5 ELABORAREA PROGRAMELOR

Tehnologia produselor software [software engineering] caracterizează mulţimea tuturor metodelor şi instrumentelor care permit elaborarea produselor software.

Dezvoltarea unei mari aplicaţii este o problemă complexă care nu constă numai în a programa, în sensul strict al cuvântului, aceasta presupune parcurgerea unui anumit număr de etape, ceea ce constituie **ciclul de viaţă al unui produs software**.

În plus, un mare proiect este dezvoltat de către echipe de programatori, ceea ce obligă la descompunerea problemei în subprobleme pentru fi repartizate între echipe, asigurarea comunicaţiei corespunzătoare între echipe, controlul evoluţiei proiectului şi a calităţii produsului software.

Ciclul de viaţă se compune din fazele următoare:

a) **Analiza şi formularea clară a problemei**: constă în a stabili funcţionalităţile, restricţiile şi obiectivele proiectului în acord cu clientul/utilizatorul. Principala problemă a acestei faze rezidă în comunicarea dintre conceptor şi client.

b) **Specificarea**: constă în determinarea funcţionalităţilor detaliate ale produsului software ce urmează a fi realizat, fără o preocupare deosebită pentru modalitatea de implementare efectivă.

c) **Concepţia**: este faza de definire a structurii modulare a produsului software, alegerea algoritmilor corespunzători precum şi a limbajelor de programare adecvate.

d) **Programarea** constă în implementarea efectivă a diferitelor module care compun produsul software.

e) **Testarea şi validarea**: constă în depistarea erorilor de concepţie şi de programare încât să se poată oferi asigurarea că produsul software răspunde bine exigenţelor formulate iniţial.

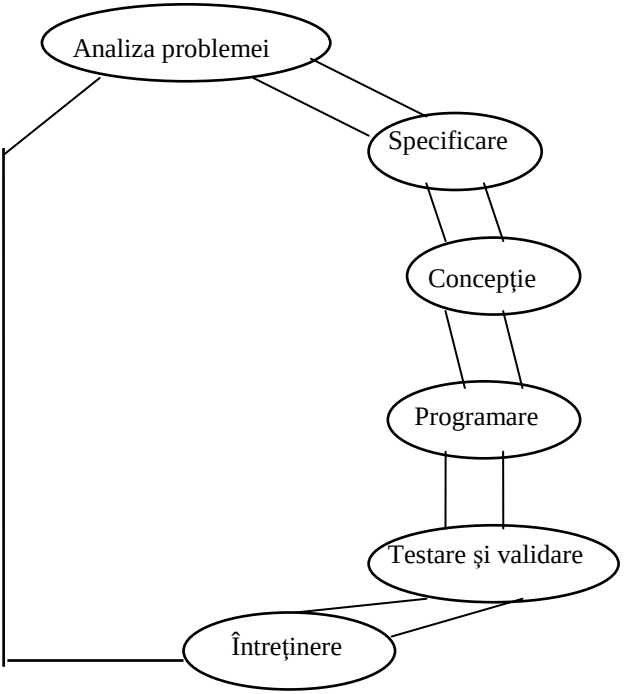
f) **Întreţinerea**: este faza cea mai îndelungată a ciclului de viaţă a unui produs software, care durează toată perioada de exploatare a produsului. În timpul acestei faze se încearcă să se răspundă cerinţelor utilizatorilor, fie corectând erorile apărute, fie efectuând modificări, fie adăugând noi funcţionalităţi.

g) **Documentarea**: constă în regruparea tuturor informaţiilor produse în timpul ciclului de viaţă. Documentarea nu este deci o fază proprie a ciclului de viaţă, dar ea trebuie să se efectueze în timpul fiecărei faze în paralel cu derularea acesteia. Documentarea trebuie să permită înţelegerea şi modificarea ulterioară a produsului software.

Trecerea de la o fază a ciclului de viaţă la următoarea este rareori definitivă, se întâmplă frecvent să se revină la o fază anterioară pentru a se

aduce un anumit număr de modificări. Într-adevăr, este foarte dificil să se prevadă în avans toate aspectele şi în plus, în timpul fazei de întreţinere, modificările importante cum ar fi crearea unei noi versiuni implică dezvoltarea completă pornind de la faza de specificare. Din acest aspect decurge noţiunea de **ciclu**.

Etapele ciclului de viaţă ale unui produs software sunt prezentate schematic în figura următoare:



Conceptul de **software** este utilizat pentru a caracteriza toate programele care sunt executate de către sistemul de calcul. Acest concept se opune aceluia de **hardware** care desemnează aspectul material, fizic al sistemului de calcul.

Execuţia unui program care este elaborat în faza de programare a ciclului de viaţă a unui produs software constă în a furniza sistemului de calcul o secvenţă de instrucţiuni direct interpretabile.

În mod obigatoriu, primele programe erau scrise în binar (**limbajul maşină**), activitate dificilă şi expusă riscurilor de eroare, datorită secvenţelor de biţi a căror semnificaţie era înscrisă într-o tabelă descriind toate operaţiile posibile şi semnificaţia lor binară.

În continuare, pentru uşurarea activităţii de programare s-au elaborat limbajele de asamblare.

Scrierea de programe în limbaj de asamblare rămâne o sarcină complicată şi de asemenea programele depind de sistemul de calcul pe care au fost concepute. Limbaje evoluate, ca de exemplu, **Fortran** sau **Pascal** au adus o soluţie (parţială) acestor probleme.

Ca şi în cazul limbajelor de asamblare, programele scrise în limbajele de programare evoluate trebuie să fie convertite în limbaj maşină pentru a fi executate, iar conversia se poate efectua în două moduri diferite: **traducere** sau **interpretare**.

Traducerea constă în a genera un program echivalent programului sursă, dar codificat în limbajul binar al calculatorului. Traducerea este un procedeu prin care, pornind de programul sursă, se generează un program **obiect**, care se poate încărca ulterior în memorie pentru execuţie.

Interpretarea efectuează conversia şi execuţia unui program într-o singură etapă: instrucţiunile sunt citite unele după altele şi sunt convertite imediat în limbaj maşină prin intermediul interpretorului care realizează execuţia lor pe măsura apariţiei. Totul se petrece ca şi când limbajul sursă ar fi acceptat de către maşină.

Interpretarea este mai bine adaptată dezvoltării şi punerii la punct a programelor deoarece execuţia începe imediat, iar traducerea este indicată pentru exploatarea programelor, când nu mai este reluată traducerea.

Limbajele evoluate dispun în general de un traducător (**compilator**), dar pot dispune şi de un intrerpretor (exemplul limbajului **Pascal**).

4.1 Limbajul de asamblare

Limbajul de asamblare este utilizat astăzi de către specialiştii în informatică, în general pentru rezolvarea unor probleme de optimizare, atunci când se impune exploatarea arhitecturii calculatorului şi funcţionarea sa la nivelul operaţiilor elementare.

Limbajul de asamblare este o variantă simbolică a limbajului maşină. Activitatea de programare este facilitată de utilizarea codurilor de operaţii mnemonice, etichete (adrese simbolice), literale (constante numerice) şi directive (rezervare de spaţiu de memorie, declararea unei macroinstrucţiuni etc.).

Spre deosebire de limbajele evoluate, limbajul de asamblare nu ascunde nimic programatorului, permiţând accesarea tuturor resurselor sistemului şi exploatarea facilităţilor de prelucrare ale acestuia.

