

Noțiuni fundamentale

1.1. Noțiunea de algoritm; caracteristici

Algoritm: un instrument de rezolvare a *problemelor*. O problemă se consideră a fi constituită din *date* de intrare și un enunț care specifică relația existentă între datele de intrare și *soluția* problemei.

În cadrul algoritmului sunt descrise prelucrările necesare pentru a obține soluția problemei pornind de la datele de intrare.

Un algoritm este o succesiune bine precizată de prelucrări care aplicate asupra datelor de intrare ale unei probleme permit obținerea în timp a soluției acesteia.

Exemple:

- rezolvarea ecuației de gradul doi,
- ciurul lui Eratostene (pentru generarea numerelor prime mai mici decât o anumită valoare),
- schema lui Horner (pentru determinarea câtului și restului împărțirii unui polinom la un binom) etc.

Soluția problemei se obține prin *execuția* algoritmului. Algoritmul poate fi executat pe o mașină formală (în faza de proiectare și analiză) sau pe o mașină fizică (calculator) după ce a fost implementat într-un limbaj de programare.

Spre deosebire de un program, care depinde de un limbaj de programare, un algoritm este o entitate matematică care este independentă de mașina pe care va fi executat.

Exemplu: Algoritmul de aflare a sumei dintre două numere

Pas 0 : START

Pas 1 : Citim numerele a,b

Pas 2 : $S := a + b$

Pas 3 : Afisam rezultatul : S

Pas 4 : STOP

DATE DE INTRARE → ALGORITM → DATE DE IESIRE

Soluția unei probleme, din punct de vedere informatic, este dată printr-o mulțime de comenzi (instrucțiuni) explicite și neambigue, exprimate într-un limbaj de programare.

Această mulțime de instrucțiuni prezentată conform anumitor reguli sintactice formează un **program**.

Algoritmul descrie soluția problemei independent de limbajul de programare în care este redactat programul.

Un algoritm este compus dintr-o mulțime finită de pași, fiecare necesitând una sau mai multe operații.

Un algoritm nu operează numai cu numere. Pe lângă algoritmi numerici există și algoritmi algebrici și algoritmi logici.

Sintetic, un algoritm și un program pot fi definiți astfel :

Program = exprimarea într-un limbaj de programare a unui *algoritm*

Algoritm = exprimarea într-un limbaj de reprezentare a unui *raționament*

Un **program** este o descriere precisă și concisă a unui algoritm într-un limbaj de programare. De aceea, noțiunile de « algoritm » și « program » se folosesc uneori greșit ca sinonime.

Algoritmizarea este o cerință fundamentală în rezolvarea oricărei probleme cu ajutorul calculatorului. Experiența a demonstrat că nu orice problemă poate fi rezolvată prin *algoritmizarea rezolvării*, adică prin descrierea unui algoritm de rezolvare.

Problemele se pot împărți în 2 clase:

- *clasa problemelor decidabile* (o problemă este decidabilă dacă există un algoritm pentru rezolvarea ei)
- *clasa problemelor nedecidabile* (o problemă este nedecidabilă dacă nu există un algoritm pentru rezolvarea ei).

Există probleme care au apărut ulterior apariției calculatoarelor. Astfel a apărut “*problema celor 4 culori*” conform căreia orice hartă poate fi colorată folosind patru culori, astfel încât oricare două țări cu frontiere comune să fie colorate diferit. Enunțarea problemei a fost făcută în anul 1852; problema a fost rezolvată în anul 1977 doar prin utilizarea calculatorului și prin utilizarea unei metode noi (*metoda Backtracking*).

Există algoritmi cu caracter general, pentru clase largi de probleme și algoritmi specifici unor probleme particulare.

Principalele categorii de algoritmi cu caracter general sunt:

- algoritmi de împărțire în subprobleme (“Divide et Impera”),
- algoritmi de căutare cu revenire (“Backtracking”),
- algoritmi de optim local (“Greedy”),
- algoritmi de programare dinamică etc.

Rezolvarea unei probleme dintr-un anumit domeniu reprezintă o *activitate ce presupune existența unor procese*:

-proces demonstrativ (*demonstrația*) care să arate existența unei soluții sau a mai multor soluții și să determine efectiv *soluțiile exacte*;

-proces computațional (*algoritmul*) care să codifice un proces demonstrativ, o metodă sau o tehnică de rezolvare în scopul *determinării aproximative* a soluțiilor exacte.

