UNIVERSITATEA TEHNICĂ
TIMIȘOARA

ing. Horia CIOCÂRLIE

CERCETĂRI ÎN DOMENIUL LIMBAJELOR DE PROGRAMARE

contribuții la definirea și implementarea unor limbaje de programare concurentă

Conducător științific:

prof. dr. ing. Mircea PETRESCU
Institutul Politehnic București

Timișoara - 1991

Cuvînt înainte

Lucrarea pe care am elaborat-o ca şi teză de doctorat cuprinde rezultatele unei activităţi de peste un deceniu în domeniul limbajelor de programare. Am încercat, pe cît a fost posibil, să ţin pasul cu progresele spectaculoase înregistrate pe plan mondial, mai ales în ceea ce priveşte noile concepte încorporate în limbajele de programare ale ultimilor ani. Perfecţionarea tehnicilor de implementare n-a înregistrat aceaşi dinamică. O parte din aceste concepte au fost implementate prin extinderea sau modificarea unor tehnici tradiţionale. Au rămas însă pînă în prezent, suficiente probleme de implementare care nu au o formalizare riguroasă sau pentru care nu s-au găsit soluţii de rezolvare completă, derivînd direct din mecanismul de formalizare. Prin lucrarea de faţă am încercat să aduc o modestă contribuţie în această direcţie.

Retrospectiva cronologică a întregii activităţi pe care am desfăşurat-o trebuie să înceapă în 1977 cînd, la numai un an de la absolvirea facultăţii am fost angajat ca inginer la Centrul de calcul electronic al Institutului Politehnic Timişoara. Din acest motiv, primele mele mulţumiri se îndreaptă în mod firesc către prof. Aurel Soceneanţu, cel care m-a îndrumat la începutul activităţii mele ştiinţifice. Faptul că, după numai cîteva luni, mi-a acordat încrederea de a conduce un important colectiv de cercetare, condiţiile de muncă pe care le-a creat acestui colectiv, răbdarea de care a dat dovadă în primii doi ani, cînd nu se întrezăreau încă rezultate concrete, au fost esenţiale atît pentru formarea mea cît şi pentru finalizarea marilor proiecte începute atunci.

Aduc, de asemenea, omagiul meu memoriei regretatului profesor Alexandru Rogojan, unul din pionierii calculatoarelor în ţara noastră şi întemeietor al şcolii tehnice superioare româneşti în acest domeniu, cel care m-a condus la începutul activităţii de doctorat.

Am o profundă stimă şi recunoştinţă faţă de actualul conducător, prof.dr.ing. Mircea Petrescu de la Institutul Politehnic Bucureşti, sub îndrumarea căruia am elaborat, în întregime, această lucrare. Analiza atentă a materialului tezei, observaţiile şi recomandările formulate, îndemnurile cu care m-a susţinut în permanenţă, au fost decisive pentru finalizarea lucrării.

Aduc deosebite mulţumiri d-lui prof.dr.ing. Crişan Strugaru, pentru atenţia şi competenţa cu care a analizat materialul tezei ca şi pentru observaţiile, recomandările şi aprecierile făcute. De asemenea, îi sînt recunoscător pentru faptul că, în calitate de şef de catedră, m-a ajutat în repetate rînduri, în ultimii ani, în scopul încheierii activităţii de elaborare a lucrării.

Îi sînt recunoscător d-lui conf.dr.ing. Ioan Jurca pentru solicitudinea şi amabilitatea de care a dat dovadă în toate prilejurile, pentru maniera competentă în care a analizat teza şi pentru observaţiile şi aprecierile pe care le-a formulat.

Îi mulţumesc de asemenea, d-lui conf.dr.ing. Vladimir Creţu pentru discuţiile fructuoase pe care le-am purtat şi pentru permanentele încurajări pe care mi le-a adresat.

Trebuie să remarc cu acest prilej şi faptul că o parte din realizările raportate în această lucrare, gîndite şi coordonate de mine, sînt totuşi realizări în colectiv. O contribuţie mai mare sau mai mică la proiectarea şi finalizarea programelor respective au avut toţi aceia cu care am conlucrat de-a lungul acestor ani: cadre didactice de la Catedra de calculatoare, cercetători de la Centrul de calcul al institutului sau studenţi. În mod cu totul deosebit îi mulţumesc colegului şi prietenului apropiat, şef lucr.ing. Petru Eleş, cu care am colaborat în ultimii ani, în munca de cercetare pe care am desfăşurat-o.

Sînt profund îndatorat întregului colectiv de soft al catedrei noastre pentru sprijinul, înţelegerea şi îndemnurile constante cu care m-au încurajat.

Mulţumesc de asemenea, d-lor ing. Radu Dragomir şi ing. Dan Manea pentru sprijinul preţios pe care mi l-au acordat la redactarea şi editarea lucrării.

În final, mulţumesc familiei şi în special soţiei mele, atît pentru încăduinţa de care a dat dovadă în această perioadă cît şi pentru sprijinul efectiv la finalizarea unor cercetări şi la redactarea tezei.

ing. Ciocârlie Horia

CUPRINS

INTRODUCERE

1. Programarea concurentă ca și concept

Cele mai multe dintre programele pe care le întîlnim în mod curent sînt *secvențiale*. Astfel de programe descriu *procese izolate* în care fiecare instrucțiune se activează în momentul în care predecesoarea ei s-a încheiat, conform fluxului de control al programului. În timpul rulării, un program secvențial utilizează diferite resurse ale calculatorului, cîte una la un moment dat. În măsura în care programul este singurul proces prezent în lucru, căruia îi sînt afectate toate resursele sistemului, aceasta conduce, evident, la utilizarea ineficientă a echipamentelor. De asemenea, nu s-ar putea profita de eventuala existență a mai multor procesoare fizice în scopul exploatării lor simultane. Chiar și în cazul imposibilității realizării unui *paralelism fizic*, din lipsă de procesoare, prezența mai multor procese simultan în memorie, cărora să li se asocieze alternativ procesorul, printr-o politică de gestiune oarecare, s-a dovedit deosebit de utilă. În acest caz se vorbește despre un *paralelism logic*, putîndu-se considera din punct de vedere abstract că procesele se execută în paralel. Spre exemplu, utilizatorii unor sisteme convenționale realizate după acest principiu, au impresia că totalitatea resurselor le stau la dispoziție în exclusivitate.

Considerentele de mai sus au determinat elaborarea sistemelor de operare cu *multiprogramare*. Acestea permit prezența simultană în memoria centrală a mai multor programe care se execută în paralel. Gradul de paralelism fizic obținut depinde de numărul și tipul procesoarelor fizice conectate la sistem. S-a putut constata că folosirea în comun de către mai multe programe aflate în memorie, a resurselor sistemului duce la o îmbunătățire spectaculoasă a gradului de utilizare al acestora.

Programele rulate sub sistemele cu multiprogramare formează, prin urmare, un set de *procese paralele* executate independent între ele. Comportarea lor individuală este practic identică celei din cazul rulării fiecăruia într-un sistem monoprogramat. Eventualele conflicte rezultate din utilizarea comună a unor resurse se rezolvă prin sistemul de operare, fără ca programatorul să ia vreo măsură în acest sens.

Programele secvențiale, reprezentînd descrierea unor procese izolate, nu permit rezolvarea unor anumite categorii de probleme. Deseori devine necesară prezența simultană a mai multor procese, executate în paralel și asincron, dar între care există relații de cooperare (schim de mesaje, transfer de date) și care gestionează în comun resurse ale sistemului. Relațiile dintre aceste procese sînt cu mult mai complexe decît cele, practic insesizabile pentru programator, dintre programele paralele din sistemele clasice cu multiprogramare. Ele reprezintă un set de *procese concurente* iar programul în care se descriu astfel de procese, precum și relațiile de comunicare și cooperare corespunzătoare, se numește *program concurent*.

Relațiile care apar între procesele concurente ale aceluiași program pot să fie deosebit de complexe și trebuie astfel implementate încît să se asigure atît o bună cooperare a lor cît și evitarea (rezolvarea) situațiilor conflictuale. Pentru aceasta trebuie avute în vedere următoarele probleme principale:

- protejarea datelor și resurselor proprii proceselor față de orice acces din exterior;
- protejarea resurselor partajabile între anumite procese față de tentative de acces din procese neautorizate;
- posibilitatea de sincronizare a două sau mai multe procese, în vederea comunicării între ele;
- luarea în considerare, atunci cînd este cazul, a timpului real, pentru a putea descrie aplicații specifice acestui domeniu.

Executarea concurentă a mai multor procese presupune suprapunerea lor în timp. Aceasta nu înseamnă neapărat că instrucțiuni din diferite procese trebuie să se

execute concomitent; ele se pot executa și alternativ. Condiția esențială pentru ca două procese să fie concurente este aceea ca prima instrucțiune a unui proces să se execute înainte de încheierea ultimei instrucțiuni a celuilalt proces.

Din punctul de vedere al gradului de simultaneitate care se poate obține la execuția proceselor concurente, se disting mai multe situații:

a) *Paralelism logic pe un sistem monoprocesor.* Procesorul se distribuie alternativ la mai multe procese. La un moment dat se execută fizic o acțiune corespunzătoare unui singur proces. Avînd în vedere însă că, alternativ, se execută acțiuni din diferite procese, acestea se desfășoară concurent [BH72].

b) *Paralelism fizic pe un sistem multiprocesor.* Existența mai multor procesoare permite ca, la un moment dat, să se desfășoare efectiv instrucțiuni corespunzătoare mai multor procese. Se realizează astfel un paralelism fizic între procese. Cuplarea între ele a procesoarelor se realizează în practică într o mare varietate de configurații, de care depinde și modul de asociere a procesoarelor cu procesele programului. Un caz frecvent este acela în care procesoarele sînt legate la o memorie comună iar fiecărui proces îi este atribuit, în exclusivitate, cîte un procesor [JS80].

c) *Paralelism fizic într-o rețea.* În acest caz legătura dintre procesoare se realizează nu prin acces la o memorie comună ci, exclusiv, prin canale de comunicație. Sistemul se numește *distribuit* iar părțile lui componente se numesc *noduri*. Nodurile, la rîndul lor, pot fi monoprocesor sau multiprocesor cu memorie comună. Modul în care sînt interconectate nodurile unei rețele determină *topologia* acesteia. Din acest punct de vedere există, în prezent, o mare varietate de structuri [LA81].

Între cazurile a), b) și c) descrise anterior există o gamă largă de situații intermediare. Spre exemplu, paralelismul fizic și cel logic pot să apară împreună într-un sistem de calcul dacă, într-un sistem multiprocesor sau într-un nod al unei rețele se execută mai multe procese decît numărul procesoarelor.

Din succinta prezentare anterioară rezultă că programele concurente se pot executa pe o mare varietate de arhitecturi de sisteme de calcul. Una din diferențele majore de la un suport fizic la altul este viteza de calcul. Influența acesteia asupra vitezei de execuție a proceselor programului este importantă, atît în ceea ce privește viteza absolută cît și cea relativă, a unor procese față de altele. Se pune în mod firesc întrebarea dacă, în faza de scriere a programului, programatorul trebuie, sau nu, să țină cont de acest lucru. Pentru limbaje de nivel înalt nici nu se furnizează astfel de informații care, chiar dacă ar exista, ar fi legate de un sistem de calcul particular și de un anumit compilator. În plus, evoluția în timp a proceselor concurente este influențată frecvent și de evenimente externe a căror apariție este imprevizibilă.

Considerentele de mai sus au determinat stabilirea următoarei cerințe fundamentale pentru programarea concurentă [BH73]: la realizarea programelor concurente se va ignora viteza absolută și relativă de evoluție a proceselor. Se va presupune doar că ea este diferită de zero și nu va influența obținerea rezultatelor scontate. Acest principiu rămîne valabil și pentru programarea în timp real cu toate că, în anumite aplicații, se pot impune anumite limitări de timp și, implicit, luarea în considerare a unor aspecte legate de viteza de execuție (ex. sisteme în timp real pentru supravegherea și conducerea unor procese industriale sau de altă natură).

În concluzie, proiectarea și programarea unei aplicații concurente constă în descrierea fiecărui proces în parte, completată cu specificarea interacțiunii dintre procese. Este necesar ca programul rezultat să poată fi executat corect pe sisteme cu arhitectură (inclusiv număr de procesoare) și performanțe diferite.

2. Avantaje ale programării concurente în limbaje de nivel înalt

Așa cum s-a arătat, programarea concurentă își are originea în proiectarea și scrierea sistemelor de operare (programare de sistem). Ulterior ea a încorporat și realizarea altor componente ale software-ului de bază (de exemplu sisteme de

gestiune a bazelor de date) precum și o categorie importantă de aplicații, în special cele în timp real sau interactive. În ultimul deceniu, programarea concurentă este strîns legată și de domeniul programării sistemelor multiprocesor sau distribuite.

În mod firesc, programarea concurentă s-a efectuat, la început, exclusiv în limbaje de asamblare. Această situație s-a prelungit foarte mult datorită performanțelor înalte la faza de execuție cerute aplicațiilor concurente (în special timpul de execuție) cît și datorită altor cerințe nerezolvate la momentul respectiv în limbaje de nivel înalt (de exemplu, accesul direct la echipamentul fizic, accesul la sistemul de întreruperi etc.).

Îmbunătățirea continuă a performanțelor tehnicii de calcul (creșterea vitezei de lucru a procesoarelor, apariția și dezvoltarea sistemelor multiprocesor) cît și perfecționarea limbajelor de programare, inclusiv a implementărilor aferente (facilități specifice nivelului de lucru în timp real și respectiv pentru descrierea proceselor concurente și a interacțiunii dintre ele completate de optimizarea continuă a codului obiect generat de compilatoare), a determinat modificarea acestui punct de vedere. S-a ajuns la utilizarea în domeniul programării concurente și a unor limbaje de nivel înalt special definite în acest scop numite, în continuare, *limbaje de nivel înalt pentru programarea concurentă* sau, pe scurt, *limbaje concurente*. Aceste limbaje prezintă, în primul rînd, avantajele cunoscute ale limbajelor de nivel înalt în comparație cu cele de nivel scăzut (sînt mai ușor de asimilat și de aplicat, impun o anumită disciplină de programare, specifică limbajului respectiv, se simplifică depanarea și întreținerea programelor, etc.), avantaje care conduc la creșterea spectaculoasă a eficienței muncii de programare.

În continuare se vor trece în revistă principalele criterii de evaluare a limbajelor concurente, derivate din criteriile generale pentru aprecierea limbajelor de programare [DH86, YO81]:

a) *Securitatea*. Depinde în mare măsură de gradul în care se detecte ă automat și se semnalează erorile de programare, atît în faza de compilare cît și în cea de execuție a programelor. Este influențată direct de securitatea intrinsecă a limbajelor de programare exprimată în primul rînd prin impunerea scrierii unor programe clare și bine structurate.

Din mai multe cauze, programarea concurentă necesită deplasarea centrului de greutate al procesului de detectare a erorilor dinspre execuție spre compilare:

- Compilarea se poate face pe orice calculator care suportă limbajul respectiv, fiind, de obicei, mai ieftină decît execuția, care se realizează pe echipamentul destinat aplicației.

- Detectarea și eventual tratarea erorilor la faza de execuție implică inserarea unor secvențe suplimentare de verificare în codul obiect generat pentru program și/sau apelarea secvențelor de verificare și tratare a erorilor incluse în sistemul de operare. În acest fel cresc atît necesarul de memorie cît, mai ales, timpul de execuție, ceea ce poate fi inacceptabil, în special pentru aplicații în timp real.

Tehnicile de compilare actuale acționează cu consecvență pentru diversificarea tipurilor de erori ce pot fi depistate la faza de compilare. Un exemplu tipic în acest sens este verificarea depășirii valorii indicelui unui tablou. În mod obișnuit aceasta se realizează la execuție, prin inserarea unei secvențe de verificare în zona din codul obiect unde se referă elementul de tablou. Tehnicile mai noi înlocuiesc această verificare cu alta, echivalentă, efectuată în compilare, la asignarea valorii indicelui, care trebuie să fie de tip subdomeniu.

Există, evident, o limită a numărului și tipurilor de erori care pot fi detectate în mod automat. Greutăți deosebite se manifestă în cazul erorilor la descrierea logică a programului. Un cuvînt important în această privință îl au tehnicile moderne privind verificarea corectitudinii programelor dar și felul în care limbajul de programare se pretează la această activitate.

b) *Lizibilitatea*. Este calitatea unui program de a fi ușor de citit și de înțeles. Depinde de foarte mulți factori, legați în primul rînd de limbajul de programare dar

și de priceperea și interesul programatorului în acest sens, și anume:

- Stabilirea cuvintelor rezervate la definirea limbajului și a celorlalți identificatori la scrierea programului astfel încît să fie cît mai sugestivi pentru scopul în care sînt utilizați.

- Introducerea în limbaj a unor structuri de date, de acțiuni și de modularizare, astfel încît procesul de trecere de la problemă la algoritm și ulterior la program să fie cît mai natural.

- Asigurarea prin limbaj a unei libertăți suficiente la scrierea și ordonarea textului programului pe hîrtie, care să poată fi folosită de programator pentru mărirea clarității (textul să reflecte structura programului). De remarcat că editoarele de texte moderne sînt înzestrate cu posibilitatea ordonării automate a programelor în scopul creșterii lizibilității.

O lizibilitate bună trebuie să reducă sau chiar să excludă documentația aferentă unui program. Programul devine astfel propria sa documentație. Fiind ușor de înțeles, se pot identifica rapid eventualele erori, ușurîndu-se în același timp și activitatea de întreținere-modificare a programelor.

Asigurarea unei bune lizibilități conduce în mod inevitabil la lungirea textului programului. Costul și efortul suplimentar implicat de aceasta sînt însă cu prisosință recuperate datorită marilor avantaje, mai ales dacă se are în vedere întregul ciclu de viață al programului, care presupune deseori preluarea și chiar modificarea lui repetată de către diferite colective de lucru.

c) *Flexibilitatea.* Este calitatea limbajului de programare de a permite exprimarea simplă și completă a operațiilor necesare într-un program, fără a se recurge la apeluri de subprograme din alte limbaje, inserări de secvențe în cod mașină sau alte artificii.

Pentru un limbaj de programare concurentă, o bună flexibilitate implică includerea unor facilități de manevrare și control a întregii game de echipamente periferice, posibilitatea măsurării timpului real și a tratării întreruperilor de timp, etc.

In general, flexibilitatea și securitatea sînt cerințe antagoniste, creatorul unui limbaj de programare fiind pus în situația de a realiza un compromis între ... pentru a nu dăuna securității, este bine ca flexibilitatea să se limiteze la strictul necesar pentru descrierea aplicațiilor din domeniul în care se recomandă limbajul de programare.

d) *Simplitatea.* Este dictată de simplitatea regulilor limbajului, de ușurința cu care ele pot fi înțelese și aplicate. Aceasta nu înseamnă neapărat evitarea unor construcții de limbaj complexe ci mai degrabă absența restricțiilor la utilizarea acestor construcții. Simplitatea limbajului are numeroase avantaje: simplifică munca de instruire a programatorilor, reduce posibilitatea de a face erori de programare rezultate din interpretarea greșită a regulilor limbajului, simplifică procesul de compilare, conduce la generarea unui cod obiect mai eficient și ușurează asigurarea portabilității.

Antagonismul dintre simplu și complex la nivelul limbajelor de programare concurentă poate fi exemplificat printr-o paralelă între Edison și Ada. Un alt exemplu poate fi conceptul de tip abstract de date în limbajele Edison și Modula pe de o parte și în Concurrent Pascal, Ada sau CLU pe de altă parte (cap. 1).

e) *Portabilitatea.* Este calitatea unui program, de a fi trecut ușor, eventual fără nici o modificare, de pe un calculator pe altul. In cazul în care programul în cauză este compilatorul unui limbaj de programare se vorbește de portabilitatea limbajului respectiv.

O modalitate frecventă pentru asigurarea portabilității unui limbaj este implementarea sa pe baza conceptului de mașină virtuală. In acest caz compilatorul nu generează la ieșire cod mașină ci un cod virtual ce urmează să fie interpretat și executat de un program special (executiv) care este eventual realizat în limbajul de asamblare al calculatorului [GO74]. Compilatorul se scrie la rîndul său într-un limbaj de nivel înalt răspîndit (frecvent chiar în limbajul pe care îl implementează)

[HR77]. Astfel, mutarea limbajului de programare de pe un sistem pe altul implică doar rescrierea executivului, cu un efort mult mai mic decît acela de a rescrie compilatorul. Portabilitatea unui limbaj afectează, de obicei, în sens negativ necesarul de memorie și timpul de execuție al programelor. Este evident că independența față de echipamentele hardware, nu poate asigura eficiență maximă la exploatarea unui anumit echipament. Din acest motiv, portabilitatea limbajelor de programare concurentă trebuie analizată de la caz la caz, cu circumspecție [CE84d, CE86b].

Portabilitatea programelor scrise într-un anumit limbaj se realizează în primul rînd prin standardizarea limbajelor de programare. Ea este de dorit, determinînd creșterea eficienței activității de scriere a programelor.

Cele cinci criterii de mai sus sînt prezentate aproximativ în ordinea importanței lor pentru porogramarea concurentă. Securitatea și lizibilitatea sînt deci considerate ca prioritare și absolut necesare în acest gen de aplicații. Se mai remarcă și faptul că în toate criteriile enumerate se mai reflectă încă unul care poate fi considerat ca fiind corolarul întregii activități de programare, și anume *eficiența*. Deși se poate vorbi și de o eficiență în sine măsurată, de exemplu, prin performanța programelor eficiența propriu-zisă a programării este încorporată în celelalte criterii.

La nivelul definirii limbajelor, criteriile de mai sus se materializează în cerințe privind, în special, introducerea unor restricții sau definirea unor facilități de limbaj care să permită realizarea unor programe concurente optime. Aceste cerințe sînt reflectate în detaliu în lucrarea de față (cap. 3, cap. 4).

3. Scurt istoric al limbajelor de programare concurentă și al abordării acestui domeniu în țara noastră

Prima realizare importantă în domeniul limbajelor de nivel înalt pentru programarea concurentă este Concurrent Pascal, definit de P.B. Hansen în 1975 [BH75a, BH75c]. Principala facilitate dezvoltată în acest limbaj este extinderea noțiunii de tip din Pascal prin definirea tipurilor sistem: **process**, **monitor**, **class** [BH76a]. Programele concurente se scriu ca o interconexiune de componente (variabile) de aceste tipuri. Trebuie remarcat și faptul că în acest limbaj concurent s-au introdus și facilități specifice programării în timp real, care sînt analizate în detaliu în cap. 4.

In anul imediat următor (1976) au apărut alte două limbaje recomandate în special pentru programarea în timp real: RTL/2 [BA80] și Portal [LI78]. Inspirat din Algol 68, RTL/2 este un limbaj redus, structurat și compact, rămînînd însă rudimentar în ceea ce privește abstractizarea datelor și mai ales în programarea concurentă. Referitor la Portal, ca limbaj concurent el aduce unele noutăți față de Concurrent Pascal. Tot ca limbaj recomandat cu prioritate pentru programarea în timp real poate fi considerat și Modula, definit în 1977 de N. Wirth [WI77a].

O altă direcție care a apărut în această perioadă este definirea de limbaje concurente destinate nu atît scrierii de aplicații ci mai ales pentru ilustrarea unor concepte teoretice noi prezentate în contextul unui limbaj de nivel înalt. Exemple marcante ale acestei categorii sînt: CSP [HO78] și DP [BH78].

După 1980, tendința de a defini limbaje concurente ca limbaje specializate, cu un număr redus de facilități și orientate spre un anumit domeniu, s-a redus considerabil. Limbajele concurente apărute în această perioadă sînt, mai degrabă, limbaje de programare universală, în care s-au introdus și facilități pentru programarea concurentă. Cele mai semnificative exemple în acest sens sînt Edison [BH81b], Modula-2 [WI82] și mai ale Ada [AN83].

Odată cu răspîndirea sistemelor multiprocesor precum și a celor distribuite a apărut în mod firesc necesitatea ca limbajele concurente să poată să pună în valoare avantajele potențiale oferite de astfel de sisteme. Aceasta a condus la diversificarea atît a facilităților de descriere a proceselor și a relațiilor proces - procesor cît mai ales a modalităților de specificare a comunicării dintre procese. Un salt teoretic

important în această direcție îl reprezintă limbajele speciale destinate rețelelor de calculatoare (array machine) sau acelea în care paralelismul operațiilor este controlat prin fluxul datelor (dataflow languages). Aceste limbaje se referă la sisteme cu arhitectură specială și se bazează pe modele teoretice deosebite în care sincronizarea propriu-zisă a proceselor iese din sarcina programatorului.

Studiul și implementarea limbajelor de nivel înalt pentru programarea concurentă au debutat în țara noastră în 1976, la Institutul Politehnic Timișoara. Inceputul acestei activități este legat de numele prof. A. Soceneanțu, în prezent la Universitatea din Provo (Utah-SUA). In acest context am avut șansa ca, din 1977, să conduc colectivul de cadre didactice și cercetători care au realizat primul compilator pentru un limbaj concurent (Pascal Concurent/Felix C) precum și programele de execuție corespunzătoare implementate pe un calculator românesc. Pînă în 1987 acest colectiv a realizat mai multe versiuni de implementare a limbajului Pascal Concurent, precum și a limbajului originar, Pascal, inclusiv o variantă interactivă pe calculatoarele din gama Felix C. In paralel, tot la Institutul Politehnic Timișoara, alte colective au realizat adaptări ale implementării lui P.B. Hansen pentru PDP 11/45, și pe calculatoarele din familiile Independent, Coral și Felix M. Este de semnalat, de asemenea, scrierea în această perioadă a primelor aplicații concurente în Pascal Concurent/Felix C atît la I.P.T., în cadrul unor contracte de cercetare-proiectare cît și în multe alte întreprinderi din țară.

Ca o continuare a preocupărilor privind implementarea limbajelor Pascal și Pascal Concurent, în activitatea de cercetare științifică sau de diplomă cu studenții secției de specialitate, am realizat, în diferite variante și pe diverse mini sau microcalculatoare, implementări ale limbajului de programare Edison. In legătură cu aceste implementări s-a avut în vedere, la fel ca în proiectul Edison inițial, conceput de P.B. Hansen, realizarea unui mediu de programare complet (compilator, editor, sistem de operare propriu, executiv, etc.).

In afara acestor realizări de la I.P.T., colective de cercetare din cadrul I.T.C.I. (București, Cluj) precum și I.C.E. - București au obținut rezultate notabile prin implementarea limbajelor RTL/2, Edison, Modula 2 și Ada.

4. Succintă trecere în revistă a conținutului lucrării

Lucrarea, în ansamblul ei, tratează o problematică relativ diversificată din domeniul limbajelor de programare. Se abordează atît aspecte teoretice legate de definirea formală a limbajelor sau rezultate din studiul comparativ al unor concepte noi, introduse în limbajele de programare al ultimului deceniu cît și aspectele practice rezolvate pe parcursul implementărilor mai multor limbaje, în special a celor orientate pentru programarea concurentă și în timp real.

Astfel, în primul capitol se identifică unele tendințe noi privind abstractizarea în limbajele de programare. In acest context, un loc important îi este acordat conceptului de programare orientată spre obiecte cît și unui studiu comparativ privind evoluția conceptului de dată abstracă și modul în care acesta a fost introdus și implementat în diferite limbaje de programare.

In capitolul doi se reliefează rolul descrierii formale complete a unui limbaj de programare atît pentru definirea și înțelegerea corectă a limbajului sub toate aspectele sale cît și pentru implementarea limbajului respectiv, prin proiectarea și realizarea compilatorului. Se trec în revistă principalele direcții și metode de modelare formală cristalizate în literatura de specialitate și, pe baza unor criterii definite anterior, se realizează o comparare a acestor metode, rezultînd unele concluzii.