O instrucţiune în limbaj de asamblare este divizată în mai multe câmpuri care sunt separate în general prin spaţii. Numărul operanzilor din al zona de adresă variază de la un sistem de calcul la altul între 0 şi 3. După acest câmp

este de dorit adăugarea comentariilor pentru documentare. Iată structura tipică a unei instrucţiuni în limbaj de asamblare:

eticheta	cod operaţie (mnemonic)	operanzi

Codurile operaţiilor mnemonice

Codurile operaţiilor sunt simbolizate într-o tablă a codurilor mnemonice care urmează a fi consultată la scrierea de programe în limbajul de asamblare, mai ales dacă setul de instrucţiuni este complex (arhitecturi CISC).

Operanzi şi etichete

Spre deosebire de limbajul maşină, limbajul de asamblare permite atribuirea de nume pentru **variabile** şi **etichete** (adrese de instrucţiuni), ceea ce uşurează programarea.

De asemenea, operanzii pot să posede un nume care permite referirea lor, iar fiecare registru are un nume predefinit, recunoscut de către programul de traducere.

Exemple de operanzi şi etichete:

```
Tab     DS      1       Definirea unei variabile Tab de un cuvânt;
Zece    DC      10      Definirea constantei Zece care are val. 10;
Ciclu:  MOVE    Zece, A1 Transferul valorii 10 în registrul A1;
        MOVE    A2,Tab  Transfer valoare registru A2 în variabila Tab;
        JUMP    Ciclu   Salt necondiţionat la adresa Ciclu.
```

Literale

Limbajul de asamblare permite definirea valorilor întregi sau reale în diverse baze de numeraţie (2, 8, 10 sau 16) ca şiruri de caractere, care sunt traduse de către asamblor. Specificarea bazei de numeraţie se face prin plasarea unui caracter particular la începutul fiecărei date. Absenţa caracterului particular specifică o dată zecimală. }irurile de caractere sunt de regulă delimitate de caracterul **“ ‘ “**.

Directive

Directivele sau pseudo-instrucţiunile sunt instrucţiuni neexecutabile (referite prin cod mnemonic) care nu au cod maşină echivalent. Acestea sunt indicaţii pentru asamblor (programul de traducere) în vederea traducerii programului.

Exemple de directive:

```
        TTL     ‘Titlul programului’
Vec     DS      50Definire variabilă Vec şi rezervare 50 cuvinte;
```

```
Zero    DC      0 Definire constantă Zero cu valoare 0;
        PLEN    50 50 linii pe pagină (PLEN=Page Length);
        END     Sfârşit de program.
```

Expresii aritmetice

Spre deosebire de limbajele evoluate, expresiile aritmetice utilizate pentru a calcula valoarea unei variabile, ca de exemplu în atribuirea următoare: **A = B + C / D**, nu sunt admise în limbajul de asamblare şi de aceea trebuie programate prin mai multe instrucţiuni.

Macroinstrucţiuni şi subprograme

Anumite asambloare permit structurarea programelor, ele oferă în general posibilitatea de a grupa o secvenţă de instrucţiuni sub forma unui subprogram sau a unei macroinstrucţiuni, în scopul modularizării programului şi a evitării scrierii repetate a unor grupuri de instrucţiuni.

Macroinstrucţiuni

O **macroinstrucţiune** se construieşte prin izolarea unei secvenţe de instrucţiuni căreia i se atribuie un nume simbolic prin care poate fi referită. Ori de câte ori în cadrul programului se face referire la acest nume, asamblorul, în etapa de traducere, înlocuieşte referinţa cu secvenţa de instrucţiuni corespunzătoare.

Utilizarea macroinstrucţiunilor prezintă mai multe avantaje, şi anume:

- extinderea setului de instrucţiuni, deoarece fiecare macroinstrucţiune se utilizează ca o altă instrucţiune;
- reducerea şi structurarea programelor, care conduce la înţelegerea şi modificarea uşoară a acestora;
- economisirea timpului de programare.

Instrucţiunile care servesc pentru definirea şi delimitatea macroinstrucţiunilor (de ex., MACRO şi ENDM) sunt cazuri tipice de directive; în timpul asamblării, fiecare apel al unei macroinstrucţiuni este înlocuit prin corpul macroinstrucţiunii şi cele două pseudo-instrucţiuni sunt eliminate.

Exemplu: Calculul cubului unui număr:

```
MACRO   CUB (val, valcub)
  MOVE  val, D1
  MOVE  val, D2
  MUL           D1, D2  (D2:= D1 × D2)
  MUL           D1, D2
      MOVE      D2, valcub
  ENDM
```

Subprograme

Subprogramele sunt definite ca şi macroinstrucţiunile, având de asemenea scopul de a evita repetarea scrierii unei secvenţe de instrucţiuni ce va fi utilizată de mai multe ori.

O diferenţă esenţială faţă de macroinstrucţiune rezidă în faptul că instrucţiunile care compun subprogramul constituie o entitate bine separată de programul principal, iar această separare se păstrează şi după traducere, încât subprogramul se găseşte o singură dată în memorie şi doar la execuţia programului se satisfac referinţele la un subprogram al său.

Această manieră de abordare a programării conduce la avantaje suplimentare faţă de utilizarea macroinstrucţiunilor, deoarece permite minimizarea taliei codului executabil, ceea ce nu este cazul macroinstrucţiunilor.

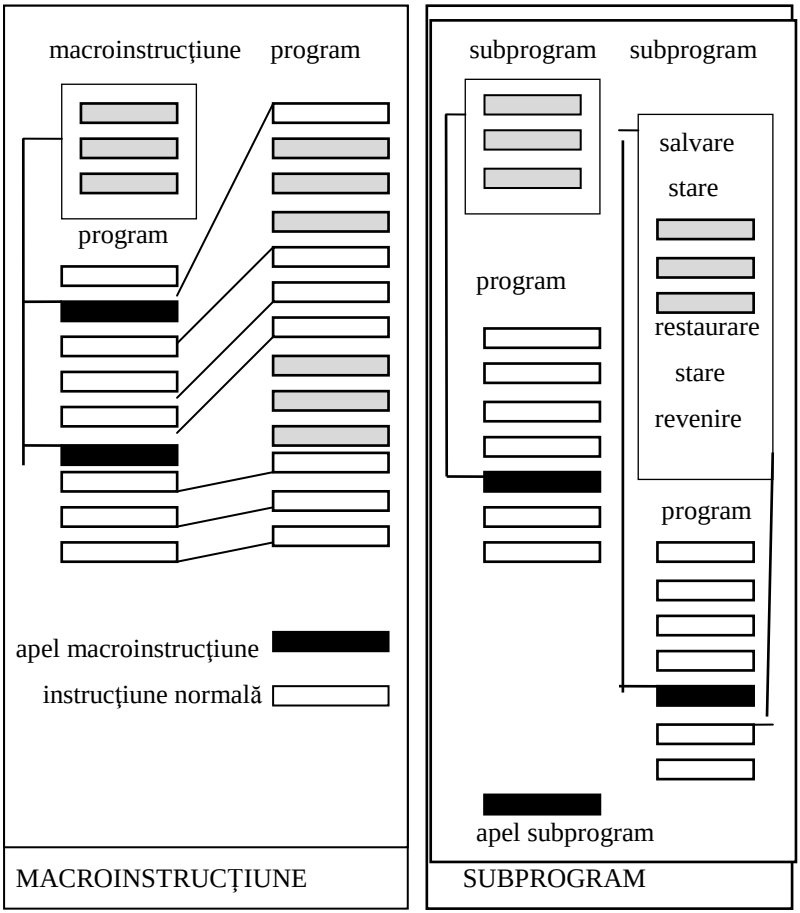


Figura anterioară ilustrează diferenţa dintre o macroinstrucţiune şi un subprogram.