Principalele **caracteristici** ale unui algoritm sunt:

-generalitatea – algoritmul trebuie să fie cât mai general astfel ca să rezolve o clasă cât mai largă de probleme și nu o problemă particulară sau punctuală. El trebuie să poată fi aplicat oricărui set de date inițiale ale problemei pentru care a fost întocmit.

Exemplu : algoritmul de rezolvare a ecuației de gradul II $ax^2+bx+c=0$ trebuie să rezolve toate cazurile pentru o mulțime infinită de date de intrare (a, b și c aparținând numerelor reale).

-claritatea– acțiunile algoritmului trebuie să fie clare, simple și riguros specificate. Un algoritm trebuie să descrie cu precizie ordinea operațiilor care se vor efectua.

Fiecare programator trebuie să respecte anumite reguli de editare a programelor. Literatura de specialitate prevede mai multe reguli de indentare a unui program:

- instrucțiunile unei secvențe se vor scrie aliniate, începând de la aceeași coloană;
- instrucțiunile unei instrucțiuni compuse se vor scrie începând toate din aceeași coloană, aflată cu 2-4 caractere la dreapta față de începutul instrucțiunii compuse;
- pe o linie pot fi scrise mai multe instrucțiuni, cu condiția ca ele să aibă ceva comun. Astfel, pentru a obține un program mai compact, 2 până la 4 instrucțiuni scurte de atribuire pot fi scrise pe același rând.
- denumirile variabilelor să fie astfel alese încât să reflecte semnificația acestor variabile.
- programul trebuie amplu comentat, prin inserarea comentariilor în text.

Se consideră că nu trebuie să existe reguli stricte de editare a unui program. În schimb, fiecare programator trebuie să aibă propriile lui reguli de scriere, care să fie unitare pe tot parcursul programului și care să ofere claritate programului.

-finitudinea –un algoritm trebuie să admită o descriere finită și să conducă la soluția problemei după un număr finit de operații.

-corectitudinea – un algoritm trebuie să poată fi aplicat și să producă un rezultat corect pentru orice set de date de intrare valide.

Corectitudinea este de 2 tipuri:

corectitudine totală (faptul că pentru orice date de intrare algoritmul determină valori corecte de ieșire)

parțială (finitudinea algoritmului pentru orice set de date de intrare).

Verificarea corectitudinii unui algoritm se poate face folosind:

- varianta experimentală prin testarea algoritmului
- varianta analitică se bazează pe demonstrarea funcționării corecte a algoritmului pentru orice date de intrare, garantând astfel corectitudinea.

- performanța – algoritmul trebuie să fie eficient privind resursele utilizate și anume să utilizeze memorie minimă și să se termine într-un timp minim.

- robustețea – reprezintă abilitatea algoritmului de a recunoaște situațiile în care problema ce se rezolvă nu are sens și de a se comporta în consecință (de exemplu, prin mesaje de eroare corespunzătoare).

Un algoritm robust nu trebuie să fie afectat de datele de intrare eronate.

-extensibilitatea - posibilitatea adaptării programului la unele schimbări în specificațiile problemei.

-reutilizabilitatea -este posibilitatea reutilizării întregului program sau a unor părți din el în alte aplicații.

-compatibilitatea -presupune ușurința de combinare cu alte produse program.

-portabilitate -este posibilitatea de folosire a produsului program pe alte sisteme de calcul, diferite de cel pe care a fost conceput.

-eficiența -reprezintă măsura în care sunt folosite eficient resursele sistemului de calcul.

Analiza mai presupune utilizarea unor tehnici de demonstrare specifice matematicii.

-testarea programului –conține două faze: depanare (debugging) și trasare (profiling). Depanarea este procesul rulării unui program pe diverse seturi de date de test și corectarea eventualelor erori depistate. Trasarea este procesul de rulare pas cu pas a execuției unui program, pe diverse seturi de date de test, pentru a urmări evoluția valorii unor variabile și a putea depista astfel eventuale erori logice ale programului.

Structura unui algoritm este constituită din următoarele *elemente de bază*:

- *Date - variabile și tipuri de date* utilizate pentru accesul la memorie și generarea de valori conform calculelor implementate în procesul de calcul prin intermediul instrucțiunilor;
- *Expresii* - este alcătuită din unul sau mai mulți operanzi, legați între ei prin operatori. Operanzii pot fi constante sau variabile ;
- *Instrucțiuni - instrucțiuni sau comenzi executabile* pentru operații Input/Output și operații de prelucrare a datelor din memorie conform procesului de calcul;
- *Proceduri și funcții - subprocese de calcul* cu o structură asemănătoare unui algoritm ce pot fi executate prin așa-numitele *instrucțiuni de apelare*.