Capitolul trei tratează problematica tipică limbajelor de nivel înalt pentru programarea concurentă. Modalitățile concrete de descriere a proceselor concurente precum și cele de sincronizare și comunicare între procese, abordate cu prioritate și tratate în detaliu sînt cele specifice limbajelor Concurrent Pascal și Edison, pentru care, în partea a doua a lucrării se prezintă variante de implementare.

In capitolul patru se analizează specificitatea aplicării limbajelor concurente de nivel înalt la domeniul programării în timp real. Se definește relația "programare-concurentă" - "programare în timp real" și se prezintă felul în care unele trăsături proprii programării în timp real se reflectă în facilitățile speciale introduse în limbalele de nivel înalt destinate acestui domeniu.

Subiectul capitolului cinci poate fi considerat ca făcînd parte din domeniul tehnicilor de compilare. La început, se prezintă comparativ mecanismele pentru controlul vizibilității identificatorilor din cîteva limbaje de programare. Apoi, pe baza unor instrumente de modelare formală, se evaluează aceste mecanisme. In final, se descriu algoritmii pentru analiza de domeniu, implementați de autor în diferite compilatoare.

Capitolele șase și șapte se referă la compilatorul pentru limbajul de programare Pascal Concurent/Felix C realizat la Institutul Politehnic Timișoara. Capitolul șase prezintă organizarea datelor compilatorului tratînd în continuare doar problemele legate de specificul concurent al limbajului: compilarea tipurilor și variabilelor active, a procedurilor, funcțiilor și a variabilelor externe precum și a declarațiilor de programe secvențiale. In capitolul șapte se descriu structura și funcționarea pe faze a compilatorului, precum și un sistem original de compilare interactivă a programelor Pascal și Pascal Concurent realizat pe baza acestui compilator.

Capitolul opt prezintă structura, organizarea și funcțiile nucleului executivului Pascal Concurent. Un loc important ocupă în acest capitol implementarea conceptului de monitor cît și mecanismele originale definite și implementate pentru comunicarea între module și programe.

In capitolul nouă se prezintă părți ale unui mediu de programare pentru limbajul Edison, realizate în mai multe variante, pentru diferite microsisteme de calcul. In acest sens se tratează proiectarea și implementarea unei mașini virtuale, inclusiv problemele rezultate din definirea unui cod virtual optim, structura și funcționarea nucleului și proiectarea unui sistem de operare propriu mediului de programare. In final se prezintă o variantă de extindere a executivului Edison pe sisteme multiprocesor.

In încheierea lucrării se trag cîteva concluzii în legătură cu problematica tratată și cu cercetările efectuate și se prezintă contribuțiile originale ale autorului.

Capitolul 1

CITEVA TENDINTE NOI PRIVIND ABSTRACTIZAREA IN LIMBAJELE DE PROGRAMARE ACTUALE

Deşi cercetările şi chiar realizările practice din ultimul timp, în domeniul calculatoarelor electronice şi al programării [JI81], încearcă să zdruncine punctul de vedere devenit clasic, sîntem obişnuiţi să privim calculatorul ca o maşină capabilă să execute operaţii simple (aritmetice, logice sau de altă natură) asupra unor entităţi înmagazinate în memoria sa în reprezentare exclusiv numerică. Programul obiect care permite unui calculator să rezolve o anumită problemă trebuie introdus în memorie sub forma unei succesiuni de comenzi acceptate de maşină, determinată de algoritmul de rezolvare a problemei.

Aplicarea *algoritmului* presupune efectuarea de operaţii (acţiuni) asupra unor entităţi reprezentate prin date. *Programul* poate fi privit ca o expresie a algoritmului de calcul într-un limbaj accesibil unui calculator [NG82].

Acest punct de vedere a stat la baza definirii marii majorităţi a limbajelor de programare, gîndite pentru a permite reprezentarea de algoritmi la un anumit nivel de abstractizare. Ele formează familia *limbajelor algoritmice sau procedurale* care domină astăzi categoric în toate domeniile în care se aplică tehnica de calcul. Aceste limbaje de programare oferă, prin urmare, modalităţi de descriere de *acţiuni* şi *date* iar evoluţia lor poate fi identificată cu diversificarea şi perfecţionarea continuă a acestor modalităţi.

1.1. Etape şi nivele de abstractizare

Nivelul de abstractizare separă net limbajele de asamblare, simple extensii ale limbajelor maşină, de cele de nivel înalt.

In *limbajele de asamblare* datele sînt reprezentate, în majoritatea cazurilor, doar prin rezervări de locaţii de memorie iar acţiunile sînt fie identice cu operaţiile fizice pe care le poate efectua calculatorul (codul maşină), fie mici grupări de astfel de operaţii elementare (macroinstrucţiuni). Pentru anumite operaţii speciale (intrări - ieşiri, gestiunea timpului real, tratarea întreruperilor şi derutelor, etc.) se pot apela funcţii puse la dispoziţie în acest scop de sistemul de operare. Limbajul de asamblare împreună cu sistemul de operare reprezintă o primă abstractizare a calculatorului real, utilizat în programare.

Limbajele de programare de nivel înalt s-au dezvoltat pe linia punerii la dispoziţia utilizatorilor calculatoarelor a unor unelte de programare care să permită formularea programului într-o manieră independentă de structura şi funcţionarea calculatorului. Zestrea de acţiuni cu care sînt prevăzute limbajele de nivel înalt a evoluat de la acţiuni simple (atribuirea, saltul condiţionat sau necondiţonat) la grupări de acţiuni din ce în ce mai complexe [CE85a]. Un moment important în această direcţie îl reprezintă definirea în 1960 a limbajului ALGOL [NA63]. Structurile de acţiuni introduse în acest limbaj reprezintă chiar structurile de control fundamentale care stau la baza *metodei programării structurate*.

Abstractizarea datelor şi acţiunilor în limbajele de programare poate fi apreciată calitativ în mai multe etape. Pe de o parte, se poate analiza raportul între descrierile şi prelucrările de date pe care le oferă limbajul şi modul în care acestea sînt reprezentabile în calculatorul real. Pe de altă parte, nivelul de raportare ar putea fi mutat în realitatea înconjurătoare pentru care urmează să se scrie programe în limbajul respectiv. In acest caz, abstractizarea va fi evaluată prin distanţa dintre obiectele reale şi reprezentarea lor în program. Cele două etape de abstractizare denumite *fizică* şi respectiv *logică* sînt ilustrate în fig. 1.1.

Evoluţia calculatoarelor şi a limbajelor de programare, scoate în evidenţă tendinţa

de a reduce din ce în ce mai mult efortul de abstractizare pe care trebuie să-l depună programatorul, cu implicație directă asupra eficienței muncii de programare. Pentru aceasta s-a acționat și se acționează în continuare asupra ambelor etape, astfel:

- s-au realizat microprocesoare și chiar calculatoare ale căror limbaje mașină sînt identice sau foarte apropiate de anumite limbaje de nivel înalt [IN81, MA82, ZE81];

- s-au diversificat foarte mult facilitățile de descriere a datelor și a acțiunilor de prelucrare aferente, incluse în limbajele de programare [WI76, CR87, CI78a].

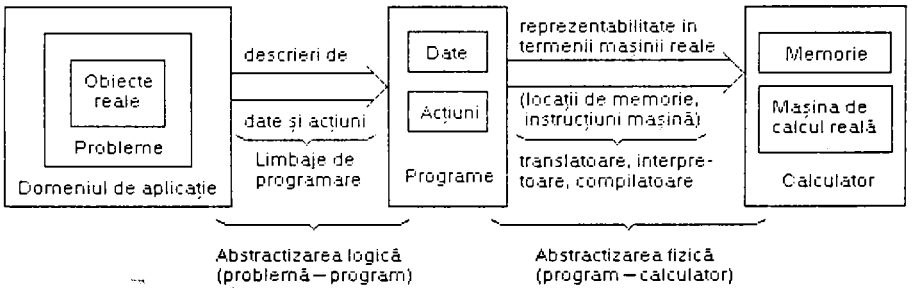


Fig. 1.1. Etape de abstractizare în limbajele de programare

In ceea ce privește abstractizarea datelor, un salt deosebit l-a reprezentat definirea, în 1971, a limbajului PASCAL [WI71, JW74]. Inițial (limbajul FORTRAN) s-a considerat suficientă reprezentabilitatea și operarea cu entități de bază ale matematicii (valorile întregi, reale, complexe, logice), luate ca elemente simple sau grupate în structuri omogene (tablouri); date de altă natură sau cu altă structură trebuiau codificate în termenii limitați ai acestor reprezentări. Ulterior (limbajele COBOL, PL/1, ALGOL68) au apărut și primele structuri neomogene (înregistrări). In limbajul PASCAL s-au diversificat atît datele elementare cît și structurile de date posibil de descris în program. Saltul calitativ esențial l-a reprezentat mecanismul de definire de tipuri pus la dispoziția programatorului. Astfel, tehnica structurării s-a extins și asupra datelor, devenind o metodă de programare complexă și completă [VB78].

Primul pas spre programarea modulară îl reprezintă conceptul de *subprogram*. Limitele utilizării acestui concept ca instrument de abstractizare se manifestă în principal din următoarele două cauze:

1) Existența unor relații predefinite între programul principal și subprogram, materializate prin reguli de domeniu și vizibilitate, mecanisme de definire și apel, etc.

2) Dificultățile de asociere biunivocă a subprogramelor cu obiectele reale, componente ale problemei de programat. In aceste condiții, structura programului reprezentat prin subprogramele sale nu reflectă natural realitatea modelată (privită ca o mulțime de obiecte).

Evoluția limbajelor de programare precum și dezvoltarea teoriei programării au condus la întrepătrunderea metodelor de abstractizare a datelor cu cele de modularizare a programelor, prin definirea conceptului de *dată abstractă* [GU77]. Acest concept a fost de obicei implementat sub forma unui tip ce poate încapsula într-o construcție de limbaj unitară atît structuri de date cît și acțiunile de prelucrare aferente lor. Este motivul pentru care el se mai numește și *tip abstract*, reprezentînd abstractizarea maximă gîndită pînă în prezent pe linia generalizării noțiunii de tip.

Apariția și utilizarea conceptului de dată abstractă a permis deplasarea și mai accentuată a limbajelor de programare și a programării în general, dinspre mașina de calcul spre domeniul concret de aplicație (fig. 1.1.), întrucît obiectele reale își găsesc

acum un corespondent direct în limbaj. Aspectele descrise mai sus sînt ilustrate sintetic în fig. 1.2. în care se face de asemenea și o propunere de stratificare pe nivele a evoluției abstractizării în programare [CE88a].

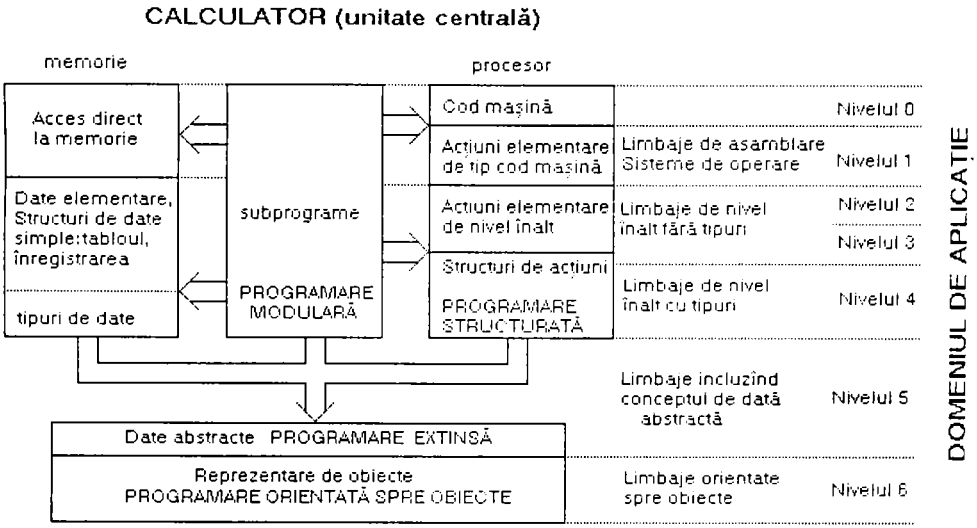


Fig. 1.2. Nivele de abstractizare în limbajele de programare

Structurarea și modularizarea au permis elaborarea unor principii științifice pentru activitatea de programare. Această cucerire pe plan teoretic a fost urmată în mod firesc de transpunerea sa în practică, sub forma unor metodologii concrete de proiectare și programare [LM79, PA72a]. Au căpătat o importanță din ce în ce mai mare, datorită implicațiilor asupra eficienței programării, aspectele ce țin de stilul de programare. S-a demonstrat că acesta este în mod decisiv influențat de limbaj care poate ușura sau, dimpotrivă, poate îngreuna ceea ce Wirth denumea încă din 1973 *programarea sistematică* [SC84, WI73].

Diferențierile ulterioare în producerea de programe mari și respectiv mici au condus la definirea termenilor de *programare extinsă* și *programare restrînsă* [RE82, VB86], ca faze distincte și succesive, prima fiind în primul rînd legată de proiectare. Programarea extinsă reprezintă activitatea de structurare a unui program mare în module. Aceasta presupune inclusiv stabilirea funcțiilor pe care trebuie să le îndeplinească modulele și a posibilităților lor de interacționare. Prin contrast, programarea restrînsă, se referă la elaborarea concretă a modulelor din care se compune sistemul, implicînd detalierea funcțiilor lor interne. Creșterea importanței programării extinse a condus la definirea de unelte și limbaje de programare noi, orientate spre această activitate. Dezavantajul lor principal constă în aceea că nu permit și programarea restrînsă, ceea ce se explică prin dificultățile majore de apropiere a operațiilor componente ale celor două faze. Realizările recente în domeniul limbajelor de programare, exploatînd conceptul de dată abstractă, au reușit să rezolve în mod satisfăcător fuzionarea exprimării ambelor faze de programare prin facilitățile aceluiași limbaj (exemple: SIMULA, EUCLID, MODULA-2, ADA, SMALLTALK, CLU, etc.).

Această succintă trecere în revistă a evoluției programării prin prisma diversificării metodelor și mijloacelor de abstractizare a scos în evidență cîteva dintre progresele remarcabile înregistrate în noile limbaje de programare. Ele n-au afectat însă aproape deloc caracterul procedural al limbajelor, comportarea lor în relația cu utilizatorii ca o mașină von Neumann abstractă [WU81]. Această comportare se

regăsește și în cazul limbajelor de programare concurentă, tratate mai amplu în lucrarea de față.

In paralel cu evoluția descrisă mai sus s-au dezvoltat (începînd cu anii '60) și *limbaje cu caracter neprocedural*. Primul reprezentant de seamă al acestei familii, LISP [AL79, WN79], se află astăzi la baza limbajelor utilizate în domeniul inteligenței artificiale [GE85]. Programele scrise în limbaje neprocedurale nu conțin specificarea secvenței acțiunilor (algoritmul) de executat. Ele încearcă să definească problema de executat, plasată în contextul general al prelucrărilor necesare. Dintre limbajele neprocedurale mai cunoscute se amintesc PROLOG [CM84] și SNOBOL [FG64]. Este dificil să se precizeze în mod categoric dacă un limbaj este neprocedural: se poate aprecia mai degrabă dacă limbajul are sau nu trăsături în acest sens. Pentru astfel de aprecieri există și încercări de definire a principalelor caracteristici ale limbajelor neprocedurale [LS74]. Tendințele de dezvoltare a unor structuri de calculatoare principial noi [JI81], permit pronosticarea unei răspîndiri tot mai accentuate și a limbajelor din această categorie.

1.2. Valori și obiecte în limbajele de programare

Teoria și practica actuală a programării au demonstrat că abordarea programelor de complexitate mare este mult ușurată dacă se face o diferențiere clară între conceptele de valoare și obiect. Acesta este și motivul pentru care, în literatura de specialitate, se utilizează tot mai frecvent, chiar fără a-i clarifica în mod satisfăcător, termenii de "orientat spre valori" și "orientat spre obiecte". In continuare se face o scurtă analiză care încearcă să elucideze conceptele de valoare și obiect atît prin enunțarea caracteristicilor mai importante cît și prin concretizarea lor în diferite limbaje de programare. In final, va rezulta că diferențierea între valori și obiecte poate fi deosebit de utilă în efortul de abstractizare presupus de realizarea unui program.

Cele mai sugestive exemple de *valori* sînt numerele din matematică. In [CE89a] se prezintă în detaliu principalele proprietăți ale valorilor: universalitatea, nealterabilitatea, unicitatea, permanența. Valorile s-au introdus în limbajele de programare prin *constante* (date direct prin valoarea lor sau desemnate de identificatori predefiniți în limbaj). Un pas important înainte pe linia "orientării spre valori" a fost marcat prin introducerea în limbajele de programare a unor mecanisme de definire de constante [CE81a, KJ82]. Acest gen de constante nu mai reprezintă însă valori pure, întrucît își pierd proprietatea de permanență. Cu această excepție, se poate aprecia că proprietățile stabilite pentru valori se regăsesc și în cazul constantelor.

La un moment dat s-a pus în mod firesc întrebarea: este posibil să se definească limbaje care să aibe doar valori? Acestea ar fi limbaje orientate spre valori, pure. Definirea limbajelor pentru programarea funcțională [JM78, ST86] a reprezentat un răspuns afirmativ.

Programarea calculatoarelor reprezintă în primul rînd simularea sau modelarea unor situații reale. Acest aspect se reflectă atît în acțiunile cît mai ales în datele și structurile de date descrise în programe. Același lucru se poate afirma și despre programarea concurentă: datele și structurile de date din sistemele de operare reflectă stările și situațiile reale în care se găsesc resursele fizice și logice ale sistemului de calcul. Dar lumea reală este alcătuită din obiecte. Aceste *obiecte* se reprezintă în programe prin *variabile*. Pornind de la însăși esența noțiunilor de obiect și variabilă [BO86, HO84], există importante analogii între ele.

Astfel, două obiecte pot fi identice ca formă și conținut și totuși se consideră diferite întrucît ocupă spații diferite. In mod analog, două variabile care ocupă regiuni de memorie diferite, pot să aibe aceeași structură și conținut.

Spre deosebire de valori, unic determinate prin relații și proprietăți interne, obiectele și reprezentările lor în calculator, variabilele se definesc doar prin relații

exterioare. Intre acestea un rol important îl ocupă numele simbolic care permite invocarea lor.

Obiectele şi variabilele se caracterizează printr-o anumită stare momentană, dată de totalitatea proprietăţilor şi atributelor interne. Această stare se poate schimba în timp fără a afecta identitatea obiectului sau variabilei. Calitatea unui obiect de a avea o stare momentană conduce la ideea existenţei lui în timp, de unde, logic, se poate trage concluzia că poate fi atît creat cît şi distrus. Acest aspect este şi mai evident în cazul variabilelor. De exemplu, pentru limbajele în care programele sînt organizate pe blocuri [CE81b, NI75, IC79], variabilele locale sînt create la intrarea în bloc şi se distrug la ieşire. Un exemplu elocvent este şi mecanismul de alocare-realocare a variabilelor dinamice care, în unele limbaje de programare [EC81a], este definit în mod explicit, ca unealtă la dispoziţia programatorului.

Acceptînd că proprietăţile obiectelor pot fi modificate în timp, devine esenţial dacă un obiect este, sau nu, partajat, din următoarele cauze:

- doi partajori diferiţi pot încerca să schimbe simultan proprietăţile unui obiect;
- o schimbare făcută asupra obiectului de un partajor poate fi vizibilă altui partajor.

Ambele cauze conduc la efecte secundare care se încadrează în două categorii:

1) nedorite - şi ca atare trebuie evitate prin eşalonarea cronologică a partajorilor;

2) utile - ca mijloace de comunicare între partajori.

Şi această proprietate se transmite variabilelor care pot şi, uneori, trebuie să fie partajate [AN78, BH73a]. In cazul valorilor, noţiunea de partajare este neesenţială datorită proprietăţilor de nealterabilitate şi unicitate. Alteori, în funcţie de limbajul de programare sau de implementarea sa, poate fi util să se evite efectele secundare prin realizarea mai multor copii ale aceleiaşi variabile, ceea ce de asemenea nu are sens sau este nesemnificativ în cazul valorilor.

Practica programării a dovedit că transpunerea pe calculator a u' or probleme din realitatea înconjurătoare este mult uşurată dacă se poate face o corespondenţă biunivocă între obiecte şi variabile. Realizînd acest deziderat, variabilele pot fi considerate ca obiecte ale mediului de programare respectiv. Acest punct de vedere a stat la baza limbajelor de programare orientate spre obiecte al căror prim reprezentant veritabil este SMALLTALK [GR83, RG81]. Un astfel de limbaj trebuie să conţină atît modalităţi de declarare a obiectelor cu structură şi funcţii corespunzătoare celor din realitate, cît şi posibilităţi de manipulare a lor, făcînd abstracţie de proprietăţile şi atributele neesenţiale. Prelucrările urmărite se rezolvă prin mesaje la care sînt receptive variabilele - obiecte corespunzătoare.

Unitatea structurală de bază în sistemul SMALLTALK este un tip abstract inspirat din noţiunea de clasă a limbajului SIMULA [DM68]. Este deosebit de dificil să se dea o definiţie formală programării orientate pe obiecte [RO81, RE82]. In continuare se va încerca o caracterizare a sa prin cîteva din atributele esenţiale, exemplificate pe limbajul SMALLTALK [SH79, TE81]:

1) Obiectele dintr-un program sînt privite întotdeauna din exterior, interesînd relaţiile lor cu utilizatorul şi cu celelalte obiecte din sistem. Structura şi proprietăţile interne sînt neesenţiale din acest punct de vedere, motiv pentru care se interzice "cercetarea" interiorului unui obiect. El singur are calitatea de a decide ce anume din interiorul său poate fi transmis în exterior (în acele puţine cazuri în care aceast lucru este posibil).

2) Obiectele se tratează în mod uniform indiferent că sînt obiecte "primitive" (ex. întregii), obiecte "sistem" (ex. clasele) sau obiecte definite de utilizator. Această tratare uniformă implică atît mecanismul de referire a obiectelor (prin acelaşi fel de nume simbolice), cît şi pe cel de comunicare între ele.

3) Activităţile de prelucrare sînt interioare obiectelor.

4) Utilizatorul comunică cu obiectele sistemului care, la rîndul lor pot să comunice între ele prin transmitere de mesaj [BR83, EC86a, HM84]. Utilizatorul cere realizarea unor prelucrări transmiţînd un mesaj. Obiectul care îl recepţionează poate

apela la rîndul lui la alte obiecte fie pentru informații, fie pentru unele din prelucrările solicitate, transmițîndu-le de asemenea mesaje. In principiu, în programarea orientată pe obiecte orice obiect poate realiza prelucrarea solicitată de orice mesaj, dirijînd scurgerea mesajului către alte obiecte.

5) Trimiterea de mesaje este uniformă indiferent de prelucrarea solicitată. Este posibil ca două mesaje să fie identice și totuși să aibă semnificații diferite, diferența rezultînd din context.

Așa după cum arată autorii sistemului SMALLTALK [IN81], programarea orientată pe obiecte se bazează pe noțiunile de activitate, comunicare și partajare, iar noțiunile clasice de dată și procedură se estompează, pînă la dispariție.

Limbajele orientate spre obiecte pot să conțină și valori cu precizarea că aceste valori nu se consideră ca simple abstractizări, așa cum sînt ele de fapt, ci ca stări ale unor obiecte care au același statut cu toate celelalte obiecte din sistem. Extrapolînd acest procedeu s-ar putea considera că, în general, în calculatoare nu există valori. Valorile sînt abstractizări și deci nereprezentabile direct. Tratarea lor în calculator trebuie să fie precedată de reprezentarea sau codificarea prin obiecte. Pe de altă parte obiectele tratate într-un calculator sînt caracterizate de stare, reprezentabilă prin valori. In concluzie, valorile și obiectele se întrepătrund în intimitatea calculatorului.

In ceea ce privește limbajele de programare, marea lor majoritate permit manipularea atît a valorilor cît și a obiectelor, fără a face o diferențiere clară între ele [ML80]. Eventuala delimitare rămîne la granița dintre constante și variabile fără a lua în considerare toate cerințele de bază ale "orientării spre obiecte" [YC79, WH84]. Un aspect important care a evoluat foarte mult este acela al creării și desființării variabilelor în timpul execuției programului. In FORTRAN/Felix C, variabilele se crează (prin rezervarea locațiilor de memorie corespunzătoare) la încărcarea în memorie a segmentului (în particular a programului) din care fac parte și se desființează la acoperirea segmentului [NS79]. In limbajele în care programele sînt organizate pe blocuri, existența unei variabile este legată de existența blocului care o include [WJ75, TA79, MM86]. In fine, există limbaje care pun la dispoziția programatorilor modalități de creare și desființare dinamică în timpul execuției, a variabilelor (facilități de alocare - relocare dinamică a memoriei).

In prezentarea multor limbaje de programare, calitatea de a fi valoare sau obiect a unei entități este legată în mod inutil de tipul său. In această idee, tipurile simple (nestructurate) de date se consideră valori, iar tipurile compuse (structurate) se tratează ca obiecte. Această concluzie complică programarea, neputîndu-se realiza analogii convenabile între obiectele lumii reale și clasele de obiecte ale limbajului. Realitatea arată că există și, în consecință, trebuie reprezentate în limbajele de programare, atît obiecte simple cît și valori compuse. Această tendință este evidentă în unele limbaje mai noi (de exemplu constructorii din majoritatea limbajelor apărute după PASCAL). Concluzia finală este următoarea:

Considerînd programarea ca modelare a realității într-un limbaj de programare universal poate fi realizat mai ușor sau mai greu orice program. Limbajele pure, orientate pe valori sau pe obiecte pot să simplifice în anumite cazuri programarea. Dar în mod cert activitatea programatorului devine mai ușoară iar efortul de modelare sau simulare cu calculatorul mai natural, dacă se face distincție între valori și obiecte într-un limbaj în care coexistă ambele concepte.

1.3. Evoluția conceptului de dată abstractă

In literatura de specialitate actuală este destul de răspîndită următoarea clasificare a tipurilor de abstractizări [AH79, ME81, NG82, VB86]:

- abstractizarea controlului, realizată prin utilizarea structurilor de acțiuni proprii programării structurate și/sau a altora echivalente cu acestea;
- abstractizarea datelor, rezolvată parțial prin definirea tipurilor de date (simple

sau structurate);

- abstractizarea procedurală, bazată pe declararea și utilizarea subprogramelor.

Uneltele de programare amintite mai sus la fiecare tip de abstractizare rezolvă numai de la un punct cerințele exprimate în practică, fiind deficitare în special în ceea ce privește programarea extinsă. Aceasta ar avea nevoie de o abstractizare hibridă (procedurală și de date) care să privească modelul ce stă la baza programului ca fiind format aproape în întregime din obiecte abstracte. În cazul general, un astfel de obiect nu poate fi reprezentat printr-o simplă structură de date, ci printr-o colecție de structuri asupra cărora acționează un set bine precizat de operații. Acestea nu sînt, de obicei, operații primitive ale limbajului. Un alt aspect important este acela că sistemele sînt deseori compuse din obiecte diferite ca identitate, dar cu aceeași structură internă din punct de vedere al datelor și acțiunilor reprezentate. Posibilitatea de a defini obiecte abstracte ca tipuri ale unui limbaj de programare reprezintă o soluție satisfăcătoare la problemele semnalate, avînd în plus, și o serie de alte avantaje:

a) Se face o distincție netă între utilizarea tipului abstract și implementarea sa în memoria calculatorului. Implementarea rămîne în sarcina compilatorului care poate decide detaliile de reprezentare și acces, fără să cunoască utilizarea tipului. În acest fel, programatorul este degrevat de o serie de probleme pentru care, anterior, consuma o bună parte din timpul alocat realizării unei lucrări. El va gîndi acum programul ca fiind compus din obiecte abstracte ce pot, eventual, interacționa și nu îl interesează reprezentarea lor în calculator.

b) Proiectarea și programarea se realizează implicit ierarhizat și modular. În aceste condiții, metoda de proiectare top - down spre exemplu [WB78], este nu numai eficientă dar și firească, naturală. Procesul de proiectare - programare poate fi organizat ca în figura 1.3.