Problemele care se manifestă în situaţia utilizării subprogramelor se referă în special la asigurarea salvării **adresei de revenire** în timpul execuţiei unui subprogram. Adresa de revenire este adresa instrucţiunii care urmează după instrucţiunea de apel a subprogramului.

Deci, principala diferenţă între o macroinstrucţiune şi un subprogram este aceea că apelurile la o macroinstrucţiune sunt înlocuite prin corpul macroinstrucţiunii în timpul traducerii, pe când apelurile la un subprogram sunt tratate în timpul execuţiei.

Transmiterea parametrilor

Un program poate să schimbe date cu macroinstrucţiunile şi subprogramele sale prin intermediul parametrilor. Un **parametru** este o variabilă al cărei nume este cunoscut, dar al cărei conţinut nu este precizat decât în momentul execuţiei. O macroinstrucţiune sau un subprogram se scrie utilizând **parametrii formali** care vor fi înlocuiţi prin **parametrii efectivi** corespunzători datelor reale care sunt tratate**.**

Substituţia parametrilor formali cu cei efectivi se realizează la traducere, pentru macroinstrucţiuni, şi la execuţie, pentru subprograme.

Pentru subprograme există mai multe tehnici de transmitere a parametrilor:

a) **Transmiterea prin valoare** constă în recopierea valorii de transmis într-o zonă cunoscută de subprogram, care poate fi o zonă de memorie sau un registru. În acest fel, subprogramele acţionează asupra unei copii a parametrilor, orice modificare a parametrului nu este posibilă decât în interiorul subprogramului, iar la revenire în programul apelant parametrul regăsește valoarea sa inițială, ceea ce permite o anumită protecţie a parametrilor.

b) **Transmiterea prin referinţă** constă în a transmite către subprogram adresele parametrilor. Subprogramul lucrează deci efectiv asupra datelor programului apelant, deci orice modificare a valorii parametrului în interiorul subprogramului este reflectată la revenire în programul apelant.

Salvarea stării maşinii

Salvarea stării maşinii constă în a salva starea registrelor CPU (în alte registre CPU prevăzute în acest scop sau în memoria centrală) pentru a permite executarea unui alt program şi la sfârşitul execuţiei acestuia, reluarea execuţiei primului.

Spre deosebire de macroinstrucţiuni, unde **expandarea** (înlocuirea apelului prin corpul macroinstrucţiunii) se realizează la traducere, în cazul subprogramelor este necesară salvarea stării maşinii la apelul acestora.

Într-adevăr, apelul unui subprogram constă în trecerea controlului CPU-ului acestui subprogram, care se comportă ca un program independent şi care trebuie să fie capabil să redea controlul programului apelant la sfârşitul execuţiei sale. Primele instrucţiuni ale subprogramului servesc la salvarea stării diverselor registre, iar ultimele instrucţiuni restaurează aceste valori salvate, revenind la contextul anterior apelului.

Recursivitate

Un subprogram este recursiv dacă poate să se autoapeleze (direct sau indirect).

Problema unui subprogram recursiv este că în timpul fiecărui apel, trebuie salvată starea maşinii. Datorită faptului că nu se poate utiliza aceeaşi zonă de memorie pentru salvare, se adoptă o soluţie bazată pe utilizarea unei stive care lucrează după principiul LIFO [Last In, First Out], în care se salvează starea maşinii pe măsura activării apelurilor recursive (este necesară o condiţie de sfârşit pentru apelurile recursive).

4.2 Limbaje de programare

Un limbaj este o modalitate de exprimare şi comunicare a gândurilor. Există o multitudine de limbaje: limbaje orale, scrise, limbajul semnelor (de exemplu, limbajul surdo-muţilor) şi multe alte limbaje care utilizează diverse moduri de transmisie (de exemplu, limbajul albinelor).

Un limbaj informatic cuprinde:

- **un alfabet**: mulţimea simbolurilor elementare disponibile;
- **nume** s**au identificatori**: grupe de simboluri ale alfabetului (cu anumite restricţii: număr de caractere, tipul primului caracter etc.);
- **fraze** sau **instrucţiuni**: secvenţe de nume şi simboluri de punctuaţie care respectă aspectele sintactice şi semantice ale limbajului.

Limbajele informatice, spre deosebire de limbajele naturale, sunt structurate, riguros neambigue şi pot fi definite în mod formal.

Limbajele de asamblare au fost primele limbaje informatice şi ele depind de arhitectura sistemelor de calcul.

Limbajele evoluate [HLL: High Level Languages] au apărut mai târziu şi permit comunicarea cu sistemul de calcul fără a ţine seama de arhitectura sa.

În informatică se disting mai multe categorii de limbaje: **limbajele de programare** şi **limbajele de comandă** sunt cele mai utilizate, dar există de

asemenea şi **limbaje de analiză** şi **limbaje de specificare**, care ajută în timpul primelor faze de dezvoltare a produselor software.

Scrierea unui program se realizează prin utilizarea simbolurilor limbajului de programare pentru constituirea frazelor sau instrucţiunilor care trebuie să respecte **sintaxa** limbajului.

Cele două modalităţi de reprezentare a sintaxei unui limbaj sunt:

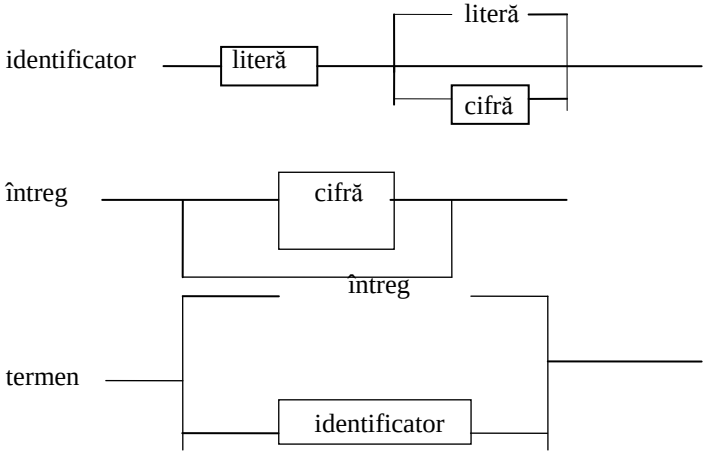
- **notaţia BNF** [Backus Naur Form];
- **diagramele sintactice**.