Practica **dezvoltării aplicațiilor software** arată următoarele faze:

1. **specificarea problemelor** – descrierea clară și precisă a problemelor indiferent din ce domeniu provin acestea. Se impune deci definirea unui enunț precis al problemei.
2. **proiectarea soluțiilor** – includerea problemelor în clasa de probleme corespunzătoare și alegerea modului de reprezentare a problemelor prin formularea *etapelor* și *procedeele* corespunzătoare pentru procesele de rezolvare;
3. **implementarea soluțiilor** – *elaborarea algoritmilor* și *codificarea* acestora într-un limbaj de programare modern. După codificare, are loc obținerea formei executabile a programului.
4. **analiza soluțiilor** – *eficiența soluțiilor* raportată la resursele utilizate: *memorie, timp, utilizarea dispozitivelor I/O*, etc.;
5. **testarea și depanarea** – *verificarea execuției programului* cu diverse seturi de date de intrare pentru a putea răspunde rezolvării oricărei probleme pentru care aplicația a fost elaborată. Această etapă presupune și adaptarea soluțiilor implementate pentru *eliminarea erorilor* în rezolvarea unei anumite probleme și *compatibilitatea* cu sistemul de calcul și sistemul de operare folosite.
6. **documentarea** – în majoritatea cazurilor, programele sunt utilizate de alte persoane decât cele care l-au elaborat. Pentru o utilizare corectă a lor, este necesară întocmirea unei documentații a programului care conține în general descrierea problemei, schema de sistem, schemele logice, programul sursă, instrucțiuni de utilizare. Deseori utilizarea programului este ilustrată figurativ, folosind exemple concrete.
7. **exploatarea, actualizarea și întreținerea** – exploatarea presupune utilizarea curentă a programului în rezolvarea cazurilor concrete din clasa de probleme pentru care a fost proiectat. Actualizarea și întreținerea au și un aspect corectiv, de a elimina eventualele erori descoperite pe parcursul exploatării programului.

Fazele 1 și 2 formează etapa de **analiză și proiectare** iar fazele 3-7 cea de **programare-execuție**.

1.2. Reprezentarea algoritmilor

Algoritmii pot fi specificați:

- În limbaj natural
- Pseudocod
- Scheme logice
- Diagrame arborescente
- Tabele de decizie

1.2.1. Pseudocod

Limbajul pseudocod are o sintaxă și semantică asemănătoare limbajelor de programare moderne, având o anumită flexibilitate în ceea ce privește sintaxa, în ideea că prin codificarea unui algoritm într-un limbaj de programare, operația să fie cât mai comodă. Semantica pseudocodului este apropiată de limbajele de programare utilizând însă cuvinte și expresii uzuale din limbajul natural.

În dicționarul de informatică pseudocodul este definit ca « limbaj utilizat în proiectarea și documentarea programelor obținut prin grefarea unor reguli sintactice pe limbajul natural. Structurile de control sunt reprezentate prin folosirea unor cuvinte cheie (dacă ... atunci ... altfel ... execută ... până când etc) și printr-o anumită aliniere în pagină a liniilor ».

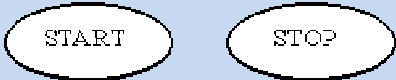
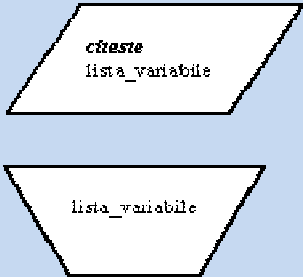
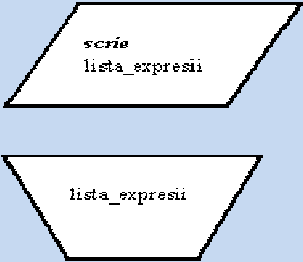
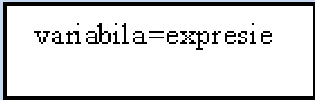
Limbajul pseudocod are două tipuri de propoziții : propoziții standard (corespund structurilor de control) și propoziții nestandard (texte ce conțin părți ale algoritmului încă incomplet elaborate, nefinisate). Comentariile în pseudocod se includ între acolade. Propozițiile standard încep cu cuvinte cheie și fie se scriu cu litere mari, fie se subliniază.