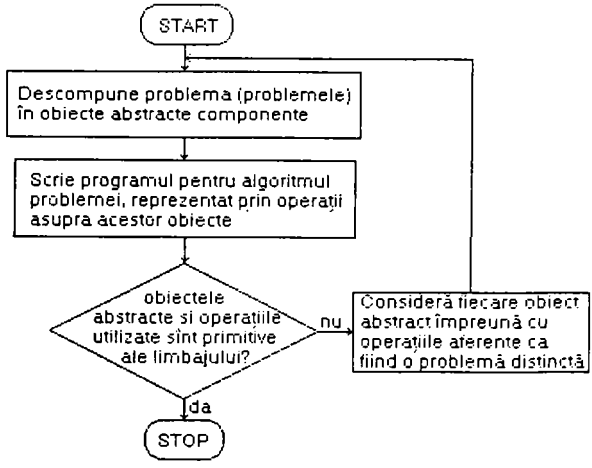


Fig. 1.3. Procesul de proiectare - programare utilizînd tipuri de date abstracte

c) Abstractizarea la acest nivel nu exclude ci, chiar presupune celelalte tipuri de abstractizări care vor acționa însă în interiorul tipului abstract. De exemplu, accesul la datele interne unui tip abstract se realizează de obicei, prin intermediul unor proceduri de acces definite în cadrul său.

d) Programul rezultat este mai bine organizat deoarece datele și acțiunile corespunzătoare aceluiași obiect sînt concentrate în același tip abstract. Acest fapt mărește claritatea programului și îl face ușor modificabil.

e) Utilizarea unui tip abstract nu necesită cunoașterea în amănunt a definiției sale. Din acest motiv, majoritatea limbajelor de programare care permit definirea de tipuri abstracte împiedică accesul direct, din exterior, la reprezentările tipurilor respective. Toate informațiile referitoare la structura internă sînt "confidențiale" și inaccesibile (invizibile) în afara tipului. În vederea comunicării între diferite obiecte abstracte se indică datele și în special procedurile (de acces) care sînt vizibile din exterior și pot fi referite în acest scop, conform anumitor reguli. Aceste reguli, împreună cu cele care definesc propriu-zis vizibilitatea, stabilesc un statut precis și verificabil încă din compilare, al comunicării între obiecte definite prin tipuri abstracte, ceea ce crează o siguranță deosebită în funcționarea programului în ansamblu.

Tipurile de date abstracte sau, pe scurt, *datele abstracte* reprezintă cel mai evoluat instrument de abstractizare a datelor utilizat în prezent în limbajele de programare. O dată abstractă este însă doar descrierea unui obiect. Modalitatea ca, pe baza acestei descrieri, să se creeze obiectul propriu-zis diferă de la un limbaj la altul [SC84].

Pentru introducerea unui concept de tip abstract într-un limbaj de programare, trebuie rezolvate două categorii de probleme diferite: unele se referă la modalitățile introduse în limbaj pentru utilizarea tipului abstract ca mijloc de descriere a obiectelor din lumea reală, iar celelalte privesc reprezentarea obiectelor descrise în memoria calculatorului și nu intră în sfera de preocupări a utilizatorului. De altfel abstractizările de date sînt folosite uneori, în anumite faze ale proiectării sau programării, chiar dacă limbajul de programare ales nu conține facilități în acest sens, tocmai în scopul amînării pentru etape ulterioare a unor decizii de implementare și reprezentare în memorie a datelor [PA72b].

Problemele care intră în prima categorie pot fi sintetizate astfel:

1. Trebuie introdusă o construcție d limbaj care să permită reprezentarea unei date abstracte ca o entitate, împreună cu toate operațiile admise a fi efectuate asupra ei. Stabilirea setului de operații concret pentru o anumită dată abstractă rămîne în sarcina programatorului care poate folosi în acest scop toate celelalte facilități oferite de limbajul de programare [PS75].

2. Accesul din exterior la datele descrise în tipul abstract trebuie realizat doar prin operațiile prevăzute în interiorul său, eventual doar printr-o parte din acestea, indicate în mod special. Pentru a face față acestei cerințe, programatorul este obligat, ca prin operațiile indicate, să acopere complet comportarea obiectelor descrise, trecerea lor în diferite stări.

3. Limbajul poate să ofere modalități explicite pentru crearea sau/și desființarea obiectelor descrise prin tipuri abstracte [AR84, HL82].

La implementarea unei date abstracte [CC81, CE87a, SS83], problemă care este lăsată pe seama compilatorului, se stabilesc reprezentările în memorie corespunzătoare diferitelor tipuri de date componente, se aleg algoritmii de acces la informații, etc.

În limbajele de programare de nivel înalt actuale se întîlnesc două maniere conceptual diferite de introducere a datelor abstracte. În prima, data abstractă se definește ca tip; obiectele corespunzătoare sînt variabile de acest tip; ele se crează și există ca entități distincte în faza de execuție a programului, motiv pentru care această manieră de a privi abstractizarea se poate numi *dinamică* (clasele din SIMULA și CONCURRENT PASCAL, modulul din EUCLID, grupul din CLU). În maniera *statică* limbajul oferă doar mijloace de modularizare a programului (pachetul din ADA, modulele din EDISON și MODULA) care pot fi utilizate în vederea abstractizării datelor. În acest caz, în interiorul modulului se descriu tipuri de date și operațiile corespunzătoare; obiectele sînt variabile de tipuri descrise în module și declarate efectiv în afara acestora. Această a doua manieră reprezintă un mijloc de modularizare a programului care permite în plus controlul explicit al vizibilității identificatorilor, garantînd astfel accesul la obiecte doar prin operațiile permise. Prin

urmare, modulul își epuizează rolul în faza de compilare el nemaifiind necesar ca entitate distinctă în faza de execuție.

In continuare se vor analiza comparativ cîteva concretizări ale conceptului de dată abstractă în diferite limbaje de programare.

1.3.1. Clasa

Introdusă inițial în limbajul SIMULA 67, noțiunea de clasă (**class**) este prima încercare de grupare într-o construcție de limbaj unitară a unor structuri de date, împreună cu toate operațiile care se pot efectua asupra lor [DD72]. Scopul definirii acestor entități a fost legat, la început de intenția de a crea un instrument util în simulare [VA77]. Ulterior, în noțiunea de clasă, s-a identificat conceptul general care permite abstractizarea completă a datelor. Programarea orientată spre obiecte (SMALLTALK) și programarea concurentă (CONCURRENT PASCAL și toate limbajele care i-au urmat) s-au bazat încă de la definirea lor pe acest concept, completat cu atribute și sensuri noi în conformitate cu noile întrebuințări. Există două variante mai semnificative ale implementării noțiunii de clasă în limbajele de programare: SIMULA 67 și CONCURRENT PASCAL.

a) *SIMULA 67* [DM68] este un limbaj de programare derivat din ALGOL 60, completat cu unele facilități utile pentru simulare. In acest context s-a observat că blocul ALGOL, care se activează prin apel și interacționează cu blocul apelant doar prin retransmiterea rezultatelor la încheierea activității sale, nu constituie un instrument corespunzător și eficient pentru modelarea obiectelor care apar în procesul de simulare. Obiectele simulate se pot crea unele pe altele (analog cu blocurile), dar apoi trebuie să existe permanent, fie independent, fie interacționînd între ele. Entitățile care satisfac aceste cerințe s-au numit în limbaj clase (**class**) și se definesc în mod similar unui tip, conform structurii din figura 1.4.

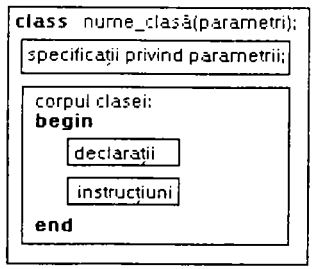


Fig. 1.4. Structura unei clase SIMULA 67

Corpul corespunde blocului din ALGOL 60. Aici se definesc și operații asupra obiectului abstract implementat, sub formă de proceduri. Crearea obiectului real (inclusiv alocarea lui în memorie) se realizează printr-o operație de alocare dinamică explicită prin care se asociază clasei o variabilă de referință (pointer). Pointerul asigură accesul la clasa respectivă fiind utilizat pentru calificarea procedurilor interne, în cazul apelului din exterior. Acest mod de apel permite efectuarea verificărilor de tip în timpul compilării. Definirea limbajului SIMULA 67 nu impune nici o restricție relativ la utilizarea identificatorilor interiori unei clase, permițînd accesul la orice parte a reprezentării. O versiune ulterioară a introdus reguli de domeniu, asigurînd astfel un anumit grad de protecție a datelor.

b) In *CONCURRENT PASCAL* [BH75a, BH75b, BH75c], P. Brinch Hansen a extins noțiunea de tip din PASCAL prin definirea tipurilor sistem: **process**, **monitor**, **class** [BH76a]. Variabilele declarate de aceste tipuri se numesc *componente sistem*, iar programele se vor descrie ca o interconexiune de astfel de componente [BH76b,

CC80]. Un tip sistem, în general, reprezintă un modul distinct de programe, format dintr-un nume, o listă de parametri și un bloc (fig. 1.5.). Blocul conține descrierea unor date dar și a unor acțiuni corespunzătoare prelucrării lor (proceduri, funcții, corp de instrucțiuni).

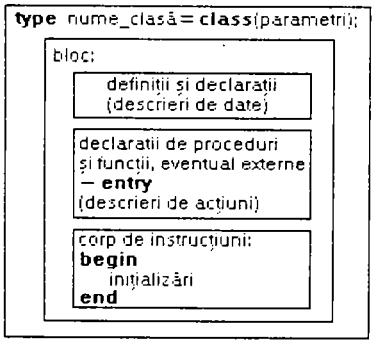


Fig. 1.5. Structura unei clase CONCURRENT PASCAL

Din punct de vedere sintactic nu există deosebiri esențiale între cele trei categorii de tipuri sistem. Semantic, procesele (**process**) sînt taskuri, adică activitățile paralele ale programului concurent iar monitoarele (**monitor**) sînt elemente de sincronizare între taskuri. Clasa, în schimb, reprezintă eminamente un mijloc de descriere a tipurilor abstracte, neavînd nici o altă semnificație suplimentară.

Clasa din CONCURRENT PASCAL este un tip. Obiectele corespunzătoare se declară în program ca variabile de acest tip. Aceasta corespunde punctului de vedere dinamic asupra tipurilor abstracte, prezentat anterior. Modificarea din exterior a datelor declarate în interiorul clasei se poate face exclusiv prin intermediul procedurilor și funcțiilor din clasa respectivă, indicate explicit ca externe (**entry**). In rest din punct de vedere al vizibilității, clasa reprezintă un domeniu închis: identificatorii definiți sau declarați în clasă nu pot fi referiți în exteriorul ei. Cu toate acestea modul în care este concepută definirea și utilizarea tipurilor sistem în ansamblul programului CONCURRENT PASCAL [BH77a, CE81c], exclude facilitatea de compilare separată.

Comparînd clasa din SIMULA 67 cu cea din CONCURRENT PASCAL se constată că prima este mai puțin restrictivă. Aceasta se explică și prin faptul că ea nu a fost gîndită ca instrument de abstractizare a datelor. Entitățile declarate în cadrul clasei SIMULA 67 se pot utiliza în întreg blocul în care este definită clasa respectivă. Nu se pun probleme de limitare a vizibilității descrierii, întreaga clasă fiind accesibilă utilizatorului. Pe lîngă o serie de greutăți și chiar inconveniente, restricțiile introduse în CONCURRENT PASCAL au și un mare avantaj: sporesc siguranța în funcționare a programului, ceea ce este esențial în programarea concurentă.

1.3.2. Pachetul

Pachetul (**package**) împreună cu taskul reprezintă formele specifice de modularizare a programelor, introduse în limbajul ADA [AN83, BA82, DD78, DD80a, DD80b]. Taskurile sînt elementele de descriere a activităților paralele din program și, din punctul de vedere al abstractizării datelor, au importante trăsături comune cu uneltele de abstractizare propriu - zise ale limbajului, pachetele.

Ca orice element de modularizare pachetul realizează modularizarea într-o entitate de program unitară a unor descrieri de date împreună cu operațiile de prelucrare aferente. Abstractizarea datelor realizată astfel corespunde punctului de vedere

denumit anterior static. Analizind structura generală a unui pachet, prezentată în fig. 1.6., se poate face o paralelă între noțiunea de pachet și aceea clasică de subprogram. In acest sens există trei deosebiri esențiale:

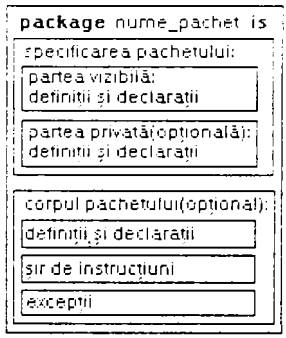


Fig. 1.6. Structura unui pachet (ADA)

1. *Corpul pachetului* poate să lipsească din declarație. Aceasta înseamnă că pachetul poate fi numai o simplă descriere de date, eventual și cu operațiile aferente, descrise ca proceduri în interiorul *părții de specificare*. In cazul în care corpul este prezent, acesta este elaborat complet în momentul în care programul ia în evidență pachetul. Se efectuează, deci, acele operații în care, în cazul subprogramelor se declanșează doar în momentul apelului și anume:

- rezervarea spațiului de memorie pentru obiectele declarate ca variabile,
- executarea inițializărilor prevăzute în declarații,
- executarea șirului de instrucțiuni din corp care, în esență, realizează de asemenea inițializări ale datelor, a căror complexitate n-a permis indicarea lor direct în declarații.

Mai trebuie remarcat și faptul că elaborarea părții de specificare (alocarea în memorie și inițializarea variabilelor declarate, luarea în evidență a procedurilor) se efectuează separat de elaborarea corpului pachetului, ceea ce permite și compilarea lor separată.

2. Identificatorii definiți sau declarați în *partea vizibilă* specificării de pachet pot fi accesibili în exteriorul pachetului (în domeniul asociat numelui său). Referirea din exterior a acestor identificatori se realizează prin calificarea cu numele pachetului. Restul pachetului (partea privată a specificării și corpul) este invizibil din exterior. Identificatorii definiți aici au domeniul limitat (în funcție și de eventuala lor suspendare) la cel mult interiorul pachetului în cauză. In acest fel se poate stabili în mod clar ce trebuie să cunoască și ce nu utilizatorul pachetului, prin plasarea informațiilor respective într-o parte sau alta [DR82]. De exemplu, referitor la un subprogram, utilizatorul trebuie să știe numele său, numărul, ordinea, tipul și semnificația parametrilor formali. Aceste informații, sub forma unei declarații parțiale vor fi incluse în partea vizibilă a pachetului. Declarația completă a subprogramului va fi indusă în corpul pachetului, inaccesibil utilizatorului. In cazul în care corpul se și compilează separat, declarația subprogramului este și fizic ascunsă pentru utilizator.

Specificarea de pachet, prin partea ei vizibilă, constituie deci interfața pachetului cu utilizatorii săi. Ea se compilează împreună cu porțiunile din program care utilizează pachetul și nu poate fi modificată fără a le afecta pe acestea. Corpul pachetului conține acele completări la specificare care nu interesează pe utilizator și care pot fi schimbate și recompilate ulterior, nederanjind prin aceasta programele deja realizate ce utilizează pachetul.

3. Variabilele declarate în cadrul unui pachet și în afara subprogramelor sale există în memoria calculatorului atîta timp cît este activ domeniul în care s-a declarat pachetul. Astfel de variabile pot reprezenta deci variabile proprii pentru un subprogram. Față de utilizarea în acest scop a variabilelor globale, pachetul prezintă avantajul introducerii unor restricții de vizibilitate care controlează accesul la variabilele respective din alte unități de program.

In același timp, variabilele declarate într-un pachet pot avea statut de variabile comune (de comunicare) între mai multe unități de program: subprograme, alte pachete, taskuri [GE83, WL81].

O noțiune importantă legată de aceea de pachet este cea de *tip privat*. Numele unui astfel de tip se indică în partea vizibilă a pachetului, iar definiția sa completă se precizează în *partea privată*, fiind inaccesibilă utilizatorului pachetului. In exteriorul pachetului (în domeniul numelui său), se pot declara obiecte de un anumit tip privat și se permit unele operații simple asupra acestor obiecte (asignarea cu valori de același tip, compararea de egalitate sau inegalitate, transmiterea ca parametri actuali ai unor subprograme indicate în partea vizibilă a aceluiași pachet.

Un caz special de tip privat este acela de *tip privat limitat* [TA85]. Operații asupra obiectelor din exteriorul pachetului, declarate de un astfel de tip, se pot efectua exclusiv prin intermediul subprogramelor interne pachetului (se transmit ca parametri actuali). Se poate constata că acest tip corespunde parțial îngrădirilor stabilite pentru un tip de dată abstractă, cu deosebirea că obiectele sînt întotdeauna exterioare blocului (pachetului) în care s-au declarat (sub formă de subprograme) operațiile de prelucrare [WE80].

In legătură cu noțiunea de pachet ca instrument de modularizare dar și de abstractizare a datelor în limbajul ADA, se mai pot face următoarele observații:

1. Implementarea unui tip de dată abstractă, în ADA, este mai greoaie întrucît el este distinct de pachetul prin care se realizează această operație.

2. Nu există posibilitatea ca același tip privat limitat să aibă mai multe implementări. Deși nu se interzice în mod explicit definirea mai multor corpuri pentru un pachet, nu există nici un mecanism care să selecteze unul dintre aceste corpuri, la un moment dat.

1.3.3. Modulul

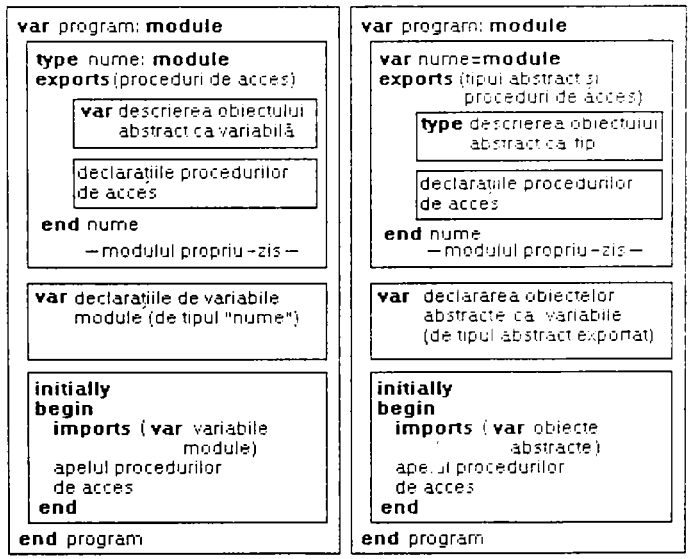
Modulul (**module**) este instrumentul de modularizare a programelor și de abstractizare a datelor cel mai răspîndit în limbajele de programare. Din punct de vedere istoric, noțiunea de modul a apărut înaintea celei de pachet, cu care prezintă importante similitudini. In această idee modulul poate fi analizat și înțeles prin analogie fie cu pachetul, fie cu clasa. In continuare se va face o scurtă trecere în revistă a definițiilor și implementărilor noțiunii de modul în diferite limbaje de programare: EUCLID, EDISON, MODULA (MODULA-2) și PASCAL PLUS.

a) *EUCLID* [LA77, CK78, HO83]. La fel ca și clasa din CONCURRENT PASCAL, modulul introdus în limbajul EUCLID este un tip abstract ce reprezintă reuniunea într-o construcție de limbaj unitară a unor structuri de date împreună cu operațiile definite asupra lor. Obiectele ce corespund datelor abstracte definite prin module pot fi declarate ca variabile de tipul abstract respectiv (fig. 1.7.a - punctul de vedere dinamic) sau ca variabile de un tip exportat din modul (fig. 1.7.b - punctul de vedere static).

Programul principal EUCLID este o variabilă modul care include toate celelalte definiții sau declarații de module. Acest modul (inițial) este înzestrat cu un corp de instrucțiuni propriu, pus în evidență de cuvîntul cheie **initially**, de unde se lansează programul în execuție (fig. 1.7.). In momentul lansării se crează, prin rezervarea spațiului de memorie corespunzător, toate obiectele declarate ca variabile globale în cadrul modulelor [HW82].

Entitățile definite sau declarate în interiorul unui modul, care pot fi referite în

exteriorul său, se pune în evidență prin *liste de export* (fig. 1.7.). Spre deosebire de clasa din CONCURRENT PASCAL, modulul EUCLID poate exporta și tipuri (analog cu tipul privat din ADA). O definiție de tip este exportată opac în afara modulului. Prin urmare, numele său este cunoscut și utilizat în afara modulului, dar reprezentarea sa nu. Se pot declara obiecte (variabile) de astfel de tipuri în exteriorul modulului (fig. 1.7.b), dar accesul direct la reprezentările lor este interzis. În consecință, prelucrările în care sînt implicate aceste obiecte se vor efectua indirect, prin operațiile definite sub formă de subprograme în cadrul modulului și exportate de către acesta.



a) Modul - tip b) Modul - variabilă

Fig. 1.7. Structura programului și a modulului EUCLID

Limbajul EUCLID privește modulul ca o generalizare a tipului articol, o structură care poate avea ca și componente: constante, variabile, tipuri și proceduri. Toți identificatorii care desemnează astfel de componente sînt locali modulului, exceptînd cazurile în care sînt exportați în mod explicit. Entitățile exportate sînt accesibile (prin calificare pe întreg domeniul asociat numelui modulului. Rezultă că și variabilele pot fi exportate. În principiu, asupra unor astfel de variabile, se acceptă efectuarea oricăror operații compatibile cu tipul lor, care nu le modifică valoarea. Pentru a se permite și modificarea valorii unei variabile exportate, acest fapt trebuie indicat în mod explicit în lista de export, prin cuvîntul cheie **var**.

Relațiile bazate pe import - export, în limbajul EUCLID, sînt valabile și în interiorul unui modul, pentru subprogramele interne acestuia. Astfel, o procedură declarată într-un modul, nu are acces direct la entitățile definite sau declarate în modulul respectiv. Ea trebuie să importe explicit chiar și aceste entități care, în alte limbaje de programare, sînt globale în procedură.

b) *EDISON* [BH81a, BH81b, BH82]. Limbajul de programare EDISON introduce cîteva reguli de bază deosebit de simple, din care se pot apoi deduce modalitățile de utilizare ale modulelor pentru eventuala reprezentare a datelor abstracte.

Programul principal EDISON este o procedură (inițială) care poate fi precedată, de definiții de constante și tipuri [BH81c]. Procedura inițială conține declarații de module și alte proceduri. La fel ca pachetul din ADA și spre deosebire de modulul EUCLID, modulul EDISON nu este un tip de dată. El reprezintă o parte distinctă a

programului care se declară în interiorul unei proceduri sau al altui modul și poate conține, la rîndul său, proceduri și module (fig. 1.8.). Ca și procedurile, modulele EDISON desemnează blocuri distincte. Spre deosebire de proceduri, modulele sînt blocuri anonime și nu pot fi activate din exterior decît prin apelul procedurilor declarate în interiorul lor.

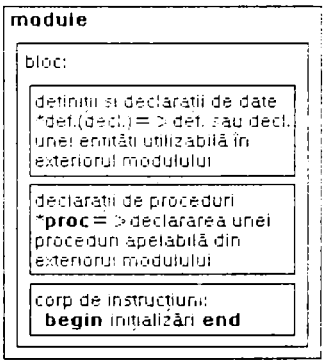


Fig. 1.8. Structura unui modul EDISON

Variabilele declarate în interiorul unui modul și în afara procedurilor sale, se crează la apelul procedurii care conține modulul și există în memorie atîta timp cît este activă această procedură [CE84a]. Tot cu această ocazie se execută și corpul de instrucțiuni propriu modulului, prin care se realizează, de obicei, operațiile de . țializare necesare.

In general, entitățile definite sau declarate într-un modul pot fi utilizate doar în interiorul lui. Unele dintre aceste entități (constante, tipuri, variabile sau proceduri) pot fi exportate și, deci, utilizate în afara modulului, prin indicarea caracterului '*' în fața definiției sau declarației (fig. 1.8.). Regulile de export în limbajul EDISON oferă o libertate suplimentară față de cele din alte limbaje [EC85a]. Astfel:

- variabilele exportate dintr-un modul pot fi utilizate fără nici o restricție în afara lui;
- importul unor entități nu se specifică explicit: toate mărimile vizibile în domeniul în care este declarat modulul sînt automat accesibile în interiorul acestuia.

Chiar și în aceste condiții, tocmai pentru a facilita descrierea unor tipuri abstracte, s-a introdus o restricție importantă: la exportul unor tipuri, reprezentarea nu este accesibilă din exterior (aspectul este important în cazul articolelor, pentru care se poate exporta doar identificatorul tipului, nu și cei ai cîmpurilor).

Intrucît variabilele și acțiunile proprii unui modul se crează și, respectiv, se execută în momentul apelării procedurii care îl conține, modulul acționează ca și o unitate de program distinctă doar în faza de compilare, cînd se verifică respectarea regulilor de domeniu și restricțiile impuse în legătură cu operația de export. Aceasta corespunde punctului de vedere static asupra datelor abstracte.

Libertatea oferită programatorului în limbajul EDISON și amintită anterior, prezintă o serie de inconveniente în ceea ce privește utilizarea în mod riguros a modulului ca mijloc de abstractizare a datelor. Acest aspect este însă, cel puțin parțial, compensat prin deosebita simplitate și flexibilitate a limbajului [EC84a].

c) *MODULA (MODULA-2)* [WI77a, WI77b, WI77c, WI82]. Limbajului PASCAL îi lipsesc complet, atît tipurile de date abstracte cît și facilitățile proprii pentru programarea concurentă [AS83]. Cu toate acestea el a stat la baza definirii multor limbaje moderne de nivel înalt (inclusiv a celor concurente) care au fost înzestrate în plus și cu modalități de descriere a datelor abstracte. Limbaje ca EUCLID, CONCURRENT PASCAL, PASCAL PLUS și chiar ADA sînt argumente în sprijinul

acestei idei. Creatorul limbajului PASCAL, N. Wirth, n-a putut rămîne indiferent la adevărata explozie care a avut loc vis-à-vis de definirea, pe baza propriei sale creații, a unei întregi familii de noi limbaje. Răspunsul său s-a materializat prin definirea în mod succesiv a două limbaje de nivel înalt, înzestrate cu facilități de abstractizare a datelor, numite MODULA și respectiv MODULA-2.

Programul MODULA este o colecție de module, eventual suprapuse, reprezentate prin blocuri. Un modul formează un domeniu închis. Fiecare identificator adus înăuntru, din afara modulului, trebuie să fie importat printr-o listă **use**. Analog, orice identificator intern care urmează să fie utilizat în exterior, va fi exportat printr-o listă **define**. La fel ca în ADA sau EUCLID, se pot exporta tipuri, a căror structură rămîne însă ascunsă și inaccesibilă pentru utilizatorul modulului. Variabilele de aceste tipuri declarate în exterior se pot manipula prin operații definite în cadrul modulului, organizate în proceduri exportate la rîndul lor.

O variabilă exportată poate fi utilizată în exterior, fără a i se modifica valoarea. Se poate considera deci, că varianta adoptată în EUCLID, de a indica explicit dacă o variabilă exportată poate fi sau nu modificată, reprezintă o generalizare a celei din MODULA. Corpul propriu de instrucțiuni al unui modul se execută atunci cînd se apelează procedura în care modulul este local (analog cu EDISON).