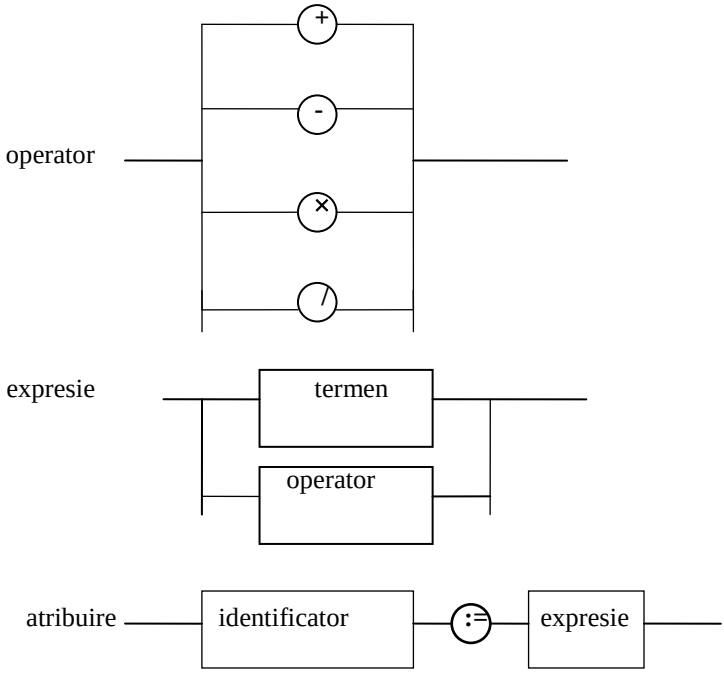
Prezentăm în continuare un exemplu de sintaxă a unui limbaj foarte simplu care permite definirea identificatorilor (încep întotdeauna cu o literă), întregii fără semn, expresiile aritmetice simple şi instrucţiunea de atribuire.

Pentru notaţia BNF, semnul **“** | **“** indică o alternativă, semnele **“**< **“** şi **“** > **“** delimitează obiectele limbajului şi semnul **“** := **“** indică atribuirea.

< literă > ::= a | b | c | d . . .y | z
< cifră > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
< identificator > ::= < literă > | <identificator > < literă > |
| < identificator > < cifră >
< întreg > ::= < cifră > | < întreg > < cifră >
< termen > ::= < întreg > | < identificator >
< operator > ::= + | - | × | /
<expresie > ::= < termen > |
| <termen> < operator><expresie>
< atribuire > ::= < identificator > := < expresie >

Utilizând diagramele sintactice, prezentăm acelaşi exemplu de definire a sintaxei limbajului:





Concepte de bază ale limbajelor evoluate

Limbajele evoluate permit uşurarea activităţii de programare printr-o apropiere de limbajul natural, dar cu respectarea exigenţelor de rigoare şi neambiguitate.

Primele concepte de bază ale limbajelor de programare evoluate se refereau la independenţa faţă de sistemul de calcul, ceea ce a permis elaborarea unor instrucţiuni posedând un nivel semantic superior limbajului de asamblare.

Elaborarea de limbaje de programare evoluate a vizat încă de la început trei direcţii importante, şi s-a concretizat prin trei categorii de limbaje:

a) limbaje bazate pe conceptele de algoritm şi prelucrarea datelor cu caracter ştiinţific (**Fortran**, **Algol**);
b) limbaje bazate pe prelucrarea datelor (**Cobol**);
c) limbaje bazate pe prelucrarea listelor (**Lisp**).

Limbajele din primele două categorii sunt **limbaje procedurale** care furnizează o descriere detaliată a algoritmilor de rezolvare a problemelor, iar limbajele din a treia categorie sunt **limbaje funcţionale**, care subliniază aspectul funcţional al rezolvării unei probleme, fără a intra în detalii.

Orientarea algoritmică a permis în continuare dezvoltarea limbajelor **Algol 60**, **Algol 68**, **Fortran II**, **Fortran IV**, **Fortran 66**, **Fortran 77**, **Fortran 90**, **Pasca**l, **C**, **Modula-2**, **Ada** şi **C++**, care furnizează metodologii bazate pe conceptele de **programare structurată**, **abstractizare**, **modularitate** etc.

A doua orientare, bazată pe prelucrarea datelor a rămas fidelă limbajului **Cobol**, dar a evoluat spre **sisteme de gestiune de baze de date** care permit rezolvarea problemelor specifice bazelor de date.

Limbajul **Prolog**, instrument de bază în domeniul **inteligenţei artificiale**, a rezultat din abordarea funcţională şi aduce nou, în afară de prelucrarea listelor, un mecanism de inferenţă foarte util pentru realizarea sistemelor expert.

Asistăm astăzi la o oarecare convergenţă între **limbajele orientate obiect** şi **bazele de cunoştinţe**, ceea ce permite gruparea şi unificarea conceptelor de bază ale diverselor orientări.

Vom prezenta în continuare o scurtă descriere pentru câteva limbaje de programare dintre cele mai utilizate:

- **Fortran** [Formula Translator] este primul limbaj algoritmic, utilizat în principal pentru rezolvarea problemelor cu caracter ştiinţific. El produce cod eficace, utilizează o mare cantitate de biblioteci matematice şi a introdus concepte importante, ca de exemplu, structurarea expresiilor aritmetice, subprograme, compilarea independentă a subprogramelor. Limbajul a evoluat mereu, versiunea **Fortran 90** permite programarea paralelă şi concurentă specifică supercalculatoarelor.
- **Cobol** [Common Business Oriented Language] este un limbaj destul de utilizat în lume, şi este adaptat aplicaţiilor de gestiune care permit un acces uşor la fişiere şi baze de date. Inconvenientele utilizării sale provin din aspectul **"**stufos**"** al scrierii şi dificultatea de structurare a programelor.
- **Algol** [Algorithmic Oriented Language] este un limbaj care a avut o influenţă primordială asupra limbajelor actuale. Definit pentru aplicaţii ştiinţifice, limbajul nu a reuşit să se impună din cauza complexităţii sale, lipsei de soliditate a intrărilor/ieşirilor şi lipsa eficacităţii. Totuşi, **Algol** a introdus conceptul de **structurare**, cu structuri de blocuri, structuri de control, proceduri, recursivitate. Este primul limbaj definit în notaţia BNF şi a evoluat în **Algol 60**, **Algol 68**, **Algol W** dar nu a reuşit să părăsească mediul academic.
- **Lisp** [List Processing] a fost conceput pentru manipularea expresiilor şi funcţiilor simbolice. Caracteristicile sale sunt: capacitatea de tratare a listelor, un număr mic de operatori de bază,

un număr mare de paranteze şi recursivitatea care joacă un rol de seamă în parcurgerea listelor. Limbajul este utilizat curent în inteligenţa artificială. Datorită lipsei de eficacitate pe sisteme de calcul tradiţionale, anumiţi constructori au dezvoltat maşini **Lisp**, care acceptă acest limbaj ca limbaj maşină, având o arhitectură particulară, bazată pe noţiunea de stivă.