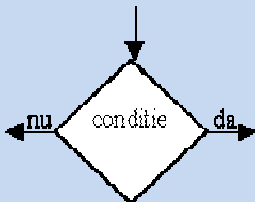
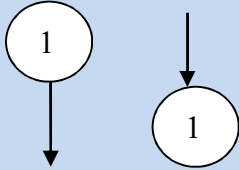
1.2.2. Scheme logice

Prin *schemă logică* se înțelege o reprezentare grafică a unui algoritm în care fiecărui pas i se atașează un simbol denumit *bloc*. Ordinea de parcurgere a blocurilor în schemele logice se precizează cu ajutorul săgeților.

Exprimarea unui algoritmi cu scheme logice sau cu pseudocod este echivalentă.

Într-o schemă logică se utilizează următoarele tipuri de blocuri:

Blocul de inceput/sfarsit - orice schema logica incepe cu un bloc de inceput si se termina cu blocul de stop		PROCEDURE ENDPROCEDURE
Blocul de citire (doua variante) - se citesc de la dispozitivul de intrare valorile variabilelor specificate in lista_variabile		READ lista_variabile
Blocul de scriere (doua variante) - se scriu la dispozitivul de iesire valorile obtinute in urma evaluarii expresiilor din lista		WRITE lista_expresii
Blocul de atribuire - se evaluează expresia, iar valoarea obtinuta este memorata în variabilă, vechea valoare a variabilei pierzându-se		Variabila = expresie

<p>Blocul de decizie - se evalueaza conditia: daca e adevarata se continua cu prelucrarea indicata de ramura <i>da</i>, altfel cu ramura <i>nu</i>;</p>		<pre>IF condiție THEN Secvență instructiuni ELSE Secvența instructiuni ENDIF</pre>
<p>Blocul conector are formă de cerc. El se folosește pentru a conecta diferite secvențe ale unei scheme logice. Sunt utile atunci când se continuă descrierea algoritmului pe mai multe pagini.</p>		

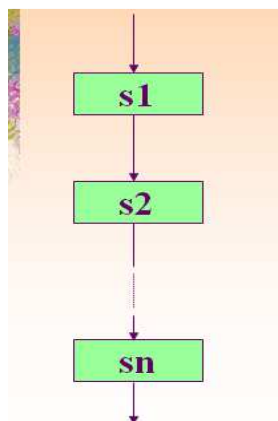
Orice schemă logică care are un singur punct de intrare și un singur punct de ieșire poate fi reprezentată cu ajutorul a trei structuri de bază: **structura secvențială (liniară)**, **structura alternativă**, **structura repetitivă**

O schemă logică construită numai cu structuri de acest tip este numită *schemă logică structurată*. Un algoritm structurat conține doar cele 3 structuri enumerate mai sus.

Structura secvențială

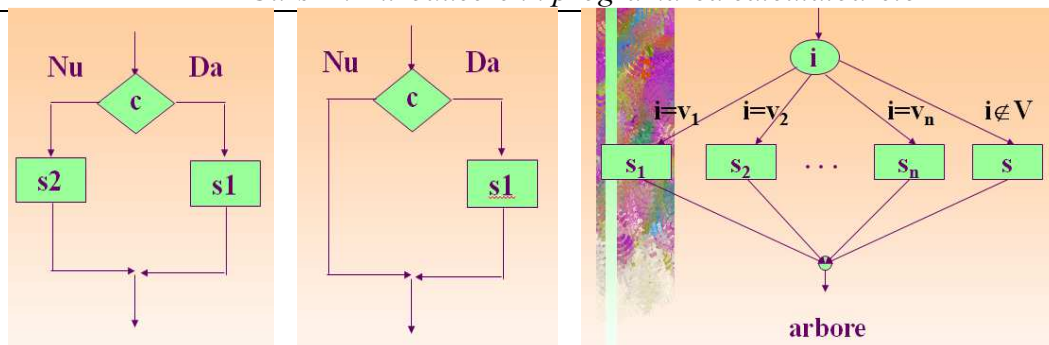
Reprezintă un proces de calcul format dintr-o operație elementară, cum este citirea unei valori sau o operație de atribuire, dar poate fi și o combinație de alte structuri. Structura secvențială indică execuția succesivă a operațiilor de bază și a structurilor de control în ordinea în care apar în schema logică; în general, orice schemă logică cuprinde secvențele:

- citirea datelor
- inițializarea variabilelor
- prelucrări
- tipărirea rezultatelor



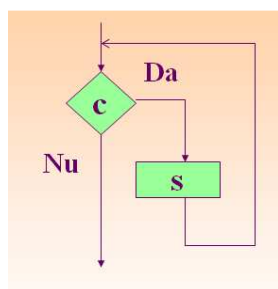
Structura alternativă

În funcție de valoarea de adevăr a condiției, se execută una din secvențe, după care se trece la prelucrarea următoare; cele două ramuri se exclud mutual; este posibil ca una din ramuri să fie vidă.