Modulul astfel definit de N. Wirth nu este un tip de dată și, ca atare, nu se pot declara variabile cu structura lui. El rămîne în principal un mecanism de restrîngere a domeniului și vizibilității care poate fi utilizat pentru reprezentarea datelor abstracte, prin export de tipuri, la fel ca pachetul din ADA (punctul de vedere static asupra datelor abstracte).

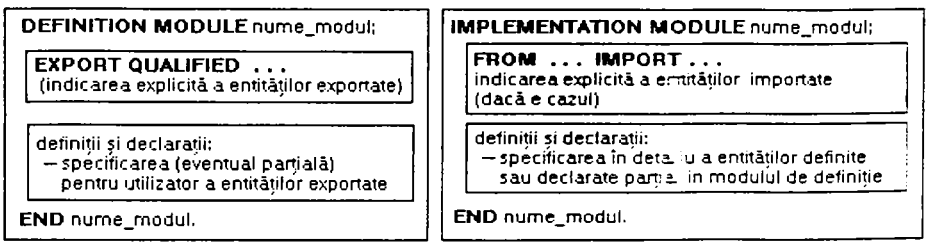
Variabilele locale într-un modul se crează la apelul procedurii în care el este declarat și dispar cînd se încheie activitatea acestuia (analog cu EDISON). Aceasta înseamnă că modulul acționează ca unitate de program distinctă doar în compilare, fără a se reflecta în mod dinamic prin obiecte abstracte corespunzătoare în execuție [HW80].

Principalele modificări aduse în MODULA-2 au fost următoarele:

1. În vederea introducerii unor facilități de compilare separată, s-au adoptat în plus două categorii speciale de module (fig. 1.9.):

- modulul de definiție (DEFINITION MODULE) în care se specifică toate entitățile exportate fără a fi, de regulă, definite sau declarate în detaliu;
- modulul de implementare (IMPLEMENTATION MODULE) care formează pereche cu modulul de definiție și conține definițiile sau declarațiile complete pentru entitățile specificate doar parțial în acesta din urmă.

Modulul de definiție (interfața cu utilizatorul) și cel de implementare (partea ascunsă utilizatorului) se pot compila separat. Doar primul din ele trebuie compilat împreună cu modulele care folosesc entități definite în interiorul lui. Analog cu ADA, modificarea implementării acestor entități, care afectează doar modulul de implementare, nu se repercutează asupra modulelor utilizator (și, prin urmare, nu necesită nici recompilarea acestora).



a) modulul de definiție b) modulul de implementare

Fig. 1.9. Structura modulelor specializate în limbajul MODULA-2

2. De regulă, exporturile se realizează din modulele de definiție iar importurile, în modulele de implementare sau în cele normale. O particularitate constă în aceea că, în cazul importului, se poate indica în mod explicit modulul de origine (fig. 1.9.b). Această facilitate servește la selectarea identificatorilor în cazul în care mai multe module exportă entități denumite la fel [HO80].

3. Spre deosebire de MODULA, unde tipurile se exportă exclusiv opac (nu este permis accesul din exterior la reprezentare) în MODULA-2 există și exportul transparent de tipuri. De exemplu, se pot exporta articole astfel încît cîmpurile sale să fie accesibile în exterior. Maniera în care se rezolvă această problemă este similară cu cea adoptată în ADA și anume [WS84]: dacă un tip exportat este specificat în întregime în modulul de definiție atunci se realizează un export transparent; dacă în modulul de definiție i se indică doar numele, iar definiția completă este dată în modulul de implementare, exportul său este opac. Și în general se poate aprecia că mecanismul de încapsulare din MODULA-2 este asemănător cu pachetul din ADA cu următoarele două deosebiri mai importante:

- nu a fost prevăzută posibilitatea declarării unor module generice;
- definiția modulului (interfața cu utilizatorul este separată de implementarea sa; în schimb echivalentul părții private în cadrul exportului opac al unor tipuri se include în modulul de implementare; ceea ce implică recompilarea modulului de implementare la eventualele modificări (aceasta se evită în ADA, incluzînd partea privată în specificarea pachetului).

d) *PASCAL PLUS* [WB79]. Reprezintă de asemenea o extensie a limbajului PASCAL în vederea programării concurente cu facilități de modularizare. Mecanismul de modularizare se numește plic (**envelope**) și este mai simplu și mai flexibil decît cele prezentate în limbajele anterioare. Flexibilitatea sa constă în principal în posibilitatea pe care o are programatorul de a defini fie un tipar de modul (analog cu tipul abstract clasă - CONCURRENT PASCAL), fie un modul propriu-zis (analog cu pachetul - ADA și modulele - EDISON și MODULA) (fig. 1.10.a și respectiv 1.10.b).

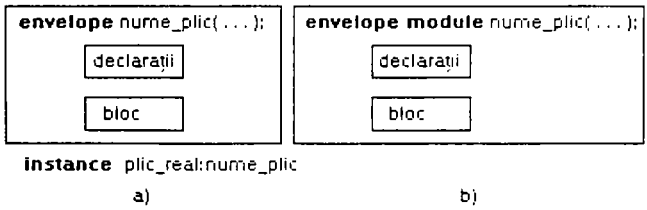


Fig. 1.10. Variante de declarare și utilizare a unui plic în limbajul PASCAL PLUS

Dacă se utilizează opțiunea **module**, atunci se declară un modul propriu-zis. In cazul în care opțiunea **module** lipsește plicul reprezintă un tip abstract iar exemplare de acest tip se declară într-o instrucțiune **instance** (fig. 1.10.a).

Identificatorii exportați se indică prin caracterul '*' (analog cu EDISON). Avantajul flexibilității acestui mecanism este compensat de dificultatea de separare a părții de specificare (interfața cu utilizatorul) de corpul plicului, complicînd eventualele intenții de compilare separată. O noutate interesantă care apare în limbajul PASCAL PLUS este indicarea explicită în corpul de instrucțiuni al corpului plicului, atît a operațiilor necesare a fi executate la crearea și inițializarea lui cît și a celor ce se impun la încheierea activității plicului.

1.3.4. Grupul

Grupul (**cluster**) reprezintă mecanismul definit pentru reprezentarea tipurilor de date abstracte în limbajul de programare CLU [LI77, LI79, LS79]. Acest limbaj a fost special conceput pentru a permite dezvoltarea programelor utilizînd abstractizări.

Grupurile sînt tipuri de date pe baza cărora se pot crea obiecte. Toate obiectele CLU sînt alocate dinamic iar rolul variabilelor este substanțial modificat față de celelalte limbaje de programare. Variabilele nu sînt obiecte ci doar referințe (nume) de obiecte în sensul pointerilor din PASCAL sau al variabilelor din LISP, de exemplu. Dacă o variabilă își schimbă starea aceasta nu înseamnă că și obiectul la care se referă și-a schimbat starea. O astfel de modificare echivalează cu schimbarea obiectului desemnat de variabilă. Deci, aceeași variabilă poate desemna la diferite momente de timp obiecte diferite. De asemenea, la un moment dat, este posibil ca același obiect să fie reprezentat (referit) de variabile diferite.

Variabilele CLU sînt locale procedurii în care se declară și nu pot fi accesibile din altă procedură. Obiectele există în mod independent de activitatea procedurilor. Spațiul de memorie aferent se alocă din aria de memorie dinamică și, teoretic, nu se relocă pînă la încheierea activității programului. Crearea obiectelor de un anumit tip este comandată prin program, invocînd printr-o anumită construcție de limbaj, tipul respectiv. În mod corespunzător statutului noțiunii de variabilă s-a adaptat și conținutul operației de atribuire [CE87a].

Atît din punct de vedere formal, cît și din cel al semnificațiilor atașate, definiția de grup se compune din trei părți distincte (fig. 1.11.):

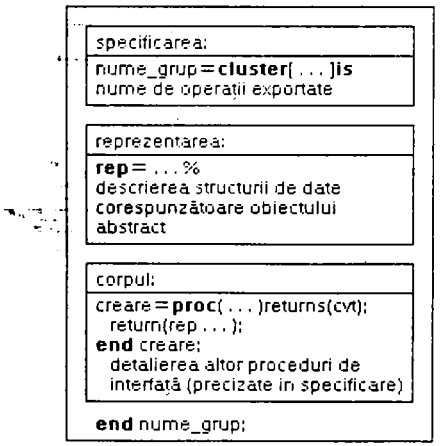


Fig. 1.11. Structura unui grup (CLU)

- *Specificarea*: indică numele grupului și numele operațiilor implementate în tipul abstract respectiv și accesibile din exterior (exportate); aceasta este partea vizibilă a grupului, adică interfața lui cu celelalte părți ale programului.
- *Reprezentarea*: este pusă în evidență prin cuvîntul cheie **rep** și realizează descrierea structurii de date reprezentată de grup. Această structură nu are un nume dat de programator, fiind deci și fizic inaccesibilă din exterior; toate operațiile de prelucrare a reprezentării trebuie implementate în interiorul grupului și, eventual, indicate în specificare, ca nume de proceduri exportate, pentru a fi apelate din exterior.
- *Corpul*: conține declarațiile complete ale procedurilor de interfață (date în specificare). O semnificație are în cadrul acestei părți, procedura *creare*; la apelul ei se crează (prin alocare dinamică) un obiect de structura reprezentării (**rep**) care va corespunde tipului abstract definit de grup.

În concluzie, oricărui grup i se asociază două tipuri diferite:
- tipul abstract propriu-zis (*nume-grup* în fig. 1.11.);
- tipul reprezentării sale (indicat în **rep**).

În exterior este accesibil exclusiv tipul abstract și, pe baza lui, se pot crea

obiecte. In interiorul grupului este necesar uneori să se identifice obiectul abstract cu tipul reprezentării, deoarece operațiile implementate sînt definite în termenii reprezentării. Această identificare se realizează prin indicarea cuvîntului cheie **cvt** utilizat ca tip, pentru acele entități care în interior sînt de tipul reprezentării, iar în exterior vor fi obiecte de tipul abstract definit prin grup.

Apelul procedurilor de interfață se obține prin calificarea identificatorilor procedurilor cu numele grupului, elementul de legătură fiind caracterul '$' (nu '.' care este notația consacrată în majoritatea limbajelor de programare). In acest fel se evită eventualele conflicte (ambiguități) la utilizare, atunci cînd mai multe grupuri furnizează aceiași identificatori de interfață.

Definițiile de grupuri nu pot fi suprapuse; este posibil în schimb ca grupurile să fie componente ale altor tipuri, inclusiv componente ale reprezentării altor grupuri. Definițiile propriu-zise sînt însă disjuncte între ele.

Specificarea de grup poate avea parametri formali tipuri și constante simple. In acest caz, fiecare utilizare a grupului va fi prevăzută cu indicarea parametrilor actuali corespunzători. Se pot preciza și anumite operații definite pe mulțimea reprezentată de un tip parametru formal (clauza **where**), operații utilizate în corpul grupului [LI79]. In principiu, utilitatea transmiterii de tipuri ca parametri trebuie privită prin analogie cu parametrii generici din limbajul de programare ADA [GE83].

Limbajul de programare CLU permite compilarea separată a grupurilor cu verificarea completă a compatibilităților de tip în timpul compilării. Pentru verificarea referințelor externe, se folosesc informațiile de interfață ale modulelor (compilate și bibliotecate) în care acestea sînt definite.

Conceptul de grup și, în general, facilitățile de abstractizare incorporate, fac din limbajul de programare CLU cel mai apropiat de cerințele programării orientate spre obiecte. Creatorii limbajului CLU afirmă, în mare măsură pe bună dreptate, că acesta permite scrierea unor programe clare ușor de modificat · de întreținut. In realitate se poate remarca o oarecare greutate la utilizare datorată asignării prin referință, care în plus, conduce și la un anumit grad de nesiguranță în menținerea și verificarea programelor prin nestăpînirea unor efecte secundare. De asemenea nu poate fi neglijat faptul că obiectele CLU (alocate exclusiv dinamic) nu sînt relocate în principiu niciodată, ceea ce poate fi inacceptabil în anumite aplicații.

1.3.5. Concluzii privind programarea bazată pe date abstracte

Calitatea unui program depinde în principal de priceperea celui care îl scrie. In cadrul acestui aspect, un rol deosebit îl joacă metodologia de programare utilizată. Limbajul de programare ales poate influența însă în mod decisiv eficiența aplicării unei anumite metodologii. O metodologie poate fi mai ușor sau mai greu de aplicat într-un limbaj dat, aceasta depinzînd de gradul de potrivire dintre construcțiile limbajului de programare și structurile impuse de metodologie. In plus, un limbaj de programare influențează modul în care utilizatorii gîndesc asupra programării. Din această cauză, apropierea limbajului utilizat de o anumită metodologie conduce aproape implicit la aplicarea acesteia. In programarea bazată pe abstractizarea datelor, îndeminarea utilizatorului se reflectă prin felul în care își alege abstractizările. Această alegere trebuie făcută astfel încît să se simplifice conexiunile între module, să se permită gîndirea și programarea lor independentă, ținînd cont doar de specificațiile de interfață. Trebuie avută în vedere de asemenea și posibilitatea de a schimba în timp acțiunile prevăzute într-o dată abstractă, fără ca prin aceasta să se afecteze restul programului.

Conceptele de date abstracte din diferite limbaje impun anumite restricții verificabile la compilare, dintre care esențială este, desigur, constrîngerea ca doar acțiunile interne, să opereze asupra reprezentării obiectului abstract. Deși uneori pot să pară incomode, aceste restricții sînt esențiale atît din punct de vedere teoretic, cît și practic. In principiu, abstractizările de date pot fi folosite în orice limbaj,

stabilind anumite convenții de programare care să protejeze reprezentările de obiecte. Convențiile însă nu pot înlocui restricțiile impuse prin limbaj. Este inevitabil ca ele să fie încălcate și probabilitatea de a le încălca este mai mare exact atunci cînd respectarea lor ar fi mai necesară, adică la programarea, întreținerea și verificarea programelor mari, la realizarea în echipă a unor produse program complexe.

Din aceste motive se poate concluziona că introducerea conceptelor de date abstracte ca instrumente specifice limbajelor de nivel înalt ușurează în mod decisiv munca programatorilor, conducînd implicit la creșterea eficienței acesteia.

1.4. Trăsături caracteristice programării orientate spre obiecte

In §1.2. s-a arătat legătura directă între programarea orientată spre obiecte și conceptul de dată abstractă. De altfel, actul de naștere al programării orientate spre obiecte este considerat definirea limbajului SIMULA [HO88] deși, după cum s-a subliniat și în paragraful 1..., primul reprezentant adevărat este mediul de programare SMALLTALK-80. In actualul deceniu, acest tip de programare a devenit un cadru potrivit pentru realizarea sistemelor de programe mari și foarte mari iar numărul și diversitatea mediilor și limbajelor de programare specifice sporește de la an la an.

Diferența de principiu între programarea orientată spre obiecte și cea tradițională este aceea că programele orientate spre obiecte sînt compuse din *obiecte* reprezentate prin date abstracte care comunică între ele prin mesaje. Un astfel de program cuprinde atît descrierea obiectelor cît și a relațiilor dintre ele. O altă diferență, din care derivă un important avantaj practic este reprezentată prin modul de interpretare al procedurilor. In programarea tradițională, efectuarea de operații (aceleași sau asemănătoare) asupra unor variabile de tipuri diferite se materializează prin proceduri diferite. Programatorul alege singur, în momentul apelului pe cea care corespunde tipului de date potrivit. Intr-un sistem orientat spre obiecte, astfel de operații se pot reprezenta printr-o singură procedură, alegerea variantei potrivite fiind rezolvată de sistemul suport al limbajului, pe baza parametrilor actuali ai procedurii.

1.4.1. Concepte de bază

Structura fundamentală într-un limbaj orientat spre obiecte este *obiectul*. In cazul unui limbaj orientat spre obiecte, pur, programele sînt alcătuite exclusiv din obiecte. Din această categorie face parte sistemul SMALLTALK [GR83]. Majoritatea sistemelor orientate spre obiecte nu sînt însă pure, provenind din completări aduse unor limbaje existente. Așa sînt, de exemplu limbajele COMMONLOOPS [BO85] și COMMONOBJECTS [SN85], derivate din COMMON LISP, în care se îmbină stilul de programare orientat spre obiecte cu programarea procedurală și cea funcțională.

In sistemele actuale orientate spre obiecte se utilizează o mare varietate de denumiri pentru aceleași concepte sau pentru concepte similare. Din motive de tratare unitară și pentru claritate, în continuare, se va utiliza cu predilecție terminologia din limbajul SMALLTALK-80 [GR83].

Structura și comportarea unei mulțimi de obiecte se descrie printr-o dată abstractă numită *clasă*. Pe baza unei clase, prin *instanțiere*, se pot crea obiecte. Un obiect se numește *instanță* a acelei clase. Instanțele care provin din aceeași clasă au aceeași structură și aceeași comportare dar pot avea stări diferite. O altă modalitate de creare de obiecte este prin *clonarea* unui prototip de obiect [AG86].

Structura fizică a unui obiect este definită de *variabilele instanței*. Starea obiectului se definește prin *metode* care sînt proceduri ce se execută ca răspuns la un *mesaj* trimis de alt obiect. Metodele pot acționa direct asupra variabilelor *receptorului* (obiectul spre care a fost trimis mesajul).

Numele mesajelor transmise către obiecte se numesc *selectori de mesaje*. Mesajele sînt înzestrate cu *argumente* care se trimit de-a lungul metodei. Pe baza selectorului și a argumentelor receptorul determină metoda de executat. Selectorii de mesaje reprezintă interfața obiectului cu sistemul în timp ce variabilele proprii și metodele definesc implementarea obiectului. Trebuie remarcat faptul că selectorul de mesaj este doar un nume pentru acțiunea solicitată indicînd *ce* trebuie să se execute. *Cum* trebuie să se execute ceea ce se solicită este atributul *receptorului*.

Mulțimea mesajelor ce pot fi tratate de un anumit obiect este definită în clasa obiectului respectiv și se numește *protocol de mesaj*. O clasă dată poate implementa mai multe protocoale și, de asemenea, același protocol poate fi definit în clase diferite. O operație care poate fi definită în clase diferite se numește *operație polimorfică*.

Procedeul prin care se realizează rafinarea descrierii obiectelor se numește *subclasare*. Structura și comportarea unei mulțimi de obiecte, definite printr-o clasă, *se moștenesc* în subclasele sale dar, în același timp pot fi extinse și/sau modificate. O anumită clasă, se numește *superclasă* pentru subclasele ei. Conceptele de clasă și moștenire s-au introdus inițial în limbajul SIMULA [DM68], de unde au fost preluate în sistemul SMALLTALK.

1.4.2. Particularități de programare

Programarea într-un sistem orientat spre obiecte este un proces de modelare. Programatorul descrie prin intermediul claselor, obiectele programului și definește mesajele și metodele utilizate în comunicarea dintre obiecte. Clasele se pot rafina prin subclasare, pornind de la un set de clase standard. In același timp, obiectele se definesc în mod natural, ierarhic, unele din altele, rezultînd un program clar și ușor de întreținut.

Utilizarea claselor cu date abstracte și specificarea interfețelor pentru comunicare reduc interdependențele dintre unitățile de program care pot fi modificate independent unele de altele. Implementarea unei clase este astfel vizibilă doar prin partea ei de interfață. In plus în unele limbaje, printre care și SMALLTALK, pot avea acces direct la variabilele clasei respective. Accesul indirect, din alte clase, se realizează prin transmitere de mesaje. In alte limbaje, variabilele unei clase se diferențiază în *publice* și *private*. Variabilele publice sînt accesibile și pot fi modificate din întreg programul iar cele private rămîn accesibile doar prin metodele obiectului. Din această categorie fac parte SIMULA și C++ [ST86].

Structura ierarhică a obiectelor programului și comunicarea prin mesaje de-a lungul ierarhiei permite ca o anumită operație să fie descrisă într-un sigur loc, indiferent de tipul operanzilor. Rezultă un program compact în care operațiile se pot localiza ușor sau se pot obține prin subclasare din clase standard.

1.4.3. Tipuri de moșteniri

Moștenirea este mecanismul cel mai puternic în limbajele orientate spre obiecte. Prin moștenire se definesc obiecte noi fie din obiectele existente, fie modificînd și combinînd descrierile claselor existente. Avantajele aplicării acestui concept sînt sintetizate în [TH87], astfel:

a) *Modelul realizat prin program este mai natural*. Intrucît ierarhiile specifice sînt părți componente ale realității înconjurătoare, modelarea directă a acestor ierarhii face ca structurile conceptuale ale programelor să fie mai firești și mai ușor de înțeles.

b) *Factorizarea*. Moștenirea permite ca proprietățile comune ale claselor să fie factorizate, (descrise o singură dată în cadrul ierarhiei programului și moștenite în toate locurile în care urmează să fie utilizate). In același timp rezultă și îmbunătățirea modularității programului.

c) *Se poate aplica tehnica de rafinare în pași succesivi atît pentru descrierea programului cît și la verificarea lui.* Se descriu și se verifică la început, clasele mai generale, conținînd proprietăți comune ale altor clase. Ulterior, prin tehnica top-down se dezvoltă clase specializate, modificînd și dezvoltînd clasele existente.

O clasă moștenește de la superclasele ei variabile și metode. Dacă, imediat după subclasare se crează o instanță a superclasei, întrucît comportarea ei este complet moștenită. Utilizînd superclasa ca și bază, atributele noii clase pot fi dezvoltate prin adăugare de variabile și metode noi. În SMALLTALK [GR83], variabilele unei clase sînt partajate atît de toate instanțele clasei respective, cît și, prin moștenire, de instanțele subclaselor.

Anumite sisteme [SC86] permit programatorului să specifice metodele care pot fi moștenite precum și cele vizibile în afara clasei în care sînt definite. Există astfel:

- operații *private*: nemoștenibile și invizibile pentru utilizatorii clasei;
- operații *publice*: moștenibile și vizibile;
- operații *subtip-vizibile*: pot fi moștenite dar sînt invizibile pentru utilizator.

Aceste diferențieri contribuie la creșterea clarității și siguranței în funcționare a programului.

În [HO88] se identifică trei tipuri de moșteniri:

1) *Moștenirea ierarhică.* Este specifică majorității limbajelor orientate spre obiecte printre care SMALLTALK și C++. Reprezintă o structură ierarhică simplă, strictă, în care o clasă poate moșteni doar dintr-o singură superclasă. S-a dovedit suficientă în practica programării pentru multe tipuri de aplicații. De exemplu, întregul sistem SMALLTALK-80 a fost realizat doar cu moștenire ierarhică. Este un mecanism simplu, eficient și direct, dar limitat în expresibilitate: nu se pot descrie ierarhic orice fel de relații.

2) *Moștenirea prin delegare.* Este specifică limbajelor în care obiectele se crează prin clonarea unui prototip de obiect [AG86]. În acest caz se moștenesc atît variabilele instanței cît și valorile lor curente. Dacă noul obiect va schimba valoarea uneia din variabilele moștenite, în prealabil, va realiza o copie a acelei variabile, valabilă numai pentru sine. Moștenirea prin delegare nu este legată de conceptul de clasă. În acest tip de moștenire, fiecare obiect este responsabil atît pentru alegerea mesajului care va fi tratat cît și pentru alegerea altui obiect care să trateze acele mesaje pe care el nu le poate rezolva. Rezultă un mecanism flexibil care poate simula alte tipuri de moșteniri care însă trebuie definite în mod explicit în program.

3) *Moștenirea multiplă.* Permite unei clase să moștenească din mai multe superclase. Față de moștenirea ierarhică reprezintă un mecanism extins, cu mai multă funcționalitate, obținut cu prețul creșterii complexității relațiilor în cadrul ierarhiei. Cea mai complicată schemă de moștenire multiplă o reprezintă *clasele booleene* [MZ86] în care orice combinație booleană a claselor existente poate defini o nouă clasă. Datorită noutății dar și complexității sale, nu există încă un sistem care să implementeze acest concept.

Capitolul 2

CONSIDERATII PRIVIND MODELAREA FORMALA A LIMBAJELOR DE PROGRAMARE

Metodele pentru descrierea limbajelor de programare existente în prezent îmbracă toată gama începînd cu limbajele naturale și terminînd cu forme ultramatematizate. In timp ce primele sînt neclare și de foarte multe ori ambigue, ultimele sînt greu de înțeles datorită abstractizărilor excesive. Foarte frecvent se utilizează o cale de mijloc: partea formală se limitează la exprimarea sintaxei (de exemplu BNF sau o formă echivalentă) iar restricțiile dependente de context și semantică se descriu în limbaj natural.

In acest capitol se analizează rolul descrierii formale complete a unui limbaj de programare atît pentru definirea și înțelegerea corectă a limbajului sub toate aspectele sale cît și pentru implementarea limbajului respectiv, prin proiectarea și realizarea compilatorului. Dat fiind faptul că reprezentarea formală a semanticii limbajelor de programare rămîne în continuare o problemă deschisă studiului și cercetărilor se insistă în special asupra ei. După o trecere succintă în revistă a principalelor direcții și metode de modelare formală cristalizate în literatura de specialitate, se detaliează unele aspecte ale modelării algebrice. Pe baza unor criterii definite anterior, în final se realizează compararea metodelor prezentate, rezultînd unele concluzii.

2.1. Obiectul descrierii formale a limbajelor de programare.

Un limbaj de programare poate fi specificat formal prin tripletul:

$$L=<Sm,St,f:Sm\rightarrow St>,$$ unde:

- Sm reprezintă semantica limbajului;
- St este sintaxa sa;
- f este funcția de asociere a semanticii, unei sintaxe date.

Este evident că o descriere formală completă a limbajului trebuie să trateze toate aceste aspecte. Introducerea conceptului de gramatică [CH59] și definirea metalimbajelor de tip BNF [NA63] au condus la dezvoltarea teoriei limbajelor formale. In cadrul acestei teorii, mecanismele de generare a formelor sintactice au fost fundamentate matematic. In același timp aceasta a permis și rezolvarea practică a problemelor analizei sintactice, în procesul translatării programelor [SE87, AU78].

Problemele ridicate de definirea formală a semanticii și de asociere a ei cu sintaxa s-au dovedit a fi mult mai dificile. Dificultățile sînt cauzate atît de complexitatea deosebită a acestor probleme cît și de faptul că, o lungă perioadă de timp limbajele de programare au fost definite în mod artizanal. Cu toate că, în prezent, conceptul de limbaj de programare, structura și componentele sale, sînt corect definite și bine fundamentate teoretic [RU83], definirea noilor limbaje de programare ar trebui să țină cont, din motive comerciale și de utilizare a experienței anterioare, de compatibilitatea cu sistemele de programare existente. Specificarea semanticii unui limbaj de programare reprezintă, în principiu, descrierea regulilor de translatare a limbajului respectiv în alt limbaj (metalimbaj) al cărui model semantic este cunoscut.

2.2. Criterii de evaluare a eficienței metodelor pentru descrierea formală a limbalelor de programare.

Primele limbaje și, evident, primele compilatoare au apărut și chiar s-au dezvoltat considerabil înaintea inventării și dezvoltării tehnicilor formale pentru descrierea limbajelor de programare. Acest fapt a avut consecințe negative asupra implementărilor limbajelor respective [MA76, CE88b] și anume:

- sintaxa și în special semantica unor instrucțiuni sînt, deseori, interpretabile.
- eforturile de standardizare a limbajelor de programare au fost amplificate.

- cu toate că, în prezent, există standarde pentru multe limbaje, portabilitatea programelor lasă de dorit, chiar și între două implementări diferite pe același calculator.

- este imposibil să se garanteze la realizarea unui compilator că acesta va asigura implementarea exactă a limbajelor.

- unele probleme de amănunt privind un limbaj de programare sînt tratate diferit de la o implementare la alta, motiv pentru care trebuie apelat la rularea de programe exemplu pentru a le înțelege.