- **Basic** [Beginner's All-purpose Symbolic Instruction Code] este un limbaj foarte rudimentar, care a fost dezvoltat doar în scop didactic. El a devenit un limbaj foarte răspândit datorită dezvoltării microcalculatoarelor care, la început nu aveau capacitatea de a utiliza alte limbaje.
- **PL/1** [Programming Language number 1] este o realizare ambiţioasă, un limbaj aproape universal, menit să înlocuiască limbajele **Fortran**, **Algol** şi **Cobol** utilizate în acea epocă. Din cauza complexităţii sale, lipsa de omogenitate şi de rigoare limbajul nu a cunoscut o mare răspândire, limitându-se la calculatoarele din familia IBM.
- **Pascal**, care poartă numele matematicianului francez creator al uneia dintre primele maşini aritmetice de calcul din secolul XVII, a fost dezvoltat de către elveţianul Niklaus Wirth. Acest limbaj provenit din **Algol** căruia îi preia conceptele, este bazat pe o mare simplitate şi este destinat înainte de toate învăţării programării.
- **Modula-2** este încă un limbaj dezvoltat de Wirth. Limbajul este un descendent al limbajului Pascal, căruia îi adaugă noţiunea de **modularitate** care permite compilarea independentă a modulelor.
- **Smalltalk** este un limbaj care caracterizează o nouă tendinţă de programare bazată pe conceptul de **obiect**. El permite o interacţiune grafică cu sistemul, bazată pe utilizarea ferestrelor şi a meniurilor, ceea ce constituie o inovaţie în domeniul programării.
- **Prolog** [Programmation Logique] (elaborat în 1972 de către Colmerauer) este un limbaj care preia conceptele limbajului **Lisp**, adăugând un mecanism de inferenţă utilizat la realizarea sistemelor expert. Japonezii şi-au pus mari speranţe în acest limbaj pentru proiectul de sisteme de calcul de generaţia a cincea.
- **C**, un succesor al limbajelor **BCPL** şi **B**, este un limbaj orientat spre programare de sistem. Succesul limbajului se datorează utilizării sale pentru dezvoltarea sistemului de operare UNIX. **C** este un limbaj relativ simplu la nivel de concepte, codul generat de compilator este eficace, dar sintaxa permite scrierea într-o linie a instrucţiunilor foarte complexe, care devin aproape ilizibile.

- **Ada**, al cărui nume aduce omagiu Adei Byron (sec. XIX), considerată ca prima informaticiană, este un limbaj conceput pentru departamentul apărării al SUA. El preia conceptele limbajelor **Pascal** şi **Modula-2**, adăugând concepte de timp real, paralelism, genericitate şi gestiunea excepţiilor.
- **C++** este un succesor al limbajului **C**, dezvoltat de către Bjarne Stroustrup la începutul anilor **'**80. Acest limbaj poate fi văzut ca o evoluţie naturală a limbajului C, reia conceptele acestui limbaj, la care se adaugă un anumit număr de concepte legate de programarea orientată obiect, ca de exemplu, noţiunea de **clasă de obiecte**, **moştenire între clase** etc.

Orientarea obiect

Evoluţia programării clasice se poate rezuma în felul următor:

a) **Programarea procedurală** [procedural programming]: accentul este pus pe algoritmi, codul este repartizat în proceduri, iar structurile de date nu sunt luate în considerare. Este cazul limbajelor **Fortran** sau **C**.

b) **Încapsularea şi abstractizarea datelor** [data hidding and abstraction]: apare noţiunea de **modul**, care permite descompunerea codului în diferite module. Datele sunt închise în interiorul modulelor ceea ce constituie **tipurile abstracte de date** [ADT: Abstract Data Type]. Practic, un ADT poate fi văzut ca o **"**cutie neagră**"**, iar seviciile (**metode)** sunt oferite prin intermediul unei interfeţe. Este cazul limbajelor **Modula-2** şi **Ada**.

c) **Programarea orientată obiect** [object oriented programming]: reia conceptele de tipuri abstracte de date dar se insistă asupra noţiunii de **reutilizare** a obiectelor sistemului**.** În acest sens, se determină principalele modele de date necesare, care se vor numi **clase de obiecte** şi în continuare se stabilesc **metodele** care vor manipula aceste modele. Un program se compune deci dintr-un ansamblu de obiecte care interacţionează între ele trimiţându-şi **mesaje** care activează metodele specifice fiecărei clase de obiecte. Este cazul limbajelor **Smalltalk**, **Eiffel**, **C++**, **Objective-C**.

Tipuri abstracte de date

Un tip abstract de date permite descrierea unei clase de structuri de date prin lista funcţiunilor (**metode**) disponibile asupra structurilor de date şi nu prin implementarea lor.

Clase şi obiecte

O clasă de obiecte corespunde implementării unui tip abstract de date. Definiţia unei clase descrie comportamentul tipului său abstract prin specificarea interfeţei tuturor operaţiilor (metode) care pot fi aplicate asupra tipului abstract.

Definiţia unei clase comportă de asemenea detalii cu privire la implementarea unor astfel de structuri de date sau codul sursă care implementează metodele.

Un **obiect** [object] este o variabilă a cărei tip este o clasă. El reprezintă o instanţă (realizare) a unei clase. Acţiunile pot fi aplicate asupra acestui obiect invocând metodele definite în clasă, care se realizează printr-un procedeu numit **trimitere de mesaje** obiectelor clasei.

Clasele sunt entităţi definite în codul sursă al unui program, ele descriu de o manieră statică o mulţime de obiecte (pot fi considerate şi ca tipuri de date), în timp ce obiectele sunt elementele dinamice, ele nu există decât la execuţie şi corespund instanţelor clasei.

Mesaje

Obiectele pot comunica între ele trimiţându-şi **mesaje** prin care se solicită efectuarea unei anumite operaţii asupra acestor obiecte. Mulţimea tipurilor de mesaje proprii unui obiect corespunde interfeţei sale.

Polimorfism

Polimorfismul permite efectuarea unei acţiuni prin trimiterea unui mesaj la un obiect pentru care sunt posibile mai multe instanţe de execuţie. Această capacitate este foarte importantă atunci când acelaşi mesaj poate fi îndeplinit în moduri diferite pentru tipuri diferite de obiecte.

Limbajele orientate obiect permit trimiterea mesajelor identice spre obiecte care aparţin unor clase diferite (dar derivate din clasa de bază).

Polimorfismul constă în esenţă în posibilitatea asocierii mai multor implementări ale aceluiaşi mesaj, iar sistemul de calcul trebuie să fie în măsură să stabilească implementarea corespunzătoare mesajului transmis.

Acestă decizie (constând în legarea mesajului de implementarea corespunzătoare) se poate lua fie la compilare [early sau static bilding], fie la execuţie [late sau dynamic bilding].

Moştenirea şi ierarhizarea claselor

Prin mecanismul moştenirii [inheritance], programarea orientată obiect permite definirea **subclaselor**. O subclasă, numită şi **clasă derivată** permite caracterizarea comportamentului unei mulţimi de obiecte care moştenesc

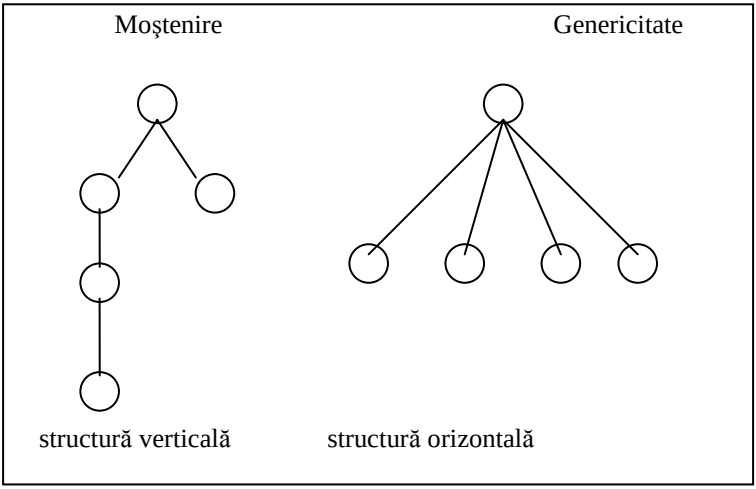
caracteristicile clasei lor **"**părinte**"**, dar care poate de asemenea să posede caracteristici particulare proprii, pe care **"**părintele**"** nu le are.