Structura repetitivă cu condiționare anterioară (WHILE-DO)

Secvența se execută ciclic, cât timp condiția este adevărată; dacă la prima evaluare a condiției, aceasta este falsă, corpul nu se execută niciodată. În cadrul secvenței A este necesar să se modifice valoarea unei variabile care să afecteze valoarea de adevăr a expresiei c. În caz contrar, vom ajunge la un ciclu infinit.



În pseudocod avem:

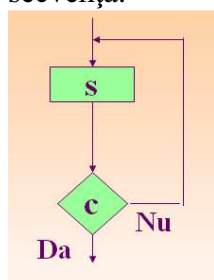
```
WHILE condiție
    Secvență instrucțiuni
ENDWHILE
```

Cele 3 structuri prezentate (secvențială, alternativă și repetitivă cu condiționare anterioară) sunt denumite structuri de bază. Orice algoritm poate fi reprezentat folosind cele 3 structuri de bază.

În continuare prezentăm încă 2 tipuri de structuri, care sunt folosite în practică, dar care au la bază și deci pot fi înlocuite de structurile de bază.

Structura repetitivă cu condiționare posterioară (REPEAT-UNTIL)

Condiția se evaluează după o primă execuție a secvenței (deci secvența se execută cel puțin o dată); se revine la execuția secvenței, dacă nu este adevărată. Dacă condiția este adevărată se încheie această secvență.



În pseudocod avem:

REPEAT

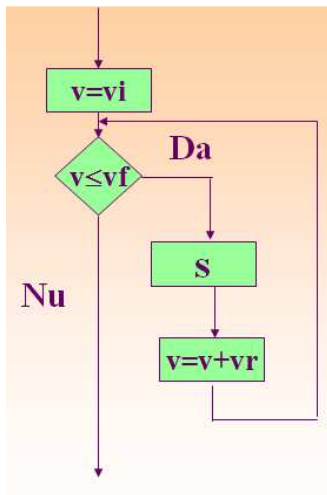
Secvență instructiuni

UNTIL condiție*Structura repetitivă cu un număr cunoscut de pași (FOR)*

O variabilă numită generic contor, controlează ciclarea; contorul se inițializează cu o valoare inițială (val vi), iar ciclarea se realizează cat timp contor \leq vf, o valoare finală.

La sfârșitul corpului ciclului, variabila contor este actualizată, fiind mărită; pentru a nu se cicla la infinit, trebuie ca pasul de mărire (vr) să fie pozitiv.

Structura se poate folosi și cu o valoare finală mai mică decât cea inițială, caz în care avem pasul negativ. Această structură este o particularizare a structurii WHILE-DO.

**FOR** limite de iterare

Secvență instructiuni

ENDFOR

1.2 Definirea limbajelor de programare

Comunicarea interumană este indispensabilă în viața de zi cu zi. Pentru realizarea acesteia, oamenii folosesc diverse modalități, limbajele reprezentând poate cea mai importantă facilitare de comunicare. Una din primele aptitudini importante pe care un copil o dobândește este să vorbească. El învață limba maternă. De obicei, aceasta îi este suficientă pentru comunicarea de zi cu zi cu alți indivizi din propria societate. Dacă însă, un individ călătorește, el are nevoie să cunoască și alte limbi pentru a se putea descurca, pentru a putea comunica cu alți oameni din alte societăți ale lumii. Cu toate că, în principiu, oamenii atribuie înțelesuri similare pentru lucruri similare, ei vorbesc (exprimă) aceste înțelesuri în limbi diferite. Diferența se manifestă atât la nivelul sunetelor și înălțurii acestora pentru a forma cuvintele cât și la nivelul simbolurilor grafice folosite. A învăța să vorbim într-o altă limbă presupune a învăța să rostim cuvintele din limba respectivă, să le înălțuim de o manieră potrivită conform regulilor gramaticale a limbii învățate și să folosim simbolurile specifice acestei limbi pentru a descrie vizual sintagmele de comunicare. Dacă nu mânuim corespunzător aceste elemente, atunci când vom fi puși în situația de a comunica cu cineva care înțelege doar limba respectivă, vom eșua în încercarea noastră.