In principiu, orice metodă de definire formală a limbajelor de programare, constă din următoarele elemente [SC72, MA76]:

i) definirea unui alfabet de simboluri de bază (A) și, implicit sau explicit, definirea mulțimii A^* reprezentînd toate șirurile de simboluri posibile care pot fi construite din A;

ii) furnizarea unui set de reguli pentru selectarea mulțimii $P \subseteq A^*$ a programelor corecte în limbajul ce se definește;

iii) specificarea înțelesului fiecărui element $p \in P$.

Metodele de definire diferă substanțial între ele prin modul în care se selectează programele corecte și se specifică înțelesul lor. Există două puncte de vedere principial diferite în ceea ce privește semnalarea erorilor: explicit, în definiție, sau implicit, specificînd regulile pentru generarea doar a programelor corecte. Se pot, de asemenea, include unelte pentru indicarea unor restricții lăsate la latitudinea implementărilor. In fine, din punctul de vedere al implementatorului limbajului, contează și ușurința cu care metoda respectivă permite realizarea sau, eventual, generarea automată a compilatorului [MK70, MC72]. Utilitatea metodei poate fi deci apreciată în funcție de calitatea răspunsurilor pe care le oferă definiția la problemele de mai sus. Se pot extrage următoarele criterii de apreciere [MA76, CE88b]:

- *Completitudinea*: reprezintă calitatea metodei de a acoperi toate problemele legate de sintaxa și semantica unui limbaj.

- *Simplitatea*: este măsurată prin ușurința cu care se pot realiza modele cît mai simple, indiferent de complexitatea limbajului.

- *Claritatea*: constă în înțelegerea ușoară a definițiilor; pentru aceasta metoda trebuie să permită descrierea cît mai naturală a limbajului, chiar cu riscul perderii de concizione.

- *Expresivitatea la erori*: este capacitatea metodei de a permite depistarea erorilor din programe.

- *Maleabilitatea*: constă în indicarea clară în definiție a locurilor în care se pot introduce restricții sau opțiuni lăsate la latitudinea implementării; ea este legată și de capacitatea metodei de a cuprinde toate detaliile.

- *Modificabilitatea*: reprezintă ușurința cu care se pot efectua modificări în descrierea anterioară a unui limbaj; este importantă numai în faza definirii limbajului.

- *Aplicabilitatea la implementare*: este măsurată prin apropierea metodei de tehnicile utilizate în proiectarea și scrierea compilatoarelor.

2.3. Clasificarea metodelor de definire formală a limbajelor de programare. Exemple

Deși definirea formală a semanticii este o problemă relativ nouă (1969 - Vienna Definition Language), pînă în prezent s-au conturat clar cîteva direcții [CE89b]. După modalitatea în care se specifică semantica programelor, metodele de definire formală pot fi clasificate astfel:

2.3.1. Metode operaționale

a) *Modelarea de tip interpretativ*: semantica este definită prin prelucrările corespunzătoare programului în timpul execuției.

Reprezentantul tipic al acestor modele este *Limbajul de la Viena* (VDL). In această metodă [WE72], programul este transformat într-o structură arborescentă, în conformitate cu sintaxa limbajului. In faza următoare, arborele este convertit într-o formă abstractă, executabilă pe o mașină virtuală. Din punctul de vedere al criteriilor enunțate în paragraful anterior, metoda este completă, expresivă la erori și maleabilă. Claritatea și modificabilitatea sînt satisfăcătoare. In schimb, complexitatea foarte ridicată o face greu de înțeles și de aplicat.

b) *Modelarea de tip compilativ*: semantica este reprezentată prin programe scrise într-un alt limbaj. S-au dezvoltat mai multe metode de acest tip, dintre care se amintesc:

- *Metoda gramaticilor W* (WGM). Gramaticile W [WJ69] sînt organizate pe două nivele, corespunzătoare sintaxei și respectiv semanticii unui limbaj. Nivelul doi asigură translatarea programelor corecte sintactic în alt limbaj (țintă). Referitor la criteriile de evaluare, metoda este completă și simplă. Din punct de vedere al claritătii, metoda corespunde doar semantic (nu și sintactic). Maleabilitatea și modificabilitatea sînt satisfăcătoare în timp ce expresivitatea la erori este insuficientă.

- *Metoda gramaticilor de atribute* (AGM). In această metodă [KN71, ȘE87] semantica limbajului, exprimată prin efectul execuției programelor, poate fi descrisă în termenii unui alt limbaj de programare. Gramatica de atribute este o gramatică independentă de context ale cărei simboluri din arborele de derivare au asociate atribute semantice. Metoda este incompletă și relativ incomplexă. Oferă o bună claritate sintactică iar claritatea semantică, precum și aprecierile după celelalte criterii sînt satisfăcătoare.

- *Metoda gramaticilor generative* (GGM). Poate fi considerată metodă de tip compilativ deși permite doar exprimarea sintaxei. Este un mecanism mai puternic decît BNF prin care se poate descrie atît mulțimea programelor corecte sintactic cît și translația lor într-un limbaj țintă [LE77]. Netratînd și semantica limbajului, este o metodă incompletă. Modificabilitatea este satisfăcătoare. Din punctul de vedere al celorlalte criterii, performanțele sînt bune.

2.3.2. Metode matematice

a) *Modelarea axiomatică*: semantica este descrisă prin calculul predicatelor din logica matematică. O metodă de această natură, *Abordarea axiomatică* (AAM), a fost definită în [HW73] doar pentru exprimarea semanticii fiind, deci, din acestă cauză, incompletă. Restul performanțelor sînt satisfăcătoare. Se poate utiliza la descrierea completă a unui limbaj de programare împreună cu metoda gramaticilor generative. In [MA76] se găsește un exemplu în acest sens.

b) *Modelarea funcțională*: semantica este definită prin clase de funcții matematice. In metodele din această categorie (FCM), s-a ales pentru modelare clasa funcțiilor parțial recursive [RU83]. Deși modelul rezultat este corect din punct de vedere matematic, el nu poate să reflecte complet și fidel modelul real al unui limbaj de programare definit anterior, fără să țină cont de această metodologie. Apar, de asemenea, probleme deosebite la încercarea de a exprima, în mod unitar, atît sintaxa cît și semantica limbajului. Expresivitatea la erori și maleabilitatea sînt nesatisfăcătoare.

c) *Modelarea algebrică*: semantica este reprezentată prin structuri algebrice. Acest tip de modelare (ALM) se tratează în detaliu în paragraful următor.

2.4. Modelarea algebrică a limbajelor de programare.

2.4.1. Particularități. Avantaje.

Dezvoltarea algebrei abstracte [ȘA89] și elaborarea în cadrul ei a conceptului de

algebră heterogenă [HI64] a permis formularea matematică a noțiunii de limbaj de programare. Pe această bază a fost posibil să se definească în mod natural, utilizînd același mecanism și aceeași metodologie, atît sintaxa cît și semantica limbajului.

Referitor la modalitatea de descriere a semanticii, se pot identica două direcții de definire algebrică a limbajelor de programare [BR87, CE89b]:

i) *Metode transformaționale.* Se bazează pe axiome care transformă limbajul inițial, extins, într-un alt limbaj restrîns, pentru care semantica este cunoscută. Metoda poate fi utilizată și în cazul celor mai complexe concepte de programare, inclusiv pentru programarea concurentă. Astfel fiecare program concurent poate fi transformat într-un program secvențial echivalent din punct de vedere funcțional, exprimat în termenii limbajului restrîns.

ii) *Metode bazate pe tipuri abstracte de date.* Tipurile abstracte, introduse inițial prin clasa din limbajul SIMULA-67, sînt prezente sub diferite forme în multe din limbajele de programare noi (CONCURRENT PASCAL, EUCLID, ALPHARD, ADA, CLU, ș.a.). Ele au contribuit la dezvoltarea ingineriei software, influențînd în general practica programării [CE87a]. Utilizat ca și concept matematic abstract, tipul abstract de date permite aplicarea formalismului algebric la specificarea limbajelor de programare. Astfel limbajul poate fi definit pur algebric ca un tip abstract, semantica fiind reprezentată de clasa de izomorfism a modelelor acestui tip. Metoda se recomandă atît pentru limbajele de programare procedurale cît și pentru cele funcționale. Modelul semantic poate fi unic (pînă la izomorfism) sau pot fi mai multe modele grupate în clase neizomorfe. Metoda algebrică permite astfel să se compare diferite modele semantice și să se analizeze relațiile dintre ele.

Utilizarea modelelor algebrice la specificarea limbajelor de programare are următoarele avantaje principale:

- Modelele rezultate sînt eficiente și corecte atît din punctul de vedere al programatorului cît și al implementatorului limbajului.
- Același mecanism formal permite generarea sintaxei, descrierea semanticii precum și exprimarea legăturii între aceste două aspecte.
- Permite abordarea cu succes și a altor probleme înrudite, cum ar fi: generarea și verificarea automată a programelor.

2.4.2. Definirea noțiunilor matematice de bază utilizate în specificarea algebrică a limbajelor de programare.

Definiția 1 (mulțimi, funcții, relații multi-sortate). Fie o mulțime M. O mulțime (funcție, relație) M-sortată este o familie $X=\{x_m\}_{m\in M}$ de mulțimi (funcții, relații) indexate de M.

Pentru simplificare se poate adopta relația fără utilizarea explicită a indicilor. De exemplu $z\in X$, în loc de $z\in x_m$ cu $m\in M$ sau $f:X\to Y$, în loc de $f=\{f_m\}_{m\in M}$ și $f_m:x_m\to y_m$ pentru $m\in M$.

Definiția 2 (semnătură). Fie I și S două mulțimi finite de simboluri astfel încît $I\cap S=\Phi$. I^+ și S^+ sînt semigrupurile libere generate de mulțimile I și respectiv S înzestrate cu operația de concatenare. O *semnătură* $\Sigma\subseteq I^+\times S^+$ este o relație binară astfel încît dacă perechea $(i,s)\in\Sigma$ cu $i\in I^+$ și $s\in S^+$, atunci $\lambda(i)=\lambda(s)$, unde [RU83, ST88]:

- i se numește tip de operație;
- s este simbol de operație;
- $\lambda(i)-1$ reprezintă aritatea operației;
- tripletul $\langle\lambda(i)-1,\ i_1i_2...i_{\lambda(i)},\ s_1s_2...s_{\lambda(i)}\rangle$ se numește schemă de operație.

Definiția 3 (algebra heterogenă). Fie I și S două mulțimi index introduse ca în def. 2 și $\Sigma=\langle I,S\rangle$, semnătura corespunzătoare. O *algebră heterogenă* sau o Σ-algebră heterogenă [HI64] este o structură reprezentată prin tripletul: $\mathcal{A}=\langle A=(a_i)_{i\in I},\ \Sigma,\ F\rangle$ unde:

- A este o familie de mulțimi $A=(a_i)_{i\in I\circ\subseteq I}$, indexată de I, numită familia tipurilor intrinseci (elementare) sau familia de mulțimi mesager.

- F este o funcție care asociază fiecărui element $(i,s)\in\Sigma$, cu $i\in I$ și $s\in S$, o operație heterogenă în familia A, al cărui domeniu și codomeniu sînt specificate prin schema de operație asociată perechii respective. Algebrele heterogene pot să fie totale sau parțiale după cum funcțiile F corespunzătoare sînt totale sau parțiale [BR87, ST88].

- perechea $\Sigma=\langle I,S\rangle$ de mulțimi index se numește semnătura algebrei $\mathcal{A}$.

Considerînd două Σ-algebre heterogene $\mathcal{A}$ și $\mathcal{B}$, definite astfel:

$\mathcal{A}=\langle A=(a_i)_{i\in I}, \Sigma, F_A\rangle$ și

$\mathcal{B}=\langle B=(b_i)_{i\in I}, \Sigma, F_B\rangle$

se poate defini un Σ-homomorfism de la $\mathcal{A}$ la $\mathcal{B}$, ca o familie de funcții $H:A\to B$, $H=(h_i)_{i\in I}$, unde $h_i:a_i\to b_i$, astfel încît pentru fiecare $i\in I$ și $x_1\in a_{i1}$, ..., $x_n\in a_{in}$, $F_A(x_1, ..., x_n)$-definită rezultă $F_B(h_{i1}(x_1), ..., h_{in}(x_n))$-definită și $h_i(F_A(x_1, ..., x_n))=F_B(h_{i1}(x_1), ..., h_{in}(x_n))$.

Conform [BR87], acesta este un homomorfism total.

Din punct de vedere matematic, mulțimea Σ-algebrelor (parțiale) heterogene formează o categorie [ST88, ȘA89] care are Σ-algebre ca obiecte și Σ-homomorfisme ca și morfisme; compunerea homomorfismelor este compunerea componentelor lor ca și funcții. Pentru reprezentarea semanticilor limbajelor de programare, în literatura de specialitate se consideră de obicei categoria Σ-algebrelor complete, ordonate, totale, împreună cu homomorfisme continue între ele [BR87].

Definiția 4 (tip abstract, subtip, ierarhie). Un *tip abstract* $T=\langle\Sigma,M\rangle$ constă dintr-o semnătură Σ și o mulțime M de formule cuantificate universal, numite axiome.

Un tip abstract $T_1=\langle\Sigma_1,M_1\rangle$ este numit *subtip* al lui T, dacă $\Sigma_1\subseteq\Sigma$ și $M_1\subseteq M$.

Un *tip ierarhic* este reprezentat printr-o pereche $\langle T,T_1\rangle$, unde T_1 este un subtip al lui T; T_1 se numește *tip primitiv* al lui T.

Definiția 5 (modelul unui tip, tip monomorfic). Fie $T=\langle\Sigma,M\rangle$ un tip ierarhic și T_1 un tip primitiv al lui T.

a) O Σ-algebră $\mathcal{A}$ se numește **model** al tipului T dacă sînt îndeplinite următoarele condiții [BR87]:

- $\mathcal{A}$ este generatoare de termeni;
- axiomele lui T sînt valide în $\mathcal{A}$;
- restricția lui $\mathcal{A}$ la familia tipurilor elementare și la operațiile din T_1 este de asemenea generatoare de termeni (de către T_1);
- valorile logice "true" și "false" sînt amîndouă definite în $\mathcal{A}$ (și diferite una de cealaltă).

b) T se numește *tip monomorfic* dacă are un model unic (de izomorfism); dacă are cel puțin un model, T este *satisfăcător*.

Ultimele două condiții din definiția modelului se numesc *constringeri ierarhice*. Ele asigură selectarea doar a algebrelor semnificative diferite de algebra trivială și care posedă o ierarhie propriu-zisă.

Definiția 6 (echivalență puternică, echivalență vizibilă). Într-un tip ierarhic T, doi termeni t_1 și t_2 se numesc *puternic echivalenți* dacă $t_1=t_2$ este adevărată în toate modelele lui T. Doi termeni de bază neprimitivi t_1 și t_2 se numesc *vizibili echivalenți* dacă pentru toate contextele primitive [BR87] p, termenii $p[t_1]$ și $p[t_2]$ sînt puternic echivalenți.

Echivalența puternică se referă la structura internă a modelului unui tip abstract; noțiunea de echivalență vizibilă privește tipul abstract din exterior, independent de structura sa internă (tipul T este interpretat ca o "cutie neagră").

2.4.3. Modelul algebric al unui limbaj de programare bazat pe o ierarhie de tipuri abstracte de date

Un principiu important în modelarea formală a limbajelor de programare este acela că: "echivalența programelor" nu se referă la reprezentarea textuală ci la înțelesul (semantica) lor. Pentru a se putea decide asupra echivalenței (vizibile),

aceasta trebuie în mod corect (și finit) axiomatizată. Tipurile abstracte, utilizînd pentru axiomatizare funcții parțiale, permit modelarea eficientă a limbajelor de programare. Definirea tipurilor abstracte, în ingineria softului, ca elemente de modularizare, implică structurarea lor ierarhică. Trebuie avut în vedere totuși faptul că aceeași ierarhie specifică de obicei o întreagă clasă de modele de limbaje (posibil neizomorfe).

i) *Reprezentarea sintaxei.* In descrierea unui limbaj de programare bazată pe tipuri abstracte, sintaxa (independentă de context) corespunde semnăturii tipului. Semnătura se construiește pornind de la forma normală Backus, prin asocierea fiecărui neterminal <i> cu un tip de operație i∈I(§2.4.2., def.2). In același timp, fiecărei reguli de producție P îi corespunde o operație p. Considerînd o regulă P de formă ($<i_k>$ - neterminal și t_k - terminal): $<i_0>=t_0<i_1>t_1 \ldots <i_n>t_n$, atunci p poate să fie reprezentat ca o funcție definită astfel: $p:i_1 \times i_2 \times \ldots \times i_n \to i_0$.

O anumită simplificare s-ar putea obține prin restrîngerea tipurilor de operații doar la neterminalele semnificative și neglijarea simbolurilor auxiliare.

Semnătura obținută prin această metodă se numește sintaxă abstractă. Algebra termenilor unei sintaxe abstracte [BL70] este izomorfă față de algebra arborilor de analiză gramaticală. In consecință ea reflectă toate detaliile strategiei acestei analize.

ii) *Reprezentarea semanticii.* Considerînd același tip abstract utilizat pentru definirea sintaxei, restricțiile sensibile la context (semantica statică) și semantica efectivă (dinamică) pot să fie specificate printr-o mulțime de ecuații condiționale (axiome de tipuri). Aceste ecuații permit reducerea algebrei termenilor sintaxei abstracte la o subalgebră a programelor corecte din punct de vedere semantic. Cele mai importante aspecte care trebuie luate în considerare la specificarea semantică sînt:

a) Specificarea efectului unui fragment de program pe baza efectelor componentelor sale. Acest principiu corespunde în mod natural cu noțiunea homomorfism din algebră.

b) Înțelesul unui program poate fi reprezentat prin comportarea sa vizibilă. De exemplu, comportarea vizibilă a unui program secvențial este o relație între intrare și ieșire. Pentru un program concurent, comportarea vizibilă este determinată de sincronizarea și comunicarea între procesele sale; pentru programe în timp real trebuie luate în plus în considerare intervalele de timp dintre acțiuni.

Modelul semantic trebuie să permită distincția programelor cu comportare vizibilă diferită. Intr-un model abstract complet, două programe sînt reprezentate prin același obiect, dacă și numai dacă au aceeași comportare vizibilă. Această cerință poate să fie exprimată în mod natural utilizînd o ierarhie de tipuri abstracte (§2.4.2., definiția 4) și conceptul de echivalență vizibilă (§2.4.2., definiția 6).

2.5. Concluzii

In tabelul 2.1. se prezintă schematic o analiză comparativă a metodelor prezentate succint în acest capitol din perspectiva criteriilor enunțate în §2.2. S-au utilizat următoarele prescurtări: bună (B), satisfăcătoare (S), nesatisfăcătoare (N), imposibil de aplicat (I).

In concluzie, nu există în momentul de față o metodă ideală. Dintre formalismele analizate, VDL se apropie cel mai mult de o definiție de limbaj care să corespundă cerințelor practice privind definirea, implementarea și utilizarea limbajului. Rezultate bune s-ar putea obține și din combinarea metodelor GGM și AAM. De asemenea, metodele de tip compilativ, în general, au avantajul apropierii firești de tehnicile de compilare utilizate în prezent.

Aplicarea conceptului matematic de algebră abstractă ca instrument de modelare a avut o mare importanță în cercetările privind descrierea și definirea limbajelor de programare, determinînd o puternică emulație în orientarea acestor cercetări pe baze noi. In ciuda acestui fapt, efectul practic, de îmbunătățire a tehnologiilor de

producere a programelor, nu a fost cel scontat. Una din cauze este probabil aceea că noul instrument matematic (algebra heterogenă) se aplică la specificarea sintaxei și semanticii unui concept încă incomplet formalizat (limbajul de programare). O anumită contribuție are, de asemenea, și tradiția dezvoltării istorice a limbajelor de programare care determină apariția unor contradicții matematice la descrierea diferitelor modele.

Tabelul 2.1.

Metode \ Criterii			Completitudine	Simplitate	Claritate sintactică	Claritate semantică	Expresivitate la erori	Maleabilitate	Modificabilitate	Aplicabilitate la implementare
Metode operaționale	de tip interpretativ	VDL	B	N	S	S	B	B	S	S
	de tip compilativ	WGM	B	B	N	S	N	S	S	B
		AGM	N	S	B	S	S	S	S	S
		GGM	I	B	B	I	B	B	S	B
Metode matematice	axiomatice	AAM	I	S	I	S	S	S	S	I
	funcționale	FCM	S	S	S	S	N	N	B	N
	algebrice	ALM	B	S	B	B	N	N	S	N

Alte direcții importante de valorificare a definițiilor formale care au fost amintite, fără a fi însă tratate în acest capitol, sînt generarea automată și verificarea automată a corectitudinii programelor.

Capitolul 3

Limbaje de nivel înalt pentru specificarea și controlul activităților concurente

Avînd în vedere o multitudine de avantaje [TS85] și care conduc în ultimă instanță la creșterea eficienței activității de programare, limbajele de nivel înalt au înlocuit treptat limbajele de asamblare în marea majoritate a aplicațiilor din toate domeniile în care se utilizează calculatorul. Din această tendință evidentă s-a născut, la mijlocul deceniului trecut, *familia limbajelor de nivel înalt pentru programarea concurentă* sau, pe scurt, *limbajele concurente*. Caracteristica esențială a acestor limbaje este aceea că prezintă facilități speciale pentru descrierea și controlul proceselor concurente.

Principalele domenii de utilizare a limbajelor concurente sînt:

- realizarea de sisteme de operare sau alte aplicații concurente (gestiunea unor baze de date, programe pentru supravegherea și conducerea unor procese industriale, etc.) pe sisteme mono și multiprocesor;
- programarea în timp real;
- programarea sistemelor distribuite.

Primul limbaj din această categorie a fost CONCURRENT PASCAL [BH75a, CE85a], definit de P. Brinch Hansen în 1975. In anii imediat următori au apărut cîteva limbaje concurente destinate în special programării în timp real (PORTAL [LI78], MODULA [WI77a]). Altele (CSP [HO78], DP [BH78]) au fost definite nu atît pentru scrierea de aplicații cît pentru ilustrarea unor concepte teoretice noi, prezentate în contextul unui limbaj de nivel înalt.

Tendința actuală este aceea de a înzestra noile limbaje de programare universale și cu facilități pentru programarea concurentă. Exemple semnificative în acest sens sînt limbaje ca: EDISON [BH81b], MODULA-2 [WI82] și ADA [AN' ,].

3.1. Procese concurente

Procesul sau taskul este reprezentat printr-o succesiune de acțiuni executate una singură la un moment dat. *Programul concurent* se compune din cel puțin două procese paralele care interacționează și se condiționează reciproc.

Orice program secvențial poate fi considerat ca fiind reprezentat printr-un singur proces care utilizează diferite resurse ale calculatorului, cîte una la un moment dat. Prezența unui singur proces în lucru, căruia îi sînt afectate toate resursele sistemului, ar avea ca rezultat un grad slab de utilizare a echipamentelor. De asemenea nu s-ar putea profita de eventuala existență a mai multor *procesoare fizice* (unități de comandă, unități aritmetice, procesoare de intrare/ieșire, etc.) în scopul exploatării lor simultane. Chiar și în cazul imposibilității realizării unui paralelism fizic (din lipsă de procesoare), prezența mai multor procese simultan în memorie, cărora să li se asocieze simultan procesorul (de exemplu prin divizare de timp) poate fi utilă [LD81]. In acest caz există un paralelism logic, putîndu-se considera din punct de vedere abstract că procesele se execută în paralel.

Considerentele de mai sus au determinat elaborarea sistemelor de operare cu multiprogramare. Acestea permit prezența simultană în memoria centrală a mai multor programe ce se execută în paralel. Gradul de paralelism fizic obținut depinde de numărul și tipul procesoarelor fizice conectate la memoria centrală. Utilizarea în comun de către mai multe programe aflate în memorie a resurselor sistemului, conduce la o îmbunătățire spectaculoasă a gradului de utilizare a acestora.

Programele rulate sub sistemele cu multiprogramare formează prin urmare un set de procese paralele executate independent între ele. Comportarea lor individuală este practic identică celei din cazul rulării fiecăruia într-un sistem monoprogramat. Eventualele conflicte rezultate din utilizarea comună a unor resurse se rezolvă prin sistemul de operare, fără ca programatorul să ia vreo măsură în acest sens.

Programele care reprezintă procese izolate (programe secvențiale) nu permit rezolvarea unor anumite categorii de probleme. Deseori devine necesară prezența simultană a mai multor procese, executate în paralel și asincron, dar între care există relații de cooperare (schimb de mesaje, transfer de date) și care gestionează în comun resurse ale sistemului. Astfel de *procese* se numesc *paralele* sau concurente. In principiu, două procese sînt concurente dacă executarea lor se suprapune în timp sau, mai exact, dacă un proces este lansat înaintea încheierii activității celuilalt proces. Executarea proceselor concurente care în marea majoritate a timpului se desfășoară asincron, necesită în anumite momente sincronizarea în vederea realizării unei comunicări între ele [PA82].

Luate separat și privite din punct de vedere al structurii lor interioare, procesele concurente sînt de natură secvențială. Ele se compun din date și acțiuni executate în ordine secvențială, cîte una la un moment dat.

3.2. Specificarea proceselor concurente

In definirea limbajelor concurente de nivel înalt s-au adoptat soluții diferite pentru specificarea în mod unitar a datelor și acțiunilor corespunzătoare unui proces. De asemenea foarte legate de acest mecanism de descriere este problema succesiunii în timp a acțiunilor procesului precum și specificarea momentelor în care este necesară sincronizarea diferitelor procese în vederea comunicării. In continuarea acestui subcapitol se vor prezenta doar mecanismele introduse în limbajele CONCURRENT PASCAL și EDISON, a căror implementare este tratată în capitolele următoare.

3.2.1. Specificarea proceselor prin mecanismul COBEGIN. Limbajul de programare EDISON

Definit cu mult înaintea limbajului EDISON, mecanismul COBEGIN pentru specificarea proceselor concurente reprezintă un instrument de descriere mai general și nelegat de un anumit limbaj de programare [DI68]. Forma sintactică în care el a fost adoptat în limbajul EDISON este următoarea [BH81b]:

COBEGIN c_1 DO li_1 ALSO c_2 DO li_2 ALSO . . . ALSO c_n DO li_n END

unde: c_1, c_2, . . . c_n sînt constante;

li_1, li_2, . . . li_n reprezintă liste de instrucțiuni.

Listele de instrucțiuni reprezintă procesele ce se vor executa în paralel. Ele se creează în momentul executării instrucțiunii COBEGIN și se desființează după ce s-au încheiat toate, moment în care se termină și instrucțiunea COBEGIN.

Semnificația constantelor care se asociază fiecărui proces nu este precizată la definirea limbajului. Ea poate să difere de la o implementare la alta (necesarul de memorie pentru procesul respectiv, numărul procesorului în cazul rulării programului pe un sistem multiprocesor, prioritatea procesului, etc.).

Descrierea proceselor paralele în această manieră arată clar și distinct structura și conținutul fiecărui proces. Numărul de procese lansate printr-o instrucțiune COBEGIN este însă fix și nu poate fi schimbat decît prin executarea altei instrucțiuni COBEGIN, după momentul în care precedenta s-a încheiat.

Un program EDISON are forma unei proceduri. El se lansează în execuție prin activarea corpului de instrucțiuni al acestei proceduri. Programul conține la rîndul lui mai multe module și alte proceduri. Structura și utilizarea modulelor EDISON au fost prezentate în Cap. 1.