Utilizarea subclaselor permite diminuarea costului şi complexităţii programelor, deoarece subclasele facilitează reutilizarea claselor existente, permiţând în acelaşi timp modificarea lor.

Genericitate

Genericitatea se exprimă prin capacitatea de a defini clase **parametrizabile**. De exemplu, presupunem că este necesară o stivă de numere întregi şi o stivă de numere reale. În locul definirii celor două tipuri de stivă se va proceda la definirea unei clase parametrizată numită **stivă**, din care se vor genera cele două clase dorite.

Genericitatea şi moştenirea corespund unor necesităti diferite şi generează structuri diferite: moştenirea favorizează rafinamente succesive ale unei aceeaşi clase rezultând astfel o structură pe verticală, în timp ce genericitatea permite definirea unei clase de bază parametrizată care se poate instanţia de mai multe ori cu tipuri diferite, rezultând astfel o structură pe orizontală. Figura următoare evidenţiază această situaţie.



Inteligenţa artificială şi sistemele expert

Inteligenţa artificială este domeniul informaticii care propune simularea pe sistemele de calcul a comportamentului inteligent al omului. Sunt implicate domeniile perceperii, înţelegerii, luării deciziilor, învăţării.

Această orientare a condus la unele rezultate notabile, în special în unele domenii: teoria jocurilor, demonstrarea teoremelor, recunoaşterea formelor, recunoaşterea parolelor, înţelegerea limbajelor naturale, rezolvarea problemelor care necesită expertiză legată de un domeniu specific (de exemplu, diagnosticul medical), matematici simbolice etc.

Inteligenţa artificială trebuie să permită rezolvarea problemelor pentru care abordarea algoritmică este ineficientă sau chiar imposibil de aplicat. Un program al inteligenţei artificiale se caracterizează prin utilizarea simbolurilor în locul informaţiilor alfanumerice.

Sisteme expert

Sistemele expert constituie cu siguranţă domeniul inteligenţei artificiale care a cunoscut cea mai mare dezvoltare. Un **sistem expert** este un program care utilizează intensiv **cunoştinţa** în scopul rezolvării problemelor care necesită în mod normal expertiza umană.

Într-un sistem expert există o separaţie netă între programe şi cunoştinţe. Arhitectura de bază a unui sistem expert cuprinde trei părţi:

a) **Baza de fapte** este un fel de bază de date care regrupează faptele şi aserţiunile vizând problema tratată;
b) **Baza de reguli** conţine cunoştinţele care permit manipularea faptelor din baza de fapte. Cunoştinţele se exprimă sub formă de **reguli de producţie**. O regulă comportă o parte stângă, exprimând o condiţie (**dacă**) şi o parte dreaptă conţinând concluzii (**atunci**);
c) **Motorul de inferenţă** exploatează **baza de cunoştinţe** (baza de fapte + baza de reguli) asociind faptele şi regulile pentru a formula un raţionament asupra problemei puse. Pentru aceasta, pornind de la baza de fapte, el determină mulţimea regulilor a căror parte stângă este verificată, faptele conţinute în partea dreaptă adăgându-se la baza de fapte. În continuare, motorul de inferenţă aplică aceste reguli (**înlănţuire înainte**) pentru a ajunge la o concluzie, procesul oprindu-se când nu se mai pot genera fapte noi. Se poate de asemenea porni de la concluzie, inferenţele propagându-se invers (**înlănţuire înapoi**). Ele determină noi **subscopuri** mai simplu de verificat până la găsirea părţilor stângi ale regulilor corespunzătoare faptelor din baza de fapte. Un anumit număr de limbaje (**Lisp** şi **Prolog**) permit dezvoltarea cu uşurinţă a sistemelor expert simple.

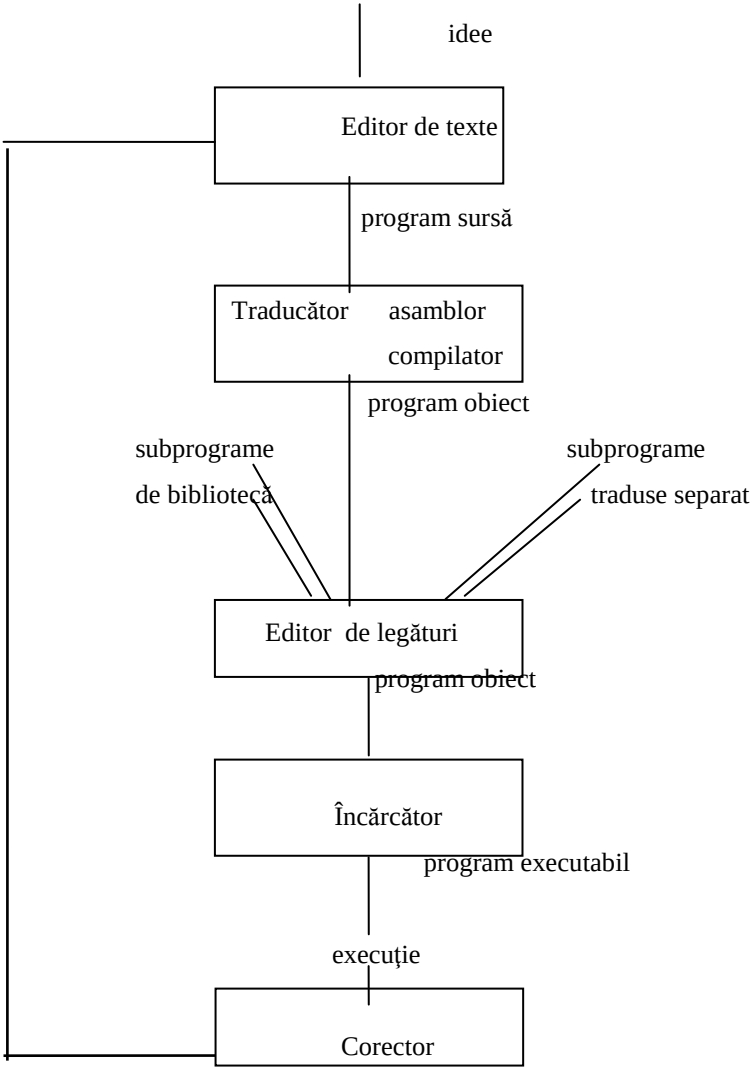
5.3 Etapele dezvoltării unui program

Dezvoltarea unui program, de la analiza problemei până la punerea sa la punct, necesită numeroase instrumente software constituite într-un **mediu**

de programare, care utilizează serviciile sistemului de operare, în special sistemul de gestiune al fişierelor.

Elementele clasice ale unui mediu de programare sunt următoarele: editorul de texte, traducătorul (compilator sau asamblor), editorul de legături, programul încărcător şi programul corector.

Figura următoare prezintă grafic un mediu de programare minim:



Editorul de texte

Un editor de texte [text editor] este un program interactiv care permite preluarea unui text pornind de la tastatură şi stocarea sa într-un fişier. Informaţiile conţinute în fişier sunt de tip text, adică o mulţime de caractere, în general structurate în linii.

Principalele funcţiuni ale unui editor de texte sunt:

- vizualizarea unei părţi a textului;
- deplasarea şi poziţionarea în fişier (se indică poziţia curentă);
- modificarea textului prin inserare, modificare, ştergere;
- regăsirea şirurilor de caractere particulare.

Editorul poate fi utilizat atât pentru tastarea textului sursă al unui program cât şi pentru introducerea datelor necesare programului.