Am realizat această scurtă introducere pentru a putea realiza o comparație între actele de comunicare interumană și comunicarea dintre om și calculator. În această comunicare, putem vedea calculatorul ca pe un partener care e dispus să ne rezolve problemele. Pentru aceasta, trebuie să-i specificăm modul în care să rezolve aceste probleme. Putem să realizăm acest lucru utilizând algoritmi. Dar algoritmi trebuie descriși

într-un limbaj inteligibil pentru calculator. Putem vedea această comunicare ca un caz special al unei solicitări. Dacă cerem ceva cuiva, atunci va trebui să folosim cuvinte, expresii pe care acesta să le înțeleagă. Conform Dicționarului Explicativ al Limbii Române, prin *limbă* se înțelege un sistem de comunicare alcătuit din sunete articulate, specifice omului, prin care acesta își exprimă gândurile sau dorințele. Astfel, un limbaj este un mijloc de comunicare a ideilor prin sunete și culoare, reprezentând un mijloc de transmitere a informației între indivizii unei categorii.

Orice limbaj are la bază simboluri care formează limbajul respectiv. Astfel, limba vorbită are la bază sunete, limbajele scrise au la bază literele, limbajele vizuale au la bază simbolurile grafice. Semiotica este ramura științei care se ocupă cu studiul simbolurilor.

Revenind la comunicarea dintre om și calculator, trebuie să definim noțiunea de limbaj de programare.

Astfel, prin *limbaj de programare* înțelegem o notație sistematică prin care este descris un proces de calcul.

Un proces de calcul este constituit dintr-o mulțime de pași pe care o mașină îi poate executa pentru a rezolva o anumită problemă.

Astfel, un limbaj de programare este un intermediar între realitatea reprezentărilor utilizatorului asupra problemei de rezolvat și realitatea calculatorului cu care lucrează [Șerbănați 1987]. La rezolvarea unei probleme cu calculatorul, un programator trebuie să privească fiecare element al limbajului din 2 puncte de vedere:

- unul logic, al problemei; astfel programatorul trebuie să știe ce înțelege să reprezinte din problemă cu ajutorul elementului de limbaj
- unul fizic, al implementării, care se referă la ceea ce realizează calculatorul la execuția elementului de limbaj considerat.

Programul este un compromis între cele 2 puncte de vedere. Astfel, programatorul trebuie să-și reprezinte și să înțeleagă următoarele universuri:

- universul problemei
- universul limbajului de programare
- universul calculatorului

Aceste trei universuri sunt în general diferite. Pentru a realiza corespondența între acestea, există definite diverse activități ale informaticii. Figura 1 surprinde corespondența între universurile limbajelor de programare.

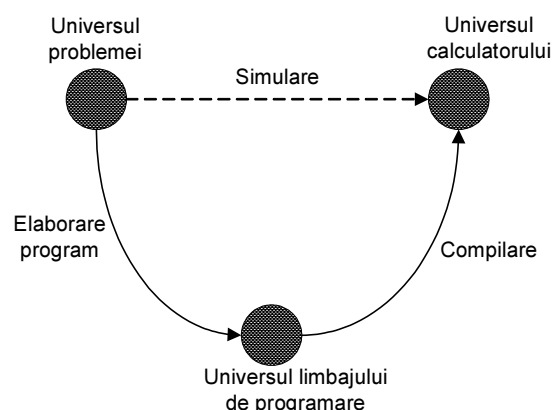


Figura 1. Universurile unui limbaj de programare [Șerbănați 1987].

Orice limbaj are 3 aspecte caracteristice:

- aspectul sintactic
- aspectul semantic
- aspectul pragmatic

Sintaxa unui limbaj conține ansamblul regulilor prin care pornind de la simbolurile de bază care alcătuiesc alfabetul limbajului, se construiesc structuri compuse.

Mulțimea regulilor sintactice care descriu ansamblul propozițiilor sau a formulelor corecte din cadrul limbajului formează *gramatica*.