In limbajul EDISON nu sînt definite nici un fel de operații de intrare/ieșire. Considerînd că acestea sînt dependente într-o măsură foarte mare de suportul hardware, ele sînt lăsate la latitudinea implementării. Prin varianta implementată pe calculatoarele PDP-11 [BH82], P. Brinch Hansen oferă un model simplu în acest sens. Tot în [BH82] se prezintă un sistem de programare complet pentru limbajul

EDISON, destinat unor calculatoare personale. Acest sistem cuprinde: compilator, sistem de operare, asamblor, editor de texte și o serie de programe utilitare, toate scrise în EDISON și executabile utilizînd un executiv comun, care asigură portabilitatea simplă a întregului sistem. De altfel, definirea unui limbaj simplu dar suficient de puternic pentru a permite scrierea unui mediu de programare complet a fost ideea principală pe care s-a bazat proiectul EDISON.

Sistemul de programare conceput de P. Brinch Hansen a stat și la baza implementărilor limbajului EDISON pe micro și minicalculatoarele de producție românească: FELIX M18, M118, M216, INDEPENDENT [RA84]. Părți din acest sistem au fost realizate în dierite variante și pe diverse calculatoare și în activitatea de cercetare științifică sau de diplomă cu studenții secției de specialitate, la I.P.T.V. Timișoara [EC82, CE84a]. Din aceste cercetări au rezultat și propuneri de completare a limbajului EDISON în vederea implementării lui pe sisteme multimicroprocesor sau distribuite [EC85b, EC86b].

3.2.2. Specificarea proceselor ca date abstracte. Limbajul de programare CONCURRENT PASCAL

Specificarea proceselor concurente ca date abstracte este modalitatea adoptată în majoritatea limbajelor de programare concurentă. In funcție de limbaj există o mare diversitate de forme sintactice și semnificații semantice asociate datelor abstracte care modelează procesele.

In CONCURRENT PASCAL, procesele paralele ale programului se declară ca variabile de tipul abstract **process**. Un tip **process** descrie într-o formă unitară și modularizată structurile de date și acțiunile proprii procesului (fig. 3.1.), după modelul sintactic discutat în §1.3.1. pentru clasă. Tipurile **process**, reprezentînd descrierea proceselor și variabilele de aceste tipuri, reprezentînd procesele propriu-zise se declară în blocul exterior, al programului principal, denumit *proces inițial*. Executarea unui program CONCURRENT PASCAL începe prin lansarea corpului de instrucțiuni al procesului inițial. Din acest corp de instrucțiuni, în mod explicit, utilizînd instrucțiuni **init** se lansează celelalte procese ale programului. Un proces odată lansat, nu poate fi desființat pînă la terminarea programului. El se va executa în paralel cu celelalte procese ale programului (inclusiv procesul inițial). In acest fel numărul de procese este fixat la activarea programului și nu poate fi schimbat pe parcurs [CE85b].

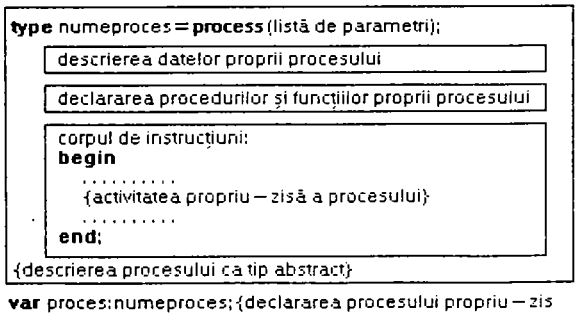


Fig. 3.1. Structura unui proces în limbajul CONCURRENT PASCAL

CONCURRENT PASCAL a fost primul limbaj din familia limbajelor concurente. El a constituit ulterior punctul de plecare pentru o serie întreagă de cercetări teoretice dovedind în același timp și practic faptul că un limbaj de nivel înalt poate constitui suportul corespunzător pentru realizarea unor sisteme de operare și a altor programe concurente.

La definirea limbajului s-au urmărit următoarele obiective:

- realizarea unui limbaj care să permită scrierea completă și modulară a unor sisteme de operare pentru minicalculatoare;
- posibilitatea de a verifica la compilare toate restricțiile impuse de interacțiunea între procese;
- independența programelor față de particularitățile mașinii de calcul;

In acest fel limbajul impune un stil de programare și o serie întreagă de restricții care asigură o bună fiabilitate a programelor [CI80a]. Prin opoziție însă, restricțiile limitează libertatea de descriere care rămîne la dispoziția programatorului. Datorită limitărilor pe care le prezintă limbajul CONCURRENT PASCAL în forma sa inițială, ulterior s-au făcut mai multe propuneri de extindere a limbajului cu noi facilități [SB78, SB79, CE83a, CE84b].

In CONCURRENT PASCAL toate operațiile de intrare/ieșire se realizează în mod unitar, prin intermediul procedurii standard **io**. Procedura efectuează operațiile la nivel fizic, fără a implica însă tratarea de către programator a registrelor fizice, a întreruperilor, etc. Procesul care execută o operație de intrare/ieșire trece în așteptare pe parcursul efectuării operației, permițînd celorlalte procese să ocupe procesorul. Această manieră prezintă avantajul tratării operațiilor periferice sub forma unor instrucțiuni obișnuite (apel de procedură), plasabilă oriunde în cadrul programului, toată gestiunea operației revenind nucleului sistemului care conține driverele de acces pentru toate perifericele prevăzute. De aici rezultă și o serie de dezavantaje:

- dimensiunile relativ mari ale nucleului;
- rigiditatea sistemului - programarea unui periferic neprevăzut la implementarea limbajului se poate face doar prin completarea nucleului cu driver-ul corespunzător;
- nu se oferă mijloace de interacțiune directă cu suportul fizic ceea ce limitează posibilitățile de utilizare a limbajului în aplicații de conducere a proceselor.

In varianta limbajului implementată de P. Brinch Hansen pe calculatorul PDP-11/45 [BH75a] s-au prevăzut și proceduri standard pentru programarea în timp real. Cu unele modificări, aceste proceduri au fost preluate și în alte implementări (vezi cap. 4.).

Aplicațiile realizate în CONCURRENT PASCAL [BH76c, BH77b, BH77c, JS81, KA81, CI81a] au demonstrat practic posibilitatea de a utiliza eficient acest limbaj pentru scrierea de sisteme de operare și alte programe concurente. Este motivul pentru care limbajul a și fost implementat în întreaga lume, pe diverse sisteme de calcul, atît mono cît și multimicroprocesoare [BH75b, KR82, MN81, SE78, SI77]. In perioada 1978-1985, el a fost implementat, în mai multe versiuni, la I.P."Traian Vuia" din Timișoara, și pe calculatoarele românești din gama FELIX C [CI79a, CI82, CE84c]. Concomitent s-au realizat adaptări ale implementării lui P. Brinch Hansen pe PDP-11/45 și pe calculatoarele din familiile INDEPENDENT, CORAL și FELIX M.

3.3. Modalități de interacțiune a proceselor

Pe parcursul rulării unui program concurent, procesele sale paralele pot să interacționeze, din următoarele cauze [BH73b]:

- *utilizarea în comun* a unor resurse cum ar fi: procesoare, echipamente periferice, memorie, zone tampon, variabile, etc.;
- *cooperarea proceselor* prin transmiterea de date de la unul la celălalt.

În principiu, s-au identificat două forme esențial diferite de interacțiune între procese [AS83]:

a) *comunicarea* - reprezentînd transmiterea de informații între procese;

b) *sincronizarea* - prin care se impun restricții asupra evoluției în timp a unui proces.

Comunicarea între procese se realizează fie prin intermediul unor variabile comune la care au acces mai multe procese, fie prin transmitere de mesaje de la un proces

la altul.

Există, de asemenea, mai multe forme de sincronizare care se pot încadra în unul din următoarele două cazuri:

a) *excluderea mutuală*: reprezintă evitarea utilizării simultane, de către mai multe procese, a unei resurse comune; o astfel de resursă, la care este interzis accesul concomitent al mai multor procese, se numește *resursă critică*.

b) *sincronizarea pe condiție*: constă în întreruperea unui proces pînă cînd o condiție devine adevărată.

Mijloacele tradiționale de programare nu sînt suficiente pentru rezolvarea directă și simplă a problemelor legate de sincronizarea proceselor. Din acest motiv, în teoria sistemelor de operare s-au introdus, sub forma *primitivelor de sincronizare*, instrumentele necesare în mecanismul sincronizării proceselor.

Între cele două forme de interacțiune există o strînsă interdependență. Astfel:

- comunicarea între două procese implică automat și sincronizarea lor pentru ca emisia - recepția de informații să se desfășoare corect;
- sincronizarea între două procese presupune că evoluția în timp a unui proces, depinde de celălalt; pentru a sesiza această situație se impune o comunicare pentru schimb de informații.

In consecință, cele două forme diferite de comunicare între procese implică și mecanisme de sincronizare diferite care vor fi prezentate pe scurt în continuare, exact în această manieră.

3.3.1. Sincronizarea bazată pe comunicarea prin variabile comune

Totalitatea acțiunilor prin care un proces manipulează o resursă critică reprezintă o *secțiune critică* [SH74]. In consecință, excluderea mutuală se poate redefini ca fiind acea formă de sincronizare care permite ca numai un proces să se afle, la un moment dat, în interiorul unei secțiuni critice relativă la aceeași resursă critică. Soluționarea corectă a problemei excluderii mutuale trebuie să îndeplinească următoarele condiții [SH74, AN78]:

a) resursa critică să poată fi utilizată de un singur proces la un moment dat;

b) dacă două sau mai multe procese doresc să intre simultan într-o secțiune critică, alegerea unuia dintre ele să se facă într-un interval finit de timp;

c) un proces care nu se găsește într-o secțiune critică nu poate opri un alt proces de a intra într-o secțiune critică;

d) respectarea condițiilor a) - c) trebuie asigurată fără a ține cont de viteza relativă a proceselor.

Pentru a rezolva acest tip de sincronizare s-au conceput o multitudine de primitive, dintre care cele mai cunoscute sînt: semafoarele [DI68], regiunile critice [BH72, HO72] și monitoarele [DI71, HO74]. Ultimele două vor fi tratate în §3.4.1. și respectiv în §3.4.2. în legătură cu adoptarea lor ca modalități de sincronizare și comunicare în limbajele EDISON și respectiv CONCURRENT PASCAL.

3.3.2. Sincronizarea bazată pe comunicarea prin transmitere de mesaje

Caracteristica esențială a primitivelor bazate pe comunicarea prin transmitere de mesaje este realizarea printr-o operație unică atît a sincronizării cît și a comunicării între procese [AN83, PS85]. Suportul comunicării nu-l mai reprezintă variabile comune ce pot fi referite și modificate din mai multe procese ci mesajele (semnale și/sau date) care se transmit între procesul emitor și cel receptor. Operațiile tipice prin care se realizează emisia și recepția se numesc **send** și respectiv **receive**. In diferite sisteme și limbaje de programare există o mare varietate de forme ale acestor operații. Diferențele principale se referă la:

- felul în care se specifică mesajul emis (recepționat);
- gradul de sincronizare între cele două procese;

- specificarea destinației respectiv a sursei mesajului.

Din punctul de vedere al gradului de sincronizare transmisia poate să fie:

a) *sincronă* - dacă procesele se sincronizează pentru a executa simultan operațiile **send** și **receive**;

b) *asincronă* - cînd mesajul emis se depune într-o zonă tampon de dimensiune nelimitată, nefiind necesar ca procesul emițător să aștepte;

c) *intermediară* - analog cu cazul precedent dar zona tampon are dimensiune limitată.

Există, de asemenea, mai multe variante de specificare a destinației și/sau sursei:

a) *numire directă* - specificarea explicită a procesului partener;

b) *numire globală* - comunicarea între procese fără restricții, prin intermediul *porturilor* [BA71];

c) *relația client/servant* - un proces (servant) oferă servicii care pot fi solicitate de către alte procese (clienți) [JU84].

Pentru sincronizarea și comunicarea prin transmitere de mesaje s-au definit mai multe mecanisme complexe de interacțiune a proceselor dintre care cele mai cunoscute sînt:

i) *apelul de procedură la distanță* (remote procedure call) - constă în gruparea într-o operație unică, de forma unui apel de procedură, a solicitării unui serviciu de către client, a așteptării și recepționării răspunsului [BN84, EC84b];

ii) *mecanismul de rendez-vous* - reprezintă mecanismul de interacțiune a proceselor introdus în limbajul ADA (instrucțiunea **accept**); prezintă diferențe față de apelul de procedură la distanță în special în ceea ce privește precizarea relației client/servant [AN83, JU84].

3.4. Facilități de comunicare și sincronizare a proceselor în limbajele de programare EDISON și CONCURRENT PASCAL

Principiul de comunicare a proceselor în limbajele de programare EDISON și CONCURRENT PASCAL se bazează pe variabile comune. În consecință, limbajele pot fi implementate fie pe sisteme monoprocesor, fie pe sisteme multiprocesor cu memorie comună [CI81a, EC83]. În continuare se prezintă instrumentele de sincronizare și comunicare specifice fiecăruia din cele două limbaje.

3.4.1. Regiuni critice condiționale

Regiunile critice au fost definite de C. A. R. Hoare [HO72] și P. Brinch Hansen [BH72], ca o primă soluție de rezolvare, printr-un instrument complet și unitar, a excluderii mutuale. Propunerea făcută permite atît evidențierea explicită, în textul programului, a variabilelor care reprezintă resurse critice cît și accesul controlat la astfel de resurse printr-o instrucțiune specială.

O declarație de forma:

var v:**shared** t

arată faptul că variabila v de tip t este o resursă comună, partajată între mai multe procese.

Accesul la o astfel de variabilă se poate realiza doar în interiorul unei așa numite *regiuni critice*, indicată astfel:

region v **do** listă_de_instrucțiuni **end**

Excluderea mutuală este asigurată prin faptul că, la un moment dat, un singur proces poate executa instrucțiuni dintr-o regiune critică referitoare la o aceeași variabilă partajată v.

Executarea unei instrucțiuni **region** de către un proces se desfășoară conform schemei din fig. 3.2.

Referitor la ordinea de extragere a proceselor care așteaptă în șirul de așteptare se precizează doar că ea trebuie să asigure activarea fiecărui proces după un interval

de timp finit.

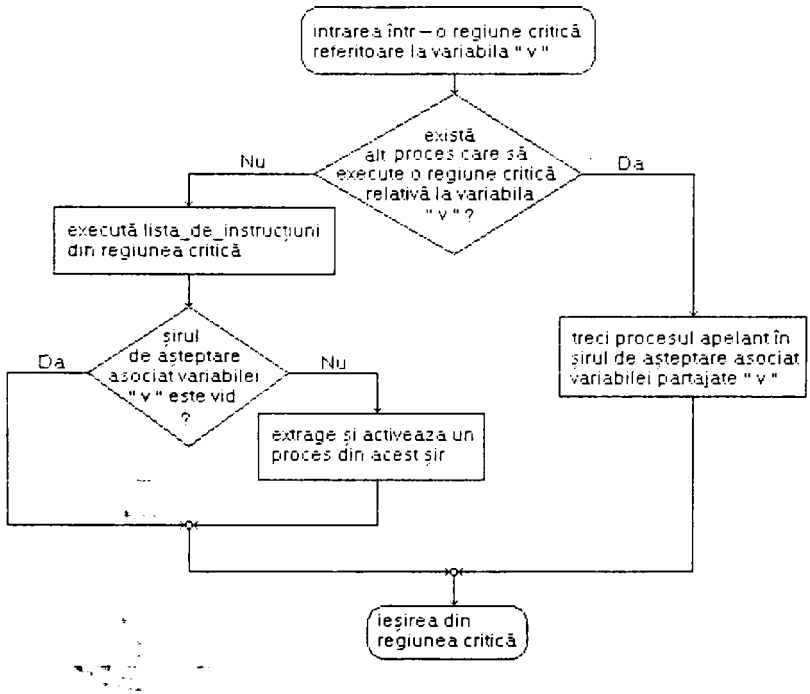


Fig. 3.2. Executarea unei regiuni critice

Pentru a rezolva și sincronizarea pe condiție, regiunile critice s-au extins la regiuni critice condiționale [HO72, BH73a], reprezentate astfel:

region v **when** b **do** listă_de_instrucțiuni **end**

b reprezintă o expresie logică oarecare, în care pot apare atît variabila partajată v sau componente ale ei cît și variabile proprii procesului. In plus față de condiționările anterioare, expresia b trebuie să aibă valoarea "adevărat" pentru ca procesul să poată executa lista_de_instrucțiuni. In caz contrar este pus în așteptare pînă la schimbarea valorii expresiei. Procesele se exclud mutual atît în timpul evaluării expresiei b cît și pe parcursul executării listei de instrucțiuni. Evaluarea expresiei face deci parte din regiunea critică.

Regiunile critice condiționale sînt simple și ușor de utilizat. Dezavantajul lor principal constă în evaluarea repetată a condiției de sincronizare care poate duce la un consum mare de timp de calcul, situație deranjantă mai ales pe un sistem monoprocesor. Consumul de timp poate fi redus dacă reevaluarea condițiilor pentru procesele care așteaptă la o anumită resursă se face doar după ce un alt proces a parcurs cu succes o regiune critică referitoare la resursa respectivă (existînd deci premiza ca, prin modificarea valorii variabilei partajate, să se fi schimbat vreuna din condițiile de sincronizare).

In limbajul de programare EDISON, P. Brinch Hansen a introdus o variantă modificată a regiunilor critice condiționale, de forma [BH81b]:

when b_1 **do** li_1 **else** b_2 **do** li_2 **else** ... **else** b_n **do** li_n **end**

unde:

b_1, b_2, ... , b_n sînt expresii logice;

li_1, li_2, ... , li_n sînt liste de instrucțiuni.

Se observă că nu se specifică variabila partajată la care se referă regiunea critică. Cauza este aceea că, în EDISON, s-a adoptat următoarea soluție simplă: la u

moment dat se exclud reciproc toate regiunile critice. Simplificarea este benefică pentru implementarea limbajului dar se resimte în mod negativ atît la scrierea programelor cît și la rularea lor.

Instrucțiunea **when** se execută în două faze, astfel [BH81a]:

- *faza de sincronizare*: procesul este întîrziat pînă cînd nici un alt proces nu execută faza critică a unei instrucțiuni **when**;
- *faza critică*: se evaluează pe rînd expresiile logice (condițiile) b_1, b_2, ... ; dacă se găsește o condiție adevărată, se execută lista de instrucțiuni corespunzătoare, după care se încheie instrucțiunea **when**; dacă toate condițiile sînt false, se revine la faza de sincronizare.

Regiunile critice condiționale, în forma în care au fost ele introduse în limbajul EDISON, oferă o mare libertate programatorului. În schimb, neexistînd restricții, nu există nici garanția utilizării corecte a variabilelor partajate, decît în măsura în care referirea lor se realizează doar în interiorul unor instrucțiuni **when**, ceea ce nu se poate verifica nici la compilare, nici în execuție (în EDISON variabilele partajate nu se declară în mod distinct și deci nu pot fi identificate între celelalte variabile).

3.4.2. Sincronizarea și comunicarea prin monitoare

Regiunile critice condiționale impun o anumită disciplină în modul de specificare al sincronizării proceselor, dar operațiile de sincronizare sînt dispersate pe tot parcursul programului, în toate punctele în care se operează asupra resurselor critice.

Odată cu progresele realizate în ceea ce privește modularizarea programelor și introducerea noțiunii de dată abstractă (vezi §1.3.) a apărut și ideea descrierii complete a unei resurse critice printr-o construcție unitară și încapsulată numită *monitor* [BH73b, HO74, DI71].

Monitorul este un modul asociat unei resurse. El conține atît structurile de date reprezentînd resursa critică, cît și operațiile prin care se prelucrează acestea, sub formă de proceduri. Monitorul concentrează deci totalitatea secțiunilor critice referitoare la o anumită resursă partajată. Unica modalitate de acces din exterior la resursa critică reprezentată prin variabilele monitorului este apelul procedurilor acestuia.

Excluderea mutuală este asigurată datorită faptului că, la un moment dat, poate fi executată o singură procedură a monitorului, de către un singur proces. Dacă un proces apelează o procedură a unui monitor și în momentul respectiv nu se execută nici o altă procedură a acestuia (monitorul este liber), procesul rămîne activ și execută procedura. Dacă în momentul apelului monitorul are o procedură în execuție (el este ocupat), procesul apelant este pus în așteptare, într-un șir de așteptare asociat monitorului. Cererile proceselor în așteptare sînt satisfăcute pe rînd, în ordinea sosirii lor, pe măsură ce monitorul este eliberat.

Mecanismul de mai sus asigură în mod automat accesul exclusiv al unui proces la resursa partajată prin monitor, pe durata executării unei proceduri a acestuia.

In limbajul de programare CONCURRENT PASCAL monitoarele se definesc ca date abstracte de tip **monitor** după un procedeu sintactic similar proceselor și claselor. Variabilele monitoare sînt elementele prin intermediul cărora interacționează procesele paralele [BH75a].

Un tip **monitor** descrie o structură de date accesibilă mai multor procese precum și operațiile permise asupra acestor date (fig. 3.3.). Accesul din exterior la structurile descrise în monitor este permis numai prin intermediul procedurilor și funcțiilor externe (marcate în declarație prin cuvîntul cheie **entry**) ale acestuia. Apelul unei astfel de proceduri sau funcții se realizează conform mecanismului de excludere mutuală descris anterior.

Obiectele existente pe parcursul rulării unui program CONCURRENT PASCAL sînt variabilele declarate de tip **process**, **monitor**, **class**. Ele reprezintă componentele

programului concurent [CI80b, BH77a]. Procesele și monitoarele se declară ca variabile în procesul inițial iar clasele se declară în interiorul altei componente (proces, monitor sau altă clasă), de care aparțin exclusiv.

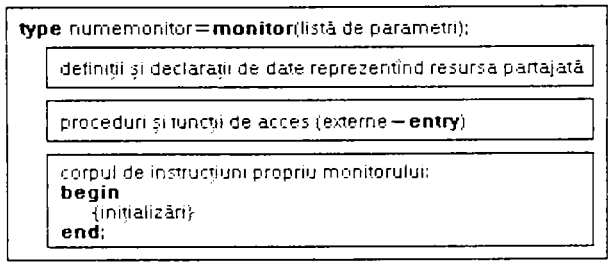


Fig. 3.3. Structura generală a unui monitor în CONCURRENT PASCAL

Toate componentele programului se lansează prin instrucțiuni **init**. In cazul monitoarelor și claselor, activarea încetează după executarea corpului de instrucțiuni propriu și se pot reactiva doar prin apelarea procedurilor și funcțiilor lor externe. Procesele odată activate, reprezintă activități continue care se execută paralel și concurent pînă la terminarea programului. . . .

Interacțiunea dintre procese se realizează prin intermediul monitoarelor. La inițializare, unui proces i se transmit ca argumente, monitoarele la care are acces. In mod analog, și monitoarele pot, la rîndul lor, să aibe acces la alte monitoare. Această explicitare a drepturilor de acces permite verificarea la compilare a unei discipline a interacțiunii proceselor, în vederea evitării unor accese monitorizate la resurse partajate.

CONCURRENT PASCAL impune monitorul ca *unică* modalitate de comunicare între procese [EC87]. Se interzice declararea unor variabile comune în afara monitoarelor, ceea ce asigură eliminarea oricăror erori legate de nerespectarea excluderii mutuale în accesul la aceste variabile. O astfel de restricție poate fi uneori excesivă și poate avea ca efect reducerea gradului de paralelism în evoluția proceselor (implicit scăderea performanțelor programului). Astfel, în unele situații accesul concomitent din mai multe procese la o variabilă ar putea fi permis dacă procesele fac doar referire la valoarea variabilei, fără să o modifice sau în cazul unei structuri la care excluderea mutuală se impune doar la nivelul fiecărei componente în parte.

Explicitarea drepturilor de acces în programul sursă, deși pare să-i confere acestuia o structură rigidă oferă, așa cum s-a arătat, posibilități suplimentare compilatorului de a decide asupra corectitudinii programului. Astfel se detectează încă din compilare una din erorile obișnuite care poate duce la blocări ale proceselor: apelurile ciclice între monitoare [EC88a, HA77, LT77]. Problema găsirii unor mijloace pentru specificarea în mod dinamic a drepturilor de acces între procese, care să confere totodată și un grad corespunzător de securitate, a constituit o preocupare încă de la apariția limbajelor concurente. Majoritatea soluțiilor actuale se bazează pe mecanismul capabilităților [KS78, KS83, MA79].

In concluzie, monitoarele impun o modularizare și o disciplină a programului care îi conferă acestuia claritate și flexibilitate. Se evidențiază în mod explicit resursele critice și totalitatea operațiilor (secțiunile critice) permise asupra lor. Rezolvînd automat excluderea mutuală, se evită implicit orice posibilitate de acces concomitent din mai multe procese la aceeași resursă. Poate fi considerat, din majoritatea punctelor de vedere, un mecanism mai evoluat, mai util și mai performant decît regiunile critice condiționale.

3.4.3. Sincronizarea pe condiție în limbajul CONCURRENT PASCAL

Realizarea sincronizării pe condiție necesită introducerea unui mecanism suplimentar, care să permită punerea în așteptare a unui proces pînă la îndeplinirea

unei anumite condiții. Dacă excluderea mutuală este rezolvată cu monitoare în mod automat, transparent la nivelul programului, sincronizarea pe condiție trebuie realizată prin mijloace minuite în mod explicit de către programator. O primă propunere în acest sens a fost făcută de C. A. R. Hoare în [HO74]. Ulterior, în literatura de specialitate s-au formulat o serie întreagă de mecanisme pentru sincronizarea pe condiție în monitoare. Unul din cele mai cunoscute este acela propus de P. Brinch Hansen [BH75c] și aplicat în limbajul CONCURRENT PASCAL [BH75a, CE85a].

Pentru rezolvarea sincronizării pe condiție în limbajul de programare CONCURRENT PASCAL s-a introdus tipul standard **queue** și, sub formă de proceduri standard, operațiile **delay** și **continue** pentru manipularea variabilelor de acest tip.

O variabilă de tip **queue** reprezintă o locație în care poate fi pus în așteptare un singur proces. Ea se inițializează în mod automat pe starea vidă (valoarea standard **nil**). Pentru a construi șiruri de așteptare pornind de la astfel de locații, se declară structuri (de obicei tablouri) cu elemente de tip **queue** executînd operațiile **delay** și **continue** asupra unei locații sau alteia, în funcție de politica de gestionare a șirului adoptată de programator.

Variabile de tip **queue** sau structuri cu componente de acest tip se pot declara doar în monitoare. În plus, parametrii conținînd locații de tip **queue** nu pot fi parametri de ieșire ai unor rutine externe (nu pot fi transmiși în exterior). Prin urmare, toate acțiunile de punere în așteptare și relansarea proceselor au loc exclusiv în monitoare. Ele se realizează prin intermediul procedurilor standard **delay** și **continue**. Ambele se apelează cu un argument de tip **queue** și execută următoarele operații:

delay (x) - Procesul activ (care a apelat rutina monitor) se pune în așteptare în variabila x. Monitorul se eliberează și devine astfel accesibil pentru alt proces.

continue (x) - Dacă în locația x așteaptă un proces, acesta se reactivează din punctul în care a executat **delay**. În continuare, acest proces va avea acces exclusiv la monitor. Totodată variabila x se poziționează pe starea vidă. Procesul care a apelat **continue** revine din rutina externă și pierde accesul la monitor. Dacă locația x a fost vidă se execută doar eliberarea monitorului și revenirea din rutina externă.

Mecanismul de sincronizare prin variabile de tip **queue** și operațiile **delay** și **continue** se mai numește și *sincronizare programată*, prin opoziție cu cea realizată prin intermediul monitoarelor, care este automată [CE85a]. Erori în utilizarea acestor operații pot duce la blocarea unor procese. O astfel de blocare poate fi cauzată, de exemplu, prin neexecutarea operației **continue** pe o locație în care așteaptă un proces introdus cu **delay**, cu toate că s-a produs condiția așteptată de procesul respectiv. Aceasta este o eroare tipică de programare pentru acest mecanism.