Vom evidenţia două dintre cele mai importante tipuri de editoare:

a) **editor sintactic**, adaptat prelucrării programelor sursă, care verifică sintaxa programelor pe măsura tastării acestora, permiţând de asemenea gestionarea automată a structurilor sintactice proprii limbajului de programare utilizat;
b) **procesor de texte**, destinat tratării textelor. Acest tip de editor oferă funcţionalităţi mult mai pronunţate pentru manipularea caracterelor: utilizarea literelor accentuate, fonturi diferite (adică diferite seturi de caractere), alinierea textului, inserare de desene etc., mai general se zice că aceste editoare oferă facilităţi de **punere în pagină** pentru scriere de scrisori, articole, cărţi etc.

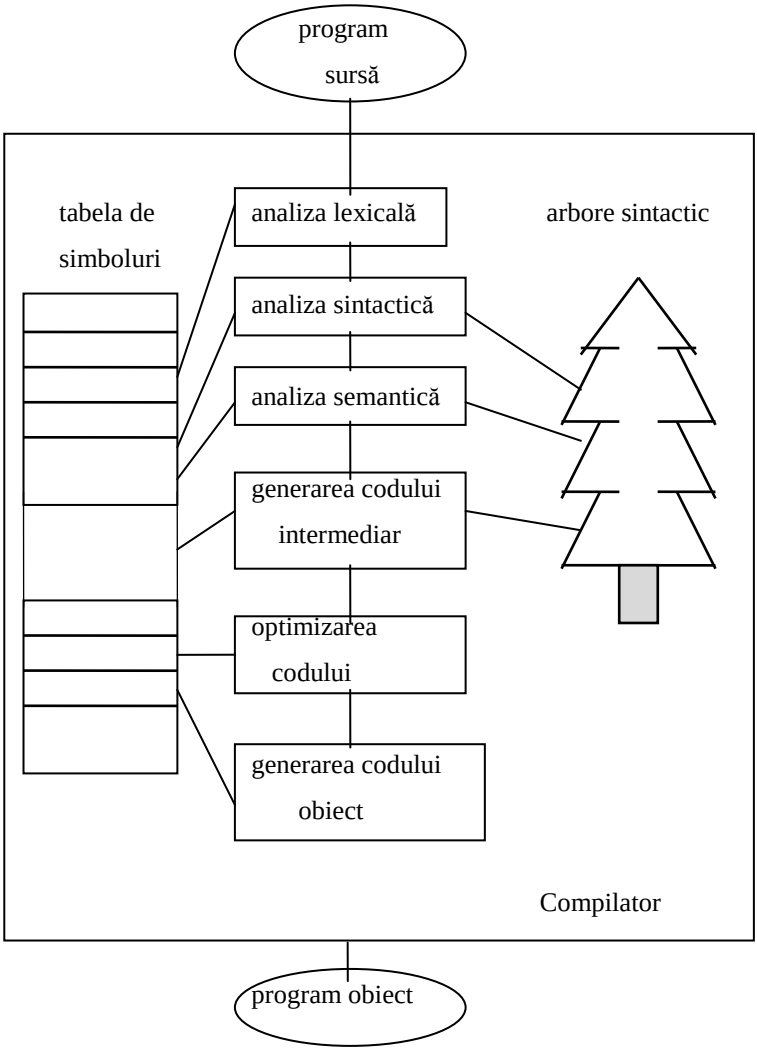
Compilatorul

Un compilator este un program sistem care traduce un program sursă scris într-un limbaj de programare de nivel înalt în program obiect.

Activitatea compilatorului se divide în două mari componente:

a) **faza de analiză**, care cuprinde următoarele etape:
   - **analiza lexicală**;
   - **analiza sintactică**;
   - **analiza semantică**.
b) **faza de sinteză**, care cuprinde următoarele etape:
   - **generarea codului intermediar**;
   - **optimizarea codului**;
   - **generarea codului obiect**.

Activitatea compilatorului este prezentată schematic astfel:



Analiza lexicală

Analiza lexicală este prima fază a compilării. Rolul său major constă în citirea secvenţei de caractere care constituie programul sursă şi producerea unor secvenţe de elemente sintactice ale limbajului.

Identificatorii, de exemplu numele de variabile, sau de proceduri ca şi atributele lor sunt stocate într-o tabelă de simboluri, în timp ce informaţiile inutile pentru compilator (comentariile) sunt eliminate.

Analiza sintactică

Analizorul sintactic primeşte o listă de elemente sintactice (cuvinte rezervate, identificatori, operatori aritmetici, semne de punctuaţie etc.) elaborată de către analizorul lexical.

El verifică dacă această listă este corectă în raport cu sintaxa limbajului şi pornind de la aceste elemente, analizorul sintactic generează arborele sintactic al programului.

Sunt posibile două abordări pentru generarea arborelui sintactic:

- **ascendent**ă: se analizează elementele componentr ale frazei de tratat şi se caută regulile care permit ascensiunea spre rădăcină;
- **descendentă**: se porneşte de la rădăcină şi se aplică regulile care permit construirea frazei dorite.

Analiza semantică

Analiza semantică se ocupă de analiza sensului şi a semnificaţiei frazelor limbajului, utilizând arborele sintactic pentru a identifica operatorii şi operanzii instrucţiunilor.

Sarcina principală a unui analizor semantic este verificarea concordanţei tipurilor, ceea ce revine la a verifica dacă fiecare operator acţionează asupra operanzilor care sunt autorizaţi prin limbaj. Pentru efectuarea acestor verificări, analizorul semantic utilizează informaţiile care sunt stocate în tabela de simboluri.

Generarea codului intermediar

După etapele fazei de analiză, se procedează la generarea programului în cod obiect. O metodă răspândită constă în divizarea acestei sarcini în două etape: generarea codului intermediar şi generarea codului obiect.

Codul intermediar se poate defini ca un cod al unei maşini abstracte, care trebuie să posede două proprietăţi: să fie uşor de generat pornind de la arborele sintactic al unui program şi să fie uşor de tradus în cod obiect.

Deci, pornind de la arborele sintactic al unui program, compilatorul generează un flux de instrucţiuni simple care se aseamănă cu macroinstrucţiunile, dar contrar asamblorului, acestea nu fac referire explicită la registrele sistemului de calcul.

Optimizarea codului

Optimizarea codului constă în ameliorarea codului pentru a-l face mai rapid de executat şi mai puţin **"**încurcat**"** în memorie, şi vizează în special eliminarea redondanţelor şi evaluarea expresiilor care utilizează constante.

Deoarece optimizarea conduce la o creştere substanţială a timpului de compilare, este de preferat evitarea acestei faze în timpul dezvoltării şi punerii la punct a programelor.

Optimizarea codului joacă un rol determinant pentru sistemele de calcul care utilizează un procesor RISC, datorită complexităţii compilatoarelor pentru astfel de maşini, care utilizează un număr mare de registre în vederea reducerii numărului de accese la memoria centrală.

Generarea codului obiect

Generarea codului obiect este fază finală a compilării, care generează codul obiect relocabil, adică relativ la originea 0. Fiecare instrucţiune a codului intermediar este tradusă într-o secvenţă de instrucţiuni în cod obiect. Generarea codului obiect atribuie poziţii în memorie pentru datele şi instrucţiunile programului.

Tabela de simboluri

În timpul compilării este necesară descrierea identificatorilor şi a caracteristicilor acestora. **Tabela de simboluri** permite gruparea acestor informaţii care sunt puse la dispoziţia diferitelor faze ale compilatorului.