Deci, sintaxa și gramatica ne ajută să identificăm modul în care putem combina simbolurile de bază ale limbajului pentru a produce elemente acceptate de limbaj. Ele reprezintă imperative riguroase, care pot fi formalizate matematic. Sintaxa se descrie teoretic cu ajutorul sistemelor formale. Următoarele elemente se utilizează pentru descrierea sintaxei unui limbaj:

- diagramele sintactice
- arbori de analiză
- metalimbaje (BNF – Backus Naur Form, EBNF – Extended Backus Naur Form, Asn.1)

Prin **semantică** se înțelege sensul construcțiilor sintactice. Ea reprezintă un set de reguli ce determină semnificația propozițiilor dintr-un limbaj. Este vorba de reguli de evaluare a acestor propoziții în termenii unor mulțimi de valori cunoscute de limbajul respectiv. Astfel, semantica reprezintă înțelesul fiecărei formule corecte admise de gramatică.

Pragmatica se referă la capacitatea de a utiliza construcțiile sintactice și semantice. Referitor la înțelegerea aspectului pragmatic al limbajelor, putem să se închipuim următorul exemplu: o persoană poate cunoaște foarte bine aspectele sintactice și semantice ale unui limbaj, dar nu are capacitatea de a utiliza corect aceste reguli. De asemenea, există persoane care vorbesc (folosesc) o limbă fără a cunoaște aspectele sintactice și semantice ale acesteia. Pragmatica nu se poate formaliza.

În consecință, fiecare problemă o vom aborda în felul următor:

1. vom descrie algoritmul care rezolvă problema fie în pseudocod, fie cu scheme logice
2. vom transpune punctual algoritmul în limbajul de programare ales
3. ne vom asigura că rezolvarea furnizată se execută în mod corect în limbajul de programare ales

1.3 Limbajul de programare C

1.3.1. Scurt istoric al limbajului C

– dezvoltat și implementat în 1972 la AT&T Bell Laboratories de Dennis Ritchie

<http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>

– limbaj de **programare structurat** (blocuri, cicluri, funcții) (concept apărut în ALGOL 60, apoi ALGOL 68, PASCAL, ...)

– necesitatea unui limbaj pentru **programe de sistem** (legătura strânsă cu **sistemul de operare UNIX** dezvoltat la Bell Labs)

– C dezvoltat inițial sub UNIX; în 1973, UNIX rescris în totalitate în C

– cartea de referință: Brian Kernighan, Dennis Ritchie: **The C Programming Language** (1978)

– în 1988 limbajul a fost standardizat de ANSI (American National Standards Institute)

– limbaj de nivel **mediu**: oferă tipuri, operații, instrucțiuni simple fără facilitățile complexe ale limbajelor de nivel (foarte) înalt (nu: tipuri mulțime, concatenare de șiruri, etc.)

– limbaj de programare **structurat** (funcții, blocuri)

– permite programarea **la nivel scăzut**, apropiat de hardware, acces la reprezentarea binară a datelor mare libertate în lucrul cu memoria foarte folosit în programarea de sistem, interfață cu hardware

– produce un cod **eficient** (compact în dimensiune, rapid la rulare) apropiat de eficiența limbajului de asamblare datorită caracteristicilor limbajului, și maturității compilatoarelor

– **slab tipizat** ! necesită mare atenție în programare, conversii implicite și explicite între tipuri, char e tip întreg, etc.

1.3.2 Structura unui program și a unei funcții C

O funcție C este un modul care grupează în interiorul unei perechi de acolade un set de operații codificate sub forma unor instrucțiuni.

Observații:

- Fiecare funcție poate accepta parametri de intrare la apel și poate returna o valoare la revenire.
- Fiecare funcție are un nume precedat de un cuvânt cheie care desemnează tipul funcției (tipul valorii returnate de funcție).
- Numele funcției este urmat de o pereche de paranteze rotunde între care se specifică tipul și numele parametrilor funcției. Parantezele sunt necesare chiar dacă nu există parametri.
- În limbajul C nu este permisă definirea unei funcții în interiorul altei funcții (lucru care este permis în limbajul Pascal). În limbajul C, toate funcțiile trebuie definite în mod independent.

Un program C se compune din una sau mai multe funcții, dintre care una este funcția principală. Fiecare funcție are un nume propriu, cu excepția funcției principale care se numește *main*. Orice program trebuie să aibă o funcție *main* iar execuția programului începe cu prima instrucțiune din această funcție.

Structura generală a unei funcții C este următoarea:

```
tipreturn numefuncție(lista_parametri)
{
    instrucțiuni specifice funcției: declarare variabile locale, instrucțiuni executabile
    return expresie;
}
```

Dacă tipul returnat de funcție nu este specificat, acesta este implicit considerat *int*. Dacă lista parametrilor nu este vidă, fiecare parametru poate fi eventual specificat doar prin numele său, tipurile parametrilor fiind precizate prin declarații care preced corpul funcției.