Pentru a ușura gestiunea șirurilor de așteptare, în limbajul de programare CONCURRENT PASCAL s-a introdus și o funcție standard de tip boolean, **empty**, care primește valoarea **true** dacă în locația de așteptare (de tip **queue**) transmisă ca parametru nu așteaptă nici un proces (locația este vidă).

Prin mecanismul de monitor, completat cu facilitatea de sincronizare programată, se pot rezolva practic toate cazurile uzuale de sincronizare și cooperare între procese.

Capitolul 4

UTILIZAREA LIMBAJELOR CONCURENTE DE NIVEL INALT PENTRU PROGRAMAREA IN TIMP REAL

Intre programarea concurentă și cea secvențială există o strînsă interdependență. Astfel:

- Elementele de concurență acționează atît la nivelul programării extinse (organizarea programului în module, între care un loc aparte îl ocupă procesele), cît și la nivelul programării restrînse (primitive de sincronizare și comunicare specifice limbajului respectiv). In ambele etape se uzează din plin de conceptele și instrumentele de bază ale programării (secvențiale): module, tipuri abstracte, subprograme și respectiv date și acțiuni elementare sau structurate. De altfel, așa cum s-a subliniat în cap. 3., unele din limbajele concurente de nivel inalt definite în ultimul timp sînt mai degrabă limbaje de programare universale cu facilități pentru programarea concurentă.
- Procesele, rupte din contextul concurent al programului, sînt procese secvențiale.
- In același timp, într-un sistem de calcul, pot coexista și chiar se pot executa atît programe concurente cît și secvențiale. Cazul cel mai frecvent este cel al sistemului de operare sub controlul căruia se execută aplicații de natură secvențială, privite adesea ca procese.

In cap. 3. nu s-a pus problema felului în care factorul timp influențează sau poate influența execuția unui program. Totuși, fiecare acțiune a programului se execută într-un interval de timp finit, mai mare sau mai mic, în funcție de viteza procesorului. Programele secvențiale sînt astfel descrise încît durata acțiunilor nu influențează corectitudinea rezultatelor calculate. Timpul scurs pentru executarea unui program interesează doar global, prin aspectul său comercial și prin impactul pe care îl au asupra bilanțului de timp al programatorului.

Cu totul altfel se pune problema în programarea concurentă. Ideea organizării programului în procese concurente, care urmează să fie executate în paralel, are la bază rațiuni de utilizare intensivă și eficientă a procesorului (sau procesoarelor) în timp. S-a arătat (cap. 3.) că și în cazul programelor concurente este preferabil ca ele să fie concepute și descrise astfel încît vitezele absolute și relative ale procesoarelor să nu influențeze corectitudinea rezultatelor obținute. Această independență de timp a programelor are importante avantaje practice:

- programele rezultate sînt generale și portabile;
- corectitudinea programului poate fi dedusă doar din textul său, static, ceea ce permite automatizarea acestui proces.

Practica utilizării calculatorului a demonstrat însă că există și aplicații în care viteza de execuție a programelor (sau a unor secvențe de program) influențează corectitudinea efectelor și rezultatelor sale. Astfel de programe aparțin domeniului *programării în timp real*.

4.1. Domeniul programării în timp real

Se pot semnala două situații tipice pentru modul de lucru în timp real și anume:

1) Efectuarea unor operații la momente bine precizate de timp sau la intervale regulate (perioade de eșantionare, intervale de control).

2) Efectuarea unor operații la momente de timp aleatoare, stabilite prin semnale care sosesc din exteriorul sistemului. Pentru a asigura corectitudinea rezultatelor se poate proceda într-unul din următoarele două moduri:

- întîrzierea (dacă este posibil) unui semnal pînă la terminarea operațiilor declanșate de cel precedent;
- garantarea unui interval minim între două semnale consecutive, timp în care

trebuie terminate obligatoriu operațiile de prelucrare.

Răspîndirea largă a utilizării tehnicii de calcul în toate domeniile activității economico-sociale a condus la diversificarea fără precedent a modului de lucru în timp real, la aplicații de genul: controlul și coordonarea unor procese tehnologice sau de altă natură, achiziția și prelucrarea datelor, conducerea roboților industriali, a instalațiilor de automatizare sau de telecomunicații, etc. Diversitatea de aplicații presupune o mare varietate de echipamente periferice legate la sistemele de calcul care lucrează în timp real: aparate de măsură, traductori, senzori, sisteme de reglare, motoare, terminale, etc. [CR84, DA83].

In concluzie, un program în timp real este în serviciul direct al unei anumite aplicații (sau a unui număr restrîns de aplicații de regulă înrudite) de la care primește și/sau spre care emite semnale. Caracteristica esențială a programului este promptitudinea cu care rezolvă serviciile solicitate de aplicație, definită prin *timpul de răspuns*: intervalul de timp scurs între solicitarea unui serviciu și rezolvarea lui. Acest interval depinde în primul rînd de natura aplicației, dar este evident influențat și de tipul echipamentelor periferice, numărul și viteza procesoarelor, numărul și complexitatea operațiilor de prelucrare.

4.2. Relația "programare concurentă" - "programare în timp real"

Un program secvențial este independent de timpul în care se efectuează acțiunile sale. Aceasta nu înseamnă că un program secvențial n-ar putea avea acces la timpul real furnizat de exemplu de orologiul calculatorului. Se pot prevedea chiar acțiuni tipice programării în timp real cum ar fi: suspendarea executării programului pentru o cuantă de timp, contorizarea unor cuante de timp, etc. Toate aceste acțiuni se succed însă secvențial, fiecare acțiune avînd la un moment dat un singur predecesor și un singur succesor, univoc determinați prin text i programului. Din acest motiv, accesul la timpul real, eventuale calcule asupra timpului, nu pot să afecteze corectitudinea rezultatelor programului.

In ceea ce privește programele în timp real propriu-zise, se pot extrage următoarele caracteristici generale [CE86a]:

1) Activitatea unui program în timp real este corelată cu comportarea în timp a sistemului fizic coordonat.

2) Componentele active ale programului sînt procese paralele și concurente care se sincronizează și comunică între ele.

3) In anumite aplicații, în care timpul de răspuns (mic) este esențial, viteza de lucru necesară se poate obține prin creșterea gradului de paralelism al proceselor: creșterea numărului de procese și/sau procesoare.

4) Numărul de procese în execuție este fix pentru o anumită configurație de procesoare și echipamente periferice.

5) Activitatea unui program în timp real este continuă (procesele se derulează ciclic) pe întreaga durată a funcționării sistemului.

Afirmațiile enumerate permit următoarea concluzie generală: *programarea în timp real implică programarea concurentă.*

Analizînd cu atenție mai multe tipuri de aplicații concurente [CE87b] se poate face următoarea observație: *într-o serie de programe concurente nu se poate neglija comportarea în timp real a proceselor paralele*, chiar dacă acest lucru nu este evident la prima vedere.

N. Wirth împarte programarea în trei mari categorii [WI77d]: *secvențială, concurentă* și *în timp real*. In acest paragraf, s-a insistat mai ales pe două aspecte:

- relativa independență a programării secvențiale de cea în timp real;
- relațiile deosebit de complexe existente între programarea în timp real și cea concurentă care nu pot fi despărțite nici din punct de vedere teoretic și nici în activitatea de proiectare - programare.

In acest context se poate aprecia că *programarea în timp real este o parte a*

programării concurente și anume acea parte în care timpii de declanșare și executare a acțiunilor programului devin semnificativi și influențează activitate sincronizată a proceselor paralele, repercutîndu-se eventual chiar asupra corectitudinii rezultatelor.

4.3. Limbaje pentru programarea în timp real

Ca și programarea concurentă în ansamblu, programarea în timp real s-a realizat inițial exclusiv în limbaje de asamblare. Au existat, și parțial mai sînt valabile trei motive esențiale care au influențat acest lucru:

- Timpul de răspuns trebuie determinat cu precizie în toate cazurile iar, pentru o serie de aplicații, el trebuie redus sub anumite limite, imposibil sau foarte greu de atins în limbaj de nivel înalt.
- Programatorul are nevoie de acces direct la echipamentul fizic (echipamente periferice, ceas de timp real, porturi de intrare/ieșire, etc.), iar limbajele de programare de nivel înalt n-au prevăzut inițial acest obiectiv.
- Programul în timp real este concurent, încorporînd deci propria sa politică de gestiune, sincronizare și întrerupere a proceselor paralele ce le conține. In acest context, întreruperile suplimentare provocate de sistemul de operare, pot deveni deranjante prin caracterul lor aleator și prin durata lor nedeterminată. In limbajele de nivel înalt tradiționale, spre deosebire de cele de asamblare, lipsesc mijloace care să permită eliminarea în anumite situații a politicilor implementate în sistemul de operare ca interfață între programul în timp real și calculator. Dealtfel în multe cazuri sistemul de calcul este dedicat exclusiv aplicației respective, programul în timp real interacționînd direct cu echipamentul fizic (prin urmare nu mai este necesară prezența unui astfel de sistem de operare).

Perfecționarea ulterioară, atît a tehnicii de calcul cît și a limbajelor de programare, a modificat acest punct de vedere. Cîteva realizări mai importante cu implicații asupra acestui fapt sînt:

1) Creșterea vitezei de lucru a procesoarelor actuale. Astfel s-au putut obține din ce în ce mai ușor timpi de răspuns mici, ceruți în unele aplicații de timp real.

2) Apariția și dezvoltarea sistemelor multiprocesor, cu implicații asupra creșterii vitezei prin mărirea gradului de paralelism fizic asigurat prelucrărilor.

3) Orientarea unor procesoare (a codului lor mașină) pe structurile (de date și acțiuni) proprii anumitor limbaje de nivel înalt [ZE81].

4) Definirea și implementarea unor limbaje de nivel înalt din ce în ce mai performante, incluzînd și facilități specifice modului de lucru în timp real (de exemplu RTL/2 [IC74]).

5) Apariția și perfecționarea limbajelor de nivel înalt pentru programarea concurentă.

6) Dezvoltarea continuă și aplicarea tehnicilor de optimizare a codului obiect generat de compilatoare.

In acest context nu pot să surprindă aprecierile făcute în literatura de specialitate actuală [WI77d], conform cărora un limbaj pentru programarea în timp real trebuie să fie în primul rînd un limbaj de programare perfecționat și performant, ceea ce înseamnă implicit, de nivel înalt. Principalele criterii de evaluare ale limbajelor de timp real sînt criteriile generale pentru aprecierea limbajelor de programare [DB86, YO81]. La nivelul definirii limbajelor, aceste criterii se materializează în cerințe privind în special intoducerea unor facilități care să permită realizarea unor programe în timp real optime. Se prezintă în continuare cele mai importante dintre aceste cerințe:

1) Pentru a asigura securitatea și lizibilitatea cerute, programele în timp real trebuie să fie în primul rînd structurate și modularizate. Limbajele de programare adecvate trebuie deci să conțină structuri corespunzătoare atît pentru descrierea acțiunilor cît mai ales pentru descrierea datelor, inclusiv tipuri abstracte (cap.1.). Alegerea tipurilor de date cele mai potrivite, asociate cu specificarea clară a

operațiilor permise asupra obiectelor (tipuri abstracte), rezolvă detectarea simplă a erorilor care pot să apară la manipularea datelor.

2) Programele în timp real sînt uneori mari și complexe. Limbajul trebuie deci să suporte necesitățile programării extinse, ceea ce înseamnă în primul rînd prezența facilităților de modularizare. Acest aspect poate fi de asemenea rezolvat prin introducerea tipurilor abstracte de date.

3) Programele în timp real sînt concurente prin natura lor. În consecință limbajul corespunzător trebuie să fie concurent (cap. 3.).

4) Prezența frecventă în sistemele în timp real a unor echipamente periferice nestandard implică furnizarea prin limbaj a unor facilități pentru programarea directă, la nivel scăzut, a operațiilor de intrare/ieșire, ceea ce este specific și limbajelor concurente în general.

5) Pentru a asigura funcționalitate maximă sistemelor de timp real, limbajul trebuie să permită tratarea în program a eventualelor erori, ceea ce asigură, în majoritatea cazurilor, restabilirea activității după incidente.

6) Limbajul de programare trebuie să asigure interceptarea și tratarea semnalelor de întrerupere emise de echipamentele periferice, inclusiv de ceasul de timp real.

7) Pentru anumite activități programul presupune accesul direct la timpul real. Acest lucru se realizează prin introducerea în limbaj a unor modalități de determinare a timpului, măsurarea unui interval de timp, întîrzierea unui proces pe o durată fixă, etc.

8) Pentru fiecare implementare a unui limbaj de timp real trebuie să se furnizeze, în documentația destinată utilizatorilor, limite de timp de execuție corecte, corespunzătoare instrucțiunilor sursă (eventual secvențelor de instrucțiuni). Acest lucru, deși pare șocant în cazul unui limbaj de programare de nivel înalt, este relativ simplu de determinat la elaborarea compilatorului (faza de generare a codului obiect).

În concluzie, *limbajul pentru programarea în timp real* trebuie să fie un *limbaj concurent* deosebit de performant care include în plus facilități pentru tratarea timpului real.

Prima realizare importantă în domeniul limbajelor de nivel înalt pentru programarea în timp real este CONCURRENT PASCAL care, după cum s-a văzut, are și meritul de a fi deschizător de drum și în programarea concurentă în general. Deși nu fac propriu-zis din limbaj, facilitățile de timp real accesibile în CONCURRENT PASCAL pot reprezenta baza unei rezolvări satisfăcătoare a multor categorii de aplicații [BH75c]. În perioada imediat următoare lui CONCURRENT PASCAL (1976) au apărut alte două limbaje recomandate pentru acest domeniu: RTL/2 și PORTAL.

Definit de J. G. P. Barnes [BA80], RTL/2 a fost inspirat din ALGOL-60 și ALGOL-68, de la care a preluat setul puternic și uniform al structurilor de acțiuni, reprezentînd un limbaj mic, compact și eficient. Este în schimb rudimentar în ceea ce privește abstractizarea datelor și mai ales în programarea concurentă. PORTAL, definit în laboratoarele firmei elvețiene Landis & Gyr și publicat în 1978 [LI78, SL80], reprezintă ca limbaj concurent un pas în plus față de CONCURRENT PASCAL, dar nu furnizează facilități noi pentru gestiunea timpului real [SC78].

Începînd cu definirea în 1977 a limbajului MODULA [WI77a], în domeniul programării în timp real s-au impus într-o măsură tot mai mare limbajele de nivel înalt pentru programarea concurentă. Marea majoritate a acestor limbaje (EDISON, EUCLID, MESA, ADA, etc.) permit cu bune rezultate realizarea de aplicații în acest domeniu.

4.4. Trăsături specifice ale aplicării programării concurente în limbaje de nivel înalt, la domeniul timpului real

Pornind de la natura (concurentă) a aplicațiilor în timp real, s-a stabilit anterior că un limbaj de programare pentru acest gen de aplicații trebuie să fie în primul

rînd un bun limbaj concurent. Trebuie deci încorporate în limbaj elementele de bază care facilitează programarea concurentă și anume:

- o notație unitară pentru descrierea proceselor;
- funcțiile de creare și eventual desființare a proceselor;
- posibilitatea de a inițializa (lansa în execuție) și de a bloca (opri) procesele create;
- modalități eficiente de sincronizare și comunicare între procese;
- punerea în așteptare și relansarea, condiționate de un eveniment exterior, a unor procese;
- intirzierea unor procese pentru intervale de timp date.

Majoritatea acestor elemente au fost tratate în cap. 3. In cele ce urmează se vor face cîteva completări pornind mai ales de la specificul aplicării lor la programarea în timp real.

4.4.1. Intreruperi

Există mai multe cauze pentru care activitatea unui proces, parte componentă a programului concurent corespunzător unei aplicații în timp real poate fi întreruptă:

a) *Intrerupere în vederea sincronizării și eventual comunicării cu alt proces.* Este provocată de o acțiune (**P**, **wait**, **delay**, etc.) executată chiar de procesul care va fi întrerupt (procesul își întrerupe singur activitatea) și are ca efect punerea sa în așteptare pînă la realizarea unui eveniment (îndeplinirea unei condiții, apariția unui semnal, poziționarea unui semafor, etc.). Acest tip de întrerupere s-a prezentat în capitolul 3.

b) *Intrerupere cauzată de un eveniment din exteriorul procesului.* Are ca și cauză un semnal recepționat de la un alt procesor, un echipament periferic sau ceasul de timp real. Efectul este, de regulă, întreruperea unui proces și (re)lansarea altuia care, fie așteaptă producerea acestui eveniment (cazul de la punctul a)) ca să-și poată continua activitatea, fie este specializat în analizarea și tratarea evenimentului produs. Acest tip de întrerupere acționează direct la nivelul procesorului și provoacă în mod indirect întreruperea și punerea în așteptare a unui proces logic. Este deosebit de utilizată de programarea în timp real.

c) *Intrerupere pentru lansarea altui proces care așteaptă să lucreze pe același procesor.* Apare în cazul partajării procesorului între mai multe procese. Se poate realiza efectiv, de exemplu, prin "divizare de timp" efectuată fie în programul concurent, fie în nucleul specializat al executivului care asigură rularea programului.

Intreruperile introduc o dificultate suplimentară în programarea în timp real și anume aceea că limitele de timp de execuție determinate static pentru grupe de instrucțiuni nu mai pot constitui o bază suficientă în analizarea și calcularea timpului de răspuns. In cazul în care întreruperea este vizibilă în textul programului (tipul a) și o parte din întreruperile de tip c)), întirzierea provocată de ea poate fi luată în calcul (dacă se cunoaște limita maximă a așteptării). Intreruperile de tip c), provocate de nucleu, sînt în schimb cu totul imprevizibile, mai ales asupra locului din proces în care pot să intervină și din această cauză sînt dăunătoare aplicațiilor în timp real. Acesta este probabil motivul principal pentru care programarea în timp real a rămas vreme îndelungată în domeniul limbajelor de asamblare unde, toate detaliile fiind la dispoziția programatorului, există aparent o siguranță superioară.

Problema întreruperilor în aplicațiile de timp real, privită în toată generalitatea ei, nu are încă o soluție satisfăcătoare. Pornind de la cazurile concrete care apar în practica programării se pot face însă o serie de simplificări, și anume:

- Operațiile pentru care timpul real este critic sînt cel mai des în legătură cu echipamentele periferice de controlat (aparate de măsură, senzori, etc.); în majoritatea cazurilor ele pot fi concentrate și chiar izolate în structuri de program cu caracter închis (date abstracte).
- Pentru fiecare echipament periferic (eventual pentru un număr de echipamente

identice) se va descrie un singur proces. In acest caz procesul se numește *driver*. Pe sistemele multiprocesor apare în plus și posibilitatea de a asocia în exclusivitate un procesor unui astfel de proces.

- In general sistemele în timp real pot avea un număr mare de echipamente periferice, dar numai cîteva din acestea necesită răspuns rapid.

Concentrarea operațiilor care implică lucrul cu echipamentele periferice în procese sau module specializate se poate realiza simplu în orice limbaj concurent de nivel înalt.

Dificultatea principală apare atunci cînd procesul este partajat între mai multe procese, caz în care informațiile despre durata acțiunilor și cele referitoare la sincronizarea proceselor, rezultate static din textul programului, nu sînt suficiente pentru analiza completă a comportării sistemului în timp real. Este nevoie să se cunoască, în plus, strategia de activare a diferitelor procese individuale care, în majoritatea cazurilor, se "ascunde" în nucleul sistemului. Este de asemenea important dacă anumite procese, eventual în anumite condiții sînt sau nu servite cu prioritate.

4.4.2. Priorități

Strategiile de servire a proceselor concurente bazate pe priorități au fost introduse timpuriu în cadrul sistemelor de operare. In aceste condiții este firesc faptul că ele s-au adoptat de la început și la implementarea limbajelor de nivel înalt pentru programarea concurentă. Spre exemplu, în prima implementare a limbajului CONCURRENT PASCAL, realizată de P. Brinch Hansen pe calculatorul PDP-11 [BH76a], s-au stabilit următoarele priorități care se acordă automat proceselor, în funcție de starea lor momentană, de către nucleul sistemului:

0: cînd procesul execută acțiuni dintr-un monitor sau dintr-o clasă activată direct sau indirect dintr-un monitor;

1: dacă procesul a terminat o operație de intrare/ieșire și nu sînt îndeplinite condițiile de la prioritatea 0;

2: la lansarea procesului, la revenirea dintr-o rutină externă de monitor (dacă nu mai sînt îndeplinite condițiile de la prioritatea 0), la depășirea unei cuante de timp admise consumată în unitatea centrală (se aplică divizarea timpului între procese).

Prioritatea cea mai înaltă este 0. Urmează în ordine 1 și 2. Nucleul ține cont de aceste priorități la acordarea procesorului unui anumit proces. Un proces mai puțin prioritar poate fi întrerupt pentru a se lansa altul mai prioritar.

Tocmai în scopul îmbunătățirii performanțelor aplicațiilor de timp real, la implementarea limbajului CONCURRENT PASCAL pe calculatoarele românești din gama FELIX C [CC80, CE83a, CE85c] s-a prevăzut o modificare importantă, prin introducerea unei priorități și mai înalte, -1. Aceasta se acordă unui proces din momentul apelării procedurii standard **wait** (§4.5.3.) și pînă la reactivarea sa. Astfel, procesele în care se tratează întreruperi de timp, sînt servite cu prioritate.

De altfel tactica de a acorda prioritate maximă rutinelor de întrerupere este deosebit de frecventă în cadrul sistemelor de operare. Se obișnuiește ca, în momentul primirii unui semnal de întrerupere să se lanseze necondiționat pe procesorul respectiv rutina (procesul) specializată în tratarea acelui semnal iar procesul care era activ se întrerupe și se pune în așteptare. Prin această măsură se face un pas înainte în direcția determinării exacte a timpului de răspuns, dar a apărut o problemă nouă: în mod normal, în timp ce se tratează o întrerupere pot să sosească alte întreruperi tratabile cu aceeași prioritate. Soluția adoptată curent pentru rezolvarea acestei situații este de a pune întreruperile în așteptare și de a le trata secvențial în ordinea sosirii. De exemplu, în nucleul MODULA [WI77c], această posibilă întîrziere este prevăzută în secvența corespunzătoare efectuării instrucțiunii **doio**. Avantajul constă în aceea că *timpul maxim de întîrziere* poate fi determinat cu precizie în mod static, din textul programului, ca sumă a timpilor de execuție

corespunzători tuturor celorlalte secvențe de întrerupere, considerate cîte una (și anume cea mai lungă) pentru fiecare proces. *Timpul de întîrziere minim* este cel necesar pentru punerea în așteptare a procesului care se întrerupe; acest timp se poate preciza exact după implementare, fiind vorba de durata unor activități care se derulează în nucleu.

Dacă se are acum în vedere, așa cum s-a spus, că nu toate echipamentele periferice necesită răspuns rapid, sistemul de întrerupere asociat implementării unui limbaj de nivel înalt pentru programarea în timp real poate fi organizat pe nivele. Astfel un nivel de prioritate devine întreruptibil de către un altul mai înalt decît el. Evident va trebui să se asigure în plus ca perifericele pentru care timpul de răspuns este critic, să provoace întreruperi de cel mai înalt nivel. În acest fel timpul de întîrziere maxim, scurs din momentul emiterii unui semnal de întrerupere de prioritatea cea mai mare și pînă cînd se lansează tratarea întreruperii, se reduce considerabil. Pentru determinarea lui nu se iau în considerare toate procesele tip *driver* ci numai cele cărora le sînt adresate întreruperile de pe nivelul cel mai înalt.

În cazul cînd echipamentele pentru care contează timpul de răspuns provoacă întreruperi de nivele de priorități diferite (nu sînt obligatoriu de nivelul cel mai înalt), calculul timpului de întîrziere maxim pentru un nivel oarecare se complică în mod substanțial. Trebuie luate în considerare numai activitățile din secvențele de tratare ale întreruperilor corespunzătoare acelui nivel, ci și cele ale nivelelor mai prioritare. O schiță a unui posibil calcul pentru acest caz este dată de N. Wirth în [WI77d].

Mai rămîne de lămurit o singură problemă în acest paragraf: cum poate decide (influența) programatorul disciplina de acordare a procesorului partajat, proceselor din programul său?

O primă cale sînt mecanismele de punere și scoatere din așteptare a proceselor, proprii fiecărui limbaj de nivel înalt pentru programarea concurentă. Ele au fost tratate pe larg în capitolele precedente.

O a doua modalitate este aceea de a se indica din program, la declararea procesului, care este prioritatea sa. O soluție de acest gen s-a adoptat în limbajul MODULA prin precizarea la declararea proceselor interioare unui **device module** a așa numitului *vector de întrerupere* (locația de memorie asociată nivelului de întrerupere) [WI77c].

4.4.3. Metodologia elaborării programelor în timp real, realizate în limbaje concurente de nivel înalt

După ce s-a parcurs și s-a analizat utilitatea și eficiența programării în timp real realizată în limbaje concurente de nivel înalt, se va încerca în continuare să se stabilească o metodologie de programare adecvată. Ea va rezulta în principal prin sintetizarea și generalizarea ideilor prezentate de N. Wirth ca recomandări privind programarea în timp real în limbajul MODULA [WI77d]. Ideea de bază este aceea de a amîna introducerea, calculul și analizarea restricțiilor de timp pentru o etapă cît mai tîrzie, moment în care programul este deja validat din punct de vedere al funcționării sale logice (prin aceasta se înțelege corectitudinea programului, inclusiv a aspectelor legate de concurență, dar ignorînd factorul timp). În continuare se prezintă etapele metodologiei propuse:

1) În funcție de specificul și conținutul aplicației se stabilesc părțile (modulele) componente ale programului. Se precizează în detaliu activitatea fiecărui modul urmărind ca porțiunile de care depinde timpul de răspuns pentru un anumit echipament periferic să fie concentrate în interiorul aceluiași modul.

2) În funcție de rolul fiecărui modul în cadrul sistemului concurent și de mijloacele specifice oferite de limbajul de programare ales se stabilesc modalitățile de sincronizare și comunicare între modulele (activitățile) cu rol de procese. Pentru aceasta se analizează în detaliu semnalele de sincronizare (întrerupere) furnizate de

echipamentele hardware și se stabilesc semnale suplimentare și condiționările impuse de funcționarea preconizată a sistemului. Din motive de fiabilitate și eficiență se recomandă respectarea următoarelor restricții:

- fiecare semnal emis de către un periferic să fie așteptat doar de un singur proces logic (din program); prin aceasta secvența de tratare a unei întreruperi este univoc determinată la un moment dat;
- un proces driver nu poate să invalideze el însuși condiția asociată cu un semnal pe care l-a emis.

3) Se scrie programul proiectat conform punctelor 1) și 2), în limbajul de programare ales.

4) Se testează și se pune la punct programul urmărind corectitudinea sincronizării și comunicării între procese din punct de vedere logic (în această etapă nu interesează durata și desfășurarea lor în timp). Dacă limbajul permite acest lucru, se va asigura ca procesele care trebuie să răspundă rapid la solicitări exterioare să aibă prioritate mare (eventual maximă).

5) Pe baza timpilor elementari furnizați pentru instrucțiuni sau grupe de instrucțiuni la implementarea limbajului, se determină duratele de execuție ale secvențelor critice din punctul de vedere al timpilor de răspuns. În cazul în care nu s-a putut realiza ca secvențele critice să nu fie întreruptibile, la duratele determinate se vor adăuga timpii de așteptare (întîrziere) maximi, calculați conform considerațiilor prezentate în paragraful precedent. Însumînd duratele secvențelor critice și eventual timpii de așteptare (sau numai regia de sistem, consumată în nucleu) la trecerea de la o secvență la alta, se determină timpii de răspuns.