În tabelă se găsesc numele variabilelor, constantelor şi procedurilor. Fiecărei intrări din tabelă i se asociază o înregistrare care conţine numele obiectului considerat şi caracteristicile proprii (tip, adresă numerică, numărul şi tipul parametrilor etc.). Accesul la această tabelă trebuie să fie rapid, deoarece toate fazele compilării pot utiliza tabela de simboluri.

Tratarea erorilor

În activitatea practică de programare s-a constatat că este dificilă scrierea programelor fără erori şi din această cauză, un bun compilator trebuie să facă posibilă detectarea şi corectarea acestor erori.

La sfârşitul compilării se întocmeşte un raport al erorilor depistate şi se încearcă specificarea cauzei care a generat eroarea precum şi poziţia sa în cadrul textului sursă.

Un program scris într-un limbaj de pogramare evoluat poate conţine erori de natură diferită:

a) **erori lexicale**: erori de ortografiere a identificatorilor sau cuvintelor rezervate, caractere ilegale etc.

b) **erori sintactice**: constituie majoritatea erorilor de programare şi se referă la: expresii aritmetice sau logice incorecte, erori de structurare a blocurilor, lipsa separatorilor etc.
c) **erori semantice**: identificatori nedeclaraţi, incompatibilitate între operatori şi operanzi etc.
d) **erori logice**: erori aritmetice de tipul împărţirii cu zero, rădăcină pătrată dintr-un număr negativ, depăşirea limitelor unui tablou, ciclu infinit, adresare incorectă etc.

Erorile din primele trei categorii sunt detectate de compilator în timpul analizei cu acelaşi nume (analiza lexicală, sintactică, semantică), şi sunt relativ uşor de corectat deoarece compilatorul indică prezenţa lor.

Erorile logice sunt vizibile doar la execuţie şi au ca efect fie efectuarea unor calcule eronate, fie oprirea execuţiei programului. Acestea sunt erorile cele mai dificil de detectat şi corectat, şi la fiecare tentativă de corectare trebuie reluat ciclul compilare - execuţie - testare.

Editorul de legături

Un editor de legături [linker, linkage editor] este un produs software care permite combinarea mai multor programe obiect, obţinându-se un singur modul obiect.

Dezvoltarea unor programe complexe se realizează prin descompunerea acestora în module care se traduc independent, deci un program poate fi constituit din mai multe fişiere (sau subprograme), unul dintre fişiere conţinând în mod obligatoriu programul principal.

Toate aceste fişiere sunt traduse separat, ele pot utiliza subprograme care se găsesc în alte fişiere, ceea ce dă naştere la **referinţe exterioare**.

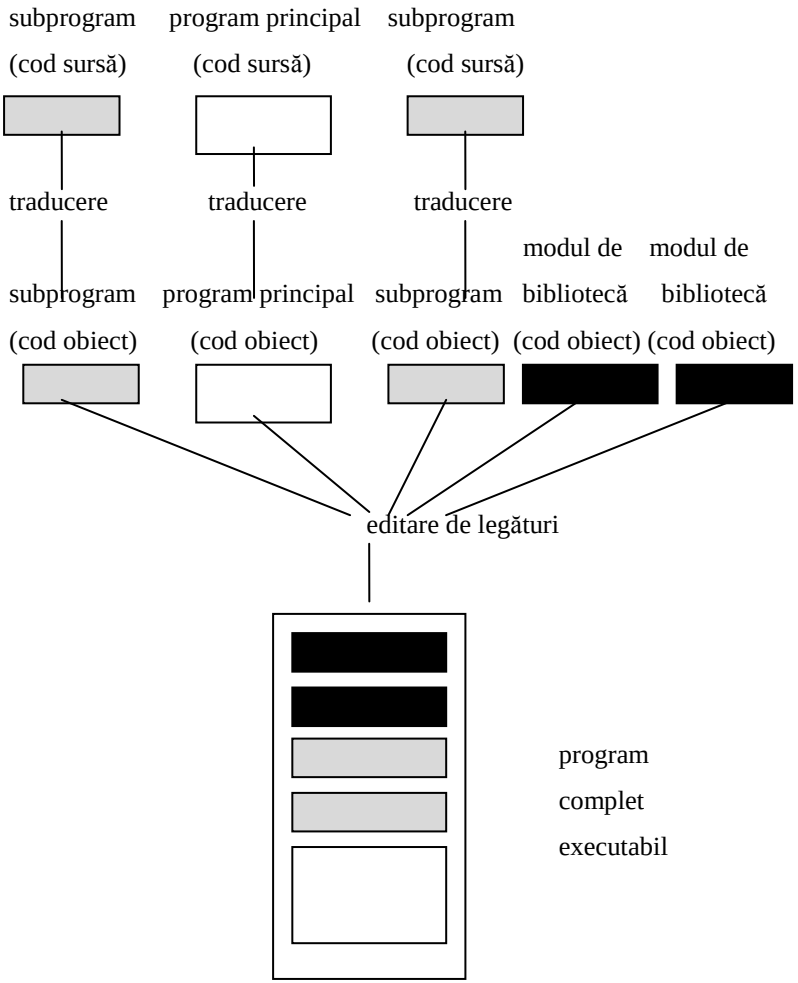
Există două tipuri de referinţe exterioare:

a) posibilitatea ca un modul să apeleze alt modul sau subprogram de bibliotecă;
b) posibilitatea ca un modul să fie referit de un alt modul;

Referinţele exterioare unui modul ridică probleme deosebite pentru programul traducător care nu poate satisface referinţele exterioare ci doar întocmeşte o listă a acestora pe care o transmite editorului de legături.

Editorul de legături preia modulele independente, le grupează şi satisface toate referinţele exterioare pentru a forma un program complet, care este executabil.

Figura următoare prezintă un exemplu de editare de legături pentru un program care conţine două subprograme traduse şi stocate în fişiere separate, şi care face de asemenea apel la două module de bibliotecă care sunt traduse în prealabil şi conservate în cod obiect.



Programul încărcător

Programul obiect care rezultă în urma editării de legături, trebuie să fie încărcat în memoria calculatorului pentru a putea fi executat. **Programul încărcător** [loader], care este de obicei cuplat cu editorul de legături realizează încărcarea programului obiect la adresa sa de încărcare.

Există două tipuri de programe încărcătoare:

a) **încărcător absolut**, specific vechilor sisteme de calcul care permite încărcarea programului (unic în memorie) la o adresă fixată în avans, ca şi toate adresele din cadrul programului.

b) **încărcător relocabil**, specific noilor tipuri de sisteme multiprogramate, care utilizează pentru încărcare în memorie **tehnica relocării.** O modalitate de realizare a relocării este aceea a utilizării indicatorului de relocare de către programul traducător în câmpul de adresă al instrucţiunii. Relocarea se mai poate realiza prin intermediul unui registru de bază, astfel:
   - se traduce programul în raport cu adresa 0 şi se realizează editarea de legături;
   - se alege un registru de bază printre cele disponibile;
   - se depune în registrul de bază adresa de bază (adresa absolută a programului);
   - se încarcă programul în memorie, pornind de la adresa de bază, fără modificarea adreselor programului. În timpul execuţiei, la fiecare referire a unei adrese, sistemul de calcul efectuează operaţiile:
     adresa efectivă = adresa de bază + adresa referită.