
                        struct {char loc_m[30];
                                struct {int zi, luna, an;}data_ang;
                                }id; } parte_vb; };
```

**//Inițializarea câmpurilor unui student la zi:**

```
struct articol a={"Popescu Felix",{4,1,1974},1,'Z',{ 'D',250.5}};
printf("\nData nasterii: %i.%i.%i, Forma de inv.: %c, Val. bursa: %6.2f",
a.datan.zi, a.datan.luna, a.datan.an, a.forma_inv, a.parte_vb.zi.valoare); }
```

4. Variabila  $c$  primește valoarea maximă dintre  $a$  și  $b$ , anume 9.

5.  $y=x++$ ; este echivalent cu secvența  $y=x$ ;  $x=x+1$ ;

$y=--x$ ; este echivalent cu secvența  $x=x-1$ ;  $y=x$ ;

6:d); 7:d); 8:e); 9:b).

## Bibliografia unității de învățare

1. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, M. Mircea, L. Bătăgan, C. Silvestru, Bazele programării calculatoarelor. Teorie și aplicații în C, Ed. ASE, București, 2006, ISBN 973-594-591-6
2. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, Programarea calculatoarelor. Știința învățării unui limbaj de programare, Teorie și aplicații, Ed. ASE, 2003
3. Ion Smeureanu, Marian Dârdală, Programarea în limbajul C/C++, Ed. CISON, București 2004, ISBN 973-99725-7-8

## Bibliografie

1. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, M. Mircea, L. Bătăgan, C. Silvestru, Bazele programării calculatoarelor. Teorie și aplicații în C, Ed. ASE, București, 2006, ISBN 973-594-591-6
2. I. Gh. Roșca, B. Ghilic-Micu, C. Cocianu, M. Stoica, C. Uscatu, Programarea calculatoarelor. Știința învățării unui limbaj de programare, Teorie și aplicații, Ed. ASE, 2003
3. Ion Smeureanu, Marian Dârdală, Programarea în limbajul C/C++, Ed. CISON, București 2004, ISBN 973-99725-7-8
4. Roșca Gh. I., Ghilic-Micu B., Stoica M., Cocianu C., Uscatu C., *Programarea calculatoarelor. Știința învățării unui limbaj de programare. Teorie și aplicații*, Ed. ASE, București 2003, ISBN 973-594-243-7