6) Se verifică încadrarea timpilor de răspuns în cei prescriși la formularea aplicației. În cazul în care acest deziderat nu este atins se încearcă reducerea la minim și optimizarea secvențelor critice apoi se revine la punctul 5). Ajustările secvențelor critice se fac de așa manieră încît să nu se afecteze mecanismul de sincronizare și comunicare între procese care a fost proiectat și validat.

7) În final se rulează programul pentru cît mai multe situații concrete și se verifică "pe viu" corectitudinea timpilor de răspuns.

Pentru ca metodologia propusă să fie eficace trebuie evident avută în vedere o condiționare inițială: limbajul de programare împreună cu sistemul de calcul alese trebuie să permită în principiu (din punctul de vedere al vitezei de execuție) rezolvarea aplicației în termenii prescriși în enunț. Altfel punctele 5) și 6) se pot transforma ușor într-un "ciclu infinit".

4.5. Alte facilități utile programării în timp real

În majoritatea limbajelor de programare concurentă au fost încorporate o serie de facilități specifice programării în timp real sau, mai general, specifice programării de sistem dar utile și în aplicațiile de timp real. O parte din aceste facilități, încorporate în limbajele CONCURRENT PASCAL și EDISON au fost prezentate în cap. 3. În continuare se va face o scurtă trecere a lor în revistă, cu caracter de sistematizare, punctînd în special rolul pe care îl are programarea în timp real.

4.5.1. Intrări/ieșiri la nivel scăzut

Una din caracteristicile importante ale sistemelor în timp real este fără discuție marea diversitate a echipamentelor hardware (periferice) cu care trebuie să comunice. Multe dintre acestea sînt specializate (nestandard), imposibil de precizat la implementarea limbajului pe un tip de sisteme de calcul și cu atît mai puțin la definirea lui. Această situație aparent dificilă și în mare măsură specifică programării concurente (de sistem) în general, a fost ocolită în etapa definirii limbajelor concurente. Marea majoritate a acestor limbaje nu înglobează instrucțiuni pentru programarea operațiilor de intrare/ieșire. De regulă însă, la implementare, s-a

introdus un suport general privind tratarea intrărilor/ieşirilor la nivel scăzut sub forma unor rutine (standard) specializate. Exemple tipice în acest sens sînt: CONCURRENT PASCAL (procedura **io**) şi EDISON (procedurile **place**, **obtain** şi funcţia **sense**). Pe baza mecanismelor de nivel scăzut prevăzute se pot realiza module specializate care să furnizeze sub formă de intrări (apeluri de proceduri şi funcţii) operaţiile periferice la nivel înalt. Aceasta se poate face la implementarea limbajului (CONCURRENT EUCLID, MESA, MODULA-2, ADA) şi/sau numai la utilizarea lui pornind de la necesităţile concrete ale aplicaţiilor sau tipurilor de aplicaţii.

Rutinele pentru tratarea la nivel scăzut a operaţiilor de intrare/ieşire au o formă suficient de generală. Ele asigură funcţii de bază cum ar fi: transferul unui bloc de date (fără prelucrări - conversii) sau a unui semnal, între periferic şi memoria internă, accesul la registrele hardware (poziţionare, testare) pentru indicarea operaţiei periferice şi respectiv pentru verificarea terminării ei. Această manieră de lucru prezintă avantajul că, uneori, chiar ceasul de timp real poate fi tratat ca un periferic obişnuit, ceea ce permite ca unele limbaje (CONCURRENT EUCLID, EDISON) să fie considerate potrivite pentru programarea în timp real fără a avea prevăzut nimic specific în acest sens. Folosirea la nivelul programului utilizator a rutinelor de nivel scăzut afectează însă în mare măsură portabilitatea programului. Este inevitabil ca funcţionarea şi uneori chiar forma de apelare (numărul, tipul şi semnificaţia parametrilor) a acestor rutine să depindă de maşina de calcul sau măcar de nucleul care asigură implementarea limbajului.

Dezavantajul evidenţiat mai sus este în mare măsură estompat în limbajele concurente mai noi în care s-a prevăzut realizarea la implementare a unor module specializate ce permit tratarea intrărilor/ieşirilor la nivel înalt (independent de maşină). Să mai remarcăm şi faptul că, sub această formă, se pot defini şi realiza şi operaţii de intrare/ieşire de nivel foarte înalt: gestiunea şi prelucrarea fişierelor şi a bazelor de date, prelucrări grafice, etc.

Tratarea operaţiilor periferice la nivel scăzut face parte dintr-o categorie mai generală de facilităţi preluată de limbajele concurente din cele de asamblare, cunoscută sub numele de "acces la nivelul fizic". Această categorie mai poate include:

- Interceptarea şi tratarea întreruperilor externe separat de tratarea intrărilor/ieşirilor. Se remarcă în acest sens modul elegant de interceptare a întreruperilor prevăzut în limbajul MODULA-2 prin procedura **iotransfer** din modulul **system** [WI82].
- Accesul direct la spaţiul de adresare al memoriei. Acest mod de lucru permite alocarea (relocarea) unei variabile la (de la) o adresă dată sau, mai general, gestionarea dinamică a memoriei în program, în conformitate cu cerinţele aplicaţiei şi includerea în unele programe de sistem a funcţiei specifice de gestiune a memoriei interne.

4.5.2. Detectarea şi tratarea excepţiilor

Excepţiile sînt situaţii speciale care apar la executarea unui program. Ele se împart, în principiu, în două categorii, astfel:

1) erorile de execuţie cu care sîntem familiarizaţi din programarea secvenţială: depăşirea domeniului unor mulţimi numerice proprii limbajului sau sistemului de calcul, depăşirea stivei de date, violarea zonei de memorie afectată altui program (proces) etc.;

2) atingerea unor situaţii limită care nu sînt permise de specificaţiile limbajului sau a căror tratare n-a fost prevăzută la nivelul nucleului de implementare.

In programarea concurentă, problema detectării şi tratării uniforme a tuturor excepţiilor capătă valenţe deosebite din următoarele motive:

- este dificil de realizat o delimitare strictă între cele două categorii de excepţii

prezentate mai sus;

- pentru a asigura un înalt grad de fiabilitate sistemului concurent este esențial ca el să nu fie întrerupt chiar dacă apare o eroare; dimpotrivă, într-o astfel de situație, programul trebuie să semnaleze eroare după care va asigura minimul de măsuri pentru continuarea normală a activității (recuperarea după eroare);
- programul propriu-zis poate fi simplificat prin excluderea la nivelul acestuia a unor situații limită.

Problema detectării și tratării excepțiilor nu este nouă în programare. Ea a constituit de timpuriu unul din obiectivele sistemelor de operare. Punctul de vedere nou care va fi tratat în continuare este acela de a include în limbajele de programare concurentă mecanisme de nivel înalt specifice, pentru detectarea și tratarea excepțiilor de către programator. Se remarcă și faptul că, într-un limbaj concurent, semnalarea și tratarea excepțiilor poate fi ușor inclusă în codul general al tratării evenimentelor (alături de întreruperile interne și externe despre care s-a scris în paragrafele anterioare) [BE80].

Mecanismele de nivel înalt pentru tratarea excepțiilor incluse în limbajele concurente (în particular cele de timp real) trebuie să respecte anumite cerințe [LS79, YO81], și anume:

- să fie ușor de înțeles și utilizat;
- codul sursă corespunzător tratării excepțiilor trebuie să fie suficient de sumar și concentrat încît să nu împieteze asupra urmăririi și înțelegerii simple a restului programului;
- mecanismul trebuie să fie astfel gîndit încît să nu necesite timp de execuție suplimentar în cazul unei execuții normale (în care nu se semnalează excepții);
- trebuie să permită tratarea uniformă a excepțiilor indiferent de nivelul la care au fost detectate (nucleu sau program);
- trebuie să includă un minim de facilități pentru refacerea condițiilor necesare continuării programului (recuperare);

Deși mecanismele de nivel înalt pentru interceptarea și tratarea excepțiilor au apărut recent, limbajele de programare și practica programării în general au impus de timpuriu diferite facilități, tehnici și metode în acest sens.

Una din cele mai simple metode de tratare a excepțiilor (în special a erorilor) detectate este aceea în care se utilizează un parametru de procedură pentru a returna coduri de eroare către programul principal apelant. In urma apelului procedurii se poate testa codul de eroare și se pot executa acțiuni corespunzătoare în funcție de valoarea lui. Metoda are avantajul simplității precum și pe acela de a nu necesita noi facilități de limbaj. Este ușor de observat însă că această metodă nu satisface nici pe departe cerințele prezentate mai sus pentru un mecanism de nivel înalt acceptabil, din următoarele motive:

- includerea tratării erorilor printre apelurile de proceduri afectează claritatea programului și îl complică; structura de control normală este fragmentată prin introducerea secvențelor de tratare;
- dacă numărul de apeluri pentru care se returnează cod de eroare este mare, tratarea erorii în fiecare caz determină mărirea în mod artificial a codului corespunzător programului și, ceea ce este și mai grav în special în aplicații de timp real, se mărește timpul de execuție chiar și în situațiile în care nu apare eroare.

O oarecare ameliorare a dezavantajelor de mai sus se poate obține acceptînd utilizarea instrucțiunii **goto** pentru ca, în cazul detectării unei erori, să se facă saltul la o etichetă unică a programului, la care este prevăzută tratarea tuturor excepțiilor.

In ceea ce privește în special tratarea excepțiilor și continuarea programului în cazul în care se produce un astfel de eveniment, în programarea concurentă se manifestă o serie de particularități importante și anume:

a) Intr-un anumit proces concurent excepția (eroarea) poate apare asincron, din interacțiunea cu alt proces; într-un astfel de caz este dificil de evaluat efectul

excepției și, prin urmare, de asigurat condițiile de recuperare și continuare a procesului; în concluzie, se impune limitarea propagării unei excepții prin interacțiunea dintre procese.

b) Se pune problema ca, în cazul detectării unei erori un anumit proces să poată fi oprit (desființat, abortat) iar restul programului să-și poată continua activitatea. Pentru aceasta se poate include în limbaj o instrucțiune de terminare anormală, activată direct din procesul în care s-a manifestat eroarea. Soluția prezintă dezavantajul că, în procesul respectiv, nu se pot prevedea și acțiuni de minimizare sau chiar anulare a efectelor pe care desființarea le-ar putea avea asupra altor procese. O soluție mai bună este declanșarea terminării unui proces, dintr-un alt proces, cu încercarea de a prevedea și trata toate operațiile rezultate din terminare (transfer de echipamente periferice, închiderea unor fișiere, etc.).

c) Există cazuri în care situația excepțională detectată într-un proces sau desființarea acestuia trebuie semnalată altor procese "partenere". Modul concret prin care se va realiza această operație într-un program concurent depinde desigur direct de metoda de comunicare specifică limbajului respectiv (monitoare, regiuni critice, apel de procedură la distanță, rendez-vous, etc.).

In continuare se va analiza mecanismul de detectare și tratare a excepțiilor prevăzut pentru limbajul ADA, care este tipic pentru nivelul actual de dezvoltare al ingineriei programării.

După cum rezultă din lucrările care tratează această problemă în detaliu [BA82, GE83, LP80], blocurile, corpurile de subprogram, de pachet și cele de task, pot include o parte distinctă de tratare a excepțiilor, marcată prin cuvîntul cheie **exception**. Excepțiile sînt în principiu de două feluri: predefinite în limbaj (**constraint-error**, **numeric-error**, **tasking-error**, etc.) sau definite în program, prin liste de identificatori urmate de același cuvînt cheie **exception**.

La apariția unei excepții predefinite care este tratată în unitatea respectivă (bloc, corp de subprogram, pachet sau task) controlul se transferă automat secvenței de tratare, după care unitatea respectivă este abandonată. Pentru excepțiile declarate în program, controlul de secvență de tratare corespunzătoare se transferă prin executarea instrucțiunii **raise**, de forma:

raise nume_de_excepție;

Secvențele de tratare a excepțiilor sînt marcate prin cuvîntul cheie **when**, avînd sintaxa următoare:

when listă_de_nume_de_excepții
        listă_de_instr;

sau

when others listă_de_instr;

Prin această a doua variantă se specifică secvența de tratare (listă_de_instr) pentru toate excepțiile (predefinite sau definite în program) a căror tratare n-a fost prevăzută în unitatea respectivă.

Dacă într-o unitate de program (bloc, corp de subprogram, pachet sau task) nu există rutină de tratare pentru o anumită excepție, aceasta se propagă în sus pe lanțul de apeluri din execuție. Propagarea se va opri la nivelul la care este prevăzută tratarea sau, dacă o asemenea tratare nu este prevăzută pe lanțul de apel, în programul principal. In acest al doilea caz programul se va încheia anormal. In cadrul task-urilor, excepțiile au o tratare deosebită în cazurile cînd ele apar în timpul comunicării (în cursul executării unei instrucțiuni **accept**). Intr-un astfel de caz și dacă, în plus, excepția nu este tratată local în instrucțiunea accept, atunci ea se propagă atît în task-ul apelat cît și în cel apelant.

In legătură cu instrucțiunea **abort**, sînt situații de interacțiune a proceselor cînd executarea ei poate avea ca urmare semnalarea, în anumite task-uri, a excepției **tasking_error**, și anume:

- într-un task care apelează un task terminat;
- în task-ul apelant, atunci cînd cel apelat se termină (prin **abort**) într-un moment

ulterior apelului dar înainte de încheierea comunicării (task-ul apelant se află în șirul de așteptare sau instrucțiunea **accept** este în curs de deservire); situația inversă nu este semnalată: dacă task-ul apelant se termină anormal, acest fapt nu se transmite (printr-o excepție) în cel apelat.

Există și alte situații anormale ce pot să apară la sincronizarea și comunicarea între task-uri și care se manifestă prin emiterea excepției **tasking_error**. Posibilitatea de a trata această excepție oferă programatorului un control suplimentar în astfel de cazuri. Apar unele particularități și în ceea ce privește propagarea acestei excepții. Ele sînt tratate pe larg în [AN83, BA82, RB81].

Tratarea excepțiilor în limbajele de programare de nivel înalt și în special în cele concurente este deosebit de complexă. Prezentarea succintă realizată în acest paragraf n-a putut, evident, să o epuizeze. Pornind de la cerințele enumerate la început, s-a încercat și se încearcă în continuare rezolvarea satisfăcătoare a acestei probleme. Pe acest drum, mecanismul introdus în limbajul ADA reprezintă un pas important.

4.5.3. Accesul direct la timpul real

Pentru a se programa întîrzierea unei activități pe o durată dată, declanșarea periodică sau la o anumită oră a unor acțiuni precum și alte activități în timp, limbajul de programare al acestor aplicații trebuie să includă facilități pentru accesul direct la dispozitivul numit *ceas de timp real*. Intr-o variantă minimă este suficient ca acest dispozitiv să emită periodic, cu o perioadă constantă bine stabilită, semnale de întrerupere sesizabile fie în nucleul sistemului fie direct în programul concurent. Pe baza interceptării și contorizării acestor semnale se pot programa, direct în limbajul de nivel înalt, "procese ceas" care să gestioneze timpul conform necesităților aplicației.

In §4.5.1. s-a amintit o primă posibilitate de legătură cu ceasul de timp real, asimilîndu-l cu un echipament periferic programabil la nivel scăzut. In afara modalității simple de mai sus, limbajele de programare concurentă au mai inclus operații de acces la timpul real sub formă de instrucțiuni dar mai ales ca proceduri și funcții predefinite (standard).

Astfel, CONCURRENT PASCAL conține rutinele **wait** și **realtime** cu următoarea semnificație:

procedura **wait**(fără parametri) - întîrzie procesul apelant timp de o secundă;

procedura **realtime**(fără parametri) - furnizează intervalul de timp (real) scurs de la începutul executării programului.

Avînd în vedere efectul procedurii **wait**, o secvență de forma

cycle wait; s:=s+1; {alte acțiuni} **end**;

Poate contoriza cu o bună aproximație timpul real în secunde. Eroarea pentru o secundă este dată de durata acțiunilor prevăzute între două apeluri consecutive ale procedurii **wait**. Ideea de mai sus poate fi dezvoltată în sensul gestionării orei exacte. Gestionarea timpului în această formă permite declanșarea periodică sau la o anumită oră a unor activități.

Pentru a mări precizia de contorizare a timpului, implementarea limbajului CONCURRENT PASCAL pe calculatoarele românești din gama FELIX C [CC80, CE83a], efectul procedurii **wait** a fost modificat astfel: în nucleul atașat oricărui program CONCURRENT PASCAL se include automat un "ceas" intern care se declanșează la lansarea în execuție a programului și pulsează (generează întreruperi) cu perioada de aproximativ o secundă; activitatea ceasului este automată și independentă de cea a programului; în aceste condiții procedura **wait** are ca efect întîrzierea procesului care a executat apelul pînă la proxima întrerupere de ceas. Avînd în vedere și faptul (subliniat în §4.4.2.) că un proces care execută **wait** primește la relansare prioritatea cea mai înaltă (-1), secvențe de program ca cele de mai sus contorizează corect timpul real cu condiția ca acțiunile prevăzute între două

apeluri ale procedurii **wait** să se execute în mai puțin de o secundă. De remarcat că dacă această condiție este îndeplinită, ceea ce este destul de simplu la viteza calculatoarelor actuale, durata acțiunilor respective nu influențează cu nimic precizia contorizării timpului.

Din motive legate de utilitatea ei în aplicații practice, în aceeași implementare a limbajului CONCURRENT PASCAL pe calculatoarele FELIX C, funcția standard **realtime** a fost înlocuită prin procedura **realtime**(x). Unicul parametru, x, este un tablou de 8 caractere. La apelul procedurii se furnizează în parametrul x ora exactă exprimată în format zecimal extern cu cîte două cifre în ordine pentru ore, minute, secunde, sutimi de secundă.

Modelul de acces la timpul real din CONCURRENT PASCAL a fost preluat și în alte limbaje concurente. Astfel procedura **delay**(x) din limbajul PATH PASCAL [CK80] are același efect ca **wait** cu deosebirea că întîrzierea provocată procesului apelant este variabilă (durata ei se transmite prin parametrul x). Același limbaj include și o funcție, **time**, similară lui **realtime** din CONCURRENT PASCAL. Tot pe linia analogiilor din această categorie se poate considera și instrucțiunea **delay** din limbajul ADA [KJ82].

O modalitate principial diferită de abordare a timpului real în programarea concurentă este combinarea în construcții de limbaj (instrucțiuni) unitare a elementelor de sincronizare a proceselor și comunicare cu cele vizînd factorul timp. In această direcție se înscriu două variante ale mecanismului **select** din limbajul ADA [MA80] și anume:

- așteptarea selectivă, pentru care s-au prevăzut alternative **delay** (dacă în intervalul de timp corespunzător nu se poate selecta nici o alternativă **accept**, atunci se selectează alternativa **delay** deschisă cu durata de așteptare minimă);
- apelul temporizat (face o tentativă de stabilire a unui rendez-vous într-un anumit interval de timp specificat într-o instrucțiune **delay**, parte componentă a instrucțiunii **select**).

4.6. Concluzii

Aplicarea limbajelor de nivel înalt pentru programarea concurentă la programarea în timp real are numeroase avantaje. Diversitatea de structuri de date și acțiuni, necesitatea de a le grupa și izola pe acelea care sînt dependente de timp, multitudinea de condiționări în vederea sincronizării activităților și alte aspecte specifice, fac ca eficiența utilizării limbajelor de nivel înalt să fie chiar mai mare în programarea în timp real decît în cea obișnuită.

In acest capitol s-a încercat și definirea unei discipline de programare adecvată aplicațiilor în timp real. O astfel de disciplină, corelată cu sporirea neîncetată a performanțelor sistemelor de calcul și a limbajelor de programare actuale va conduce, fără îndoială, la răspîndirea utilizării limbajelor de nivel înalt în domeniul timpului real.

Capitolul 5

Modele și metode pentru controlul vizibilității identificatorilor la faza de compilare

O perioadă îndelungată, problema controlului vizibilității identificatorilor a fost exclusiv legată de declararea și utilizarea identificatorilor în limbajele în care programele au structură de blocuri suprapuse (familiile ALGOL - PASCAL). Într-o serie întreagă de limbaje mai noi (CONCURRENT PASCAL [BH75c], EDISON [BH81b], SMALLTALK [GR83], CLU [LI79], EUCLID [LA77], GYPSY [GC78], MODULA-2 [WI82], etc.), controlul vizibilității identificatorilor s-a complicat substanțial, în special datorită încorporării conceptelor de date abstracte (exportul și importul identificatorilor) și a problematicii specifice programării concurente (sincronizarea și comunicarea între procese).

Ca și majoritatea celorlalte concepte din domeniul limbajelor de programare și compilatoarelor, mecanismele pentru controlul vizibilității au apărut și s-au dezvoltat la început empiric fiind materializate prin algoritmii concreți care au fost proiectați și implementați în diferite compilatoare. Ulterior, toate aspectele vizate de controlul vizibilității s-au închegat și s-au individualizat într-o problemă de sine stătătoare în cadrul teoriei și practicii compilatoare: *analiza de domeniu* (scope analysis).

În continuare, în cadrul acestui capitol, se vor urmări patru direcții principale:

1) Prezentarea și compararea mecanismelor pentru controlul vizibilității identificatorilor din cîteva limbaje de programare: PASCAL, CONCURRENT PASCAL, EDISON.

2) Definirea unor instrumente pentru modelarea formală a principalelor aspecte legate de controlul vizibilității identificatorilor.

3) Evaluarea mecanismelor pentru controlul vizibilității din diferite limbaje de programare, pe baza modelării formale.

4) Prezentarea unor algoritmi pentru analiza de domeniu implementați în compilatoarele PASCAL/FELIX C, PASCAL CONCURENT/FELIX C, EDISON.

5.1. Noțiunea de domeniu în limbajele de programare PASCAL, CONCURRENT PASCAL și EDISON

Domeniul unei entități căreia i s-a asociat un identificator este porțiunea de program în care identificatorul poate fi utilizat (este vizibil) pentru a indica entitatea respectivă.

În limbajele de programare din familia ALGOL, din care fac parte atît PASCAL și CONCURRENT PASCAL cît și EDISON, noțiunea de domeniu este strîns legată de aceea de *bloc* [CE85a]. Acesta este format dintr-o parte de declarații prin care, unor entități, li se asociază identificatori și care, la rîndul ei, poate conține alte blocuri și dintr-un corp propriu de instrucțiuni. În principiu, un *program* scris în oricare din limbajele de mai sus reprezintă deci un bloc alcătuit din alte blocuri interioare și eventual suprapuse: *procedurile* și *funcțiile*. Există, desigur, și anumite particularități care depind de regulile specifice fiecărui limbaj și anume [JW74, BH75c, BH82]:

- în PASCAL și EDISON, spre deosebire de CONCURRENT PASCAL, în cadrul unei proceduri sau funcții se pot declara alte proceduri sau funcții;
- tipurile sistem din CONCURRENT PASCAL, *procese*, *monitoare* și *clase*, declarabile, de regulă, doar în programul principal (procesul inițial) reprezintă blocuri în care se pot declara proceduri și funcții;
- în limbajul EDISON apar, de asemenea, două aspecte specifice: a) programul principal este organizat și declarat ca o procedură; b) în orice procedură se pot declara, ca blocuri interioare, *module*.

In această structură de blocuri suprapuse, complet interioare unul altuia şi toate conţinute în blocul programului principal, domeniul unei entităţi desemnată printr-un identificator (pe scurt domeniul unui identificator) nu coincide în mod obligatoriu cu întreg programul ci se stabileşte în funcţie de felul şi tipul entităţii respective precum şi de locul definirii şi declarării ei.

In principiu, domeniul se întinde din locul în care se declară entitatea, pînă la sfîrşitul blocului în care apare declaraţia, incluzînd şi blocurile interioare. In blocul în care se declară, identificatorul se consideră *local* iar în cele interioare este *global*.

Un identificator poate fi declarat într-un bloc doar cu un singur sens (nu este posibilă declararea multiplă a aceluiaşi identificator). In schimb, un identificator definit sau declarat într-un bloc, poate fi redeclarat cu un nou sens într-un bloc interior, caz în care domeniul identificatorului în sensul iniţial nu cuprinde acest bloc interior. Se spune că identificatorul din blocul exterior este *suspendat* în cel interior.

Pe lîngă aceste reguli generale, mecanismele pentru controlul vizibilităţii identificatorilor din fiecare dintre limbajele analizate încorporează reguli de domeniu specifice, şi anume:

- pentru a permite înlănţuiri de structuri alocate dinamic, în limbajul PASCAL se pot defini pointeri la structuri încă nedefinite; este totuşi obligatoriu ca structurile în cauză să fie definite pînă la sfîrşitul blocului respectiv [CR87];
- apelarea recursivă a procedurilor şi funcţiilor, acceptată în PASCAL şi EDISON [CE85a, BH82], necesită următoarele două reguli suplimentare: a) o procedură sau o funcţie se poate apela pe ea însăşi (recursivitate directă); b) o procedură sau o funcţie se poate apela şi dacă ea a fost declarată incomplet - **forward** (indicîndu-i numele şi lista de parametri formali); în acest caz blocul ei va apare ulterior în program, după alte proceduri sau funcţii pe care le poate apela şi din care, eventual, este apelată; astfel de apeluri circulare reprezintă recursivitatea indirectă;
- în tipurile procese, monitoare şi clase dintr-un program CONCURRENT PASCAL se pot declara proceduri, funcţii şi variabile externe (entităţi exportate); identificatorii acestor entităţi nu pot fi referiţi (nu sînt vizibili) în cadrul blocurilor în care au fost declaraţi, ci numai în acele blocuri exterioare care intră în domeniul unei variabile sau parametru formal de tipul respectiv [CE85a];
- în limbajul EDISON [BH81b], domeniul unui identificator declarat într-un modul poate fi extins (tot prin export), pînă la sfîrşitul blocului imediat exterior modulului; un identificator nu poate fi exportat într-un bloc exterior în care este deja declarat sau în care este exportat şi din alt modul.

Parte integrantă a compilatorului, analizorul de domeniu trebuie să asigure verificarea tuturor regulilor de mai sus, atît cele generale cît şi cele specifice limbajului respectiv. In esenţă, se poate spune că rolul analizei de domeniu este acela de a verifica dacă entităţile din program sînt utilizate corect pe domeniul lor. Se semnalează erori dacă un identificator este referit în afara domeniului său precum şi în cazul unor declarări multiple ale aceluiaşi identificator în acelaşi bloc (ambiguitate).

Literatura de specialitate [BH82, HR77, WI75] conţine diverşi algoritmi care răspund în mod satisfăcător problemelor enumerate mai sus. O trăsătură comună a acestor algoritmi este noţiunea de nivel, ca mărime asociată blocurilor programului analizat, după următoarele reguli:

- blocul programului principal este de nivel 0 sau 1;
- dacă un bloc este de nivel n, cele imediat interioare lui sînt de nivel n+1.

In §5.4. şi §5.5. se prezintă algoritmii pentru analiza de domeniu implementaţi în compilatoarele PASCAL/FELIX C, PASCAL CONCURENT/FELIX C şi respectiv EDISON.

5.2. Modelarea formală a vizibilităţii identificatorilor

Din punctul de vedere al vizibilităţii, între entităţile unui program se stabilesc două tipuri de relaţii:

1) Relaţii de tipul *solicită referire*: între o entitate şi mulţimea celorlalte entităţi

(identificatori) care pot să fie utilizate (referite) la definirea sau declararea sa. Relațiile pot să fie explicite (import) sau implicite. Exemplu: relația dintre un subprogram și entitățile la care are acces (subprogramul însuși - se poate apela recursiv, entitățile declarate în subprogram - locale și unele entități declarate în blocurile înconjurătoare - globale).

2) Relații de