Prima linie este headerul (antetul) funcției iar ceea ce este inclus între acolade se numește corpul funcției.

Corpul funcției este un bloc care poate conține definiții, declarații de variabile locale și instrucțiuni necesare realizării scopului funcției. De altfel, corpul funcției poate fi considerat o instrucțiune compusă care descrie prelucrările necesare pentru a ajunge la valoarea funcției pornind de la valorile parametrilor.

Rezultatul acțiunii funcției este dat de expresia conținută de instrucțiunea (sau instrucțiunile) return aflată în corpul funcției.

Există funcții care nu au parametri și/sau care nu returnează nici un rezultat. Specificarea acestui aspect se face utilizând cuvântul cheie *void*. Utilizarea lui *void* pentru a indica o listă de parametri vidă este redundantă în C++.

Afișarea unui mesaj pe ecranul utilizator necesită prezența unei instrucțiuni *printf* (Exemplul 2).

Exemplu 2:

```
#include<stdio.h>
int main()
```

```
{
printf("Salut anul 2IE!\n");
return 0;
}
```

Observații:

- prima linie: obligatorie pentru orice program care citește sau scrie, este o directivă de preprocesare, include fișierul `stdio.h` care conține declarațiile funcțiilor standard de intrare/ieșire – adică informațiile (nume, parametri) necesare compilatorului pentru a le folosi corect
- `printf` ("print formatted"): o funcție standard implementată într-o bibliotecă care e inclusă (linkeditată) la compilare; e apelată aici cu un parametru șir de caractere
- șirurile de caractere: incluse între ghilimele duble "
- `\n` este notația pentru caracterul de linie nouă.

Exemplul 4: Citește de la tastatură numele unei persoane, afișează pe ecran o urare de bun venit în lumea C, folosind acest nume

```
#include <stdio.h>
int main()
{
    char nume[10];
    printf("Numele dumneavoastra:");
    scanf("%s", nume);
    printf("Bine ai venit in lumea C! %s\n", nume);
    return 0;
}
```

Sfârșitul unei instrucțiuni se delimitează folosind caracterul `;`.

Apelul unei funcții transferă controlul de la funcția apelantă la funcția apelată. Apelul se face utilizând numele funcției, având între paranteze lista parametrilor actuali. Apelul, fiind o instrucțiune, se termină cu caracterul `;` (punct și virgulă).

Exemplu: Media a doua numere

```
#include <stdio.h>
/* calculeaza si afiseaza media a doua numere */
int main ( ) {
    int a,b;
    float c;          /* declaratii de variabile */
    printf("a="); scanf("%d", &a); /* citire date initiale */
    printf("b="); scanf("%d", &b); /* citire date initiale */
    c= (a+b) / 2.0;    /* instructiune de calcul */
    printf ("%f\n", c);      /* afisare rezultat */
    return 0;
}
```

Exemplu: Pentru două variabile de tip întreg , interschimbați conținutul lor.

```
#include <stdio.h>
int main() {
    int a,b,aux;
    printf("Introduceti cele doua variabile (a,b):");
    scanf("%d,%d",&a,&b);
    aux=a; /* se face interschimbarea */
    a=b;
    b=aux;
    printf("Dupa interschimbare: a=%d, b=%d\n",a,b);
    return 0;
}
```

Exemplu: Pentru două variabile de tip întreg , interschimbați conținutul lor. Nu se folosește o variabilă suplimentară.

```
#include <stdio.h>
int main() {
    int a,b;
    printf("Introduceti cele doua variabile (a,b):");
    scanf("%d,%d",&a,&b);
    a=a-b;
    b=a+b;
    a=b-a;
    printf("După interschimbare: a=%d, b=%d\n",a,b);
    return 0;
}
```

Bibliografie

1. Negrescu Liviu, *Limbajul C*, ed. Agora 1997
2. Logofătu Doina, *Bazele programării în C*, Ed. Polirom 2006
3. Cerchez Emanuela și Șerban Marinel, *Programarea în limbajul C/C++*, Ed. Polirom 2005
4. Bologa Cristian., *Algoritmi*, ed. Risoprint
5. G.C. Silaghi, Mircea Moca, *Limbaje de programare. Metode obiectuale*. Ed. Risoprint 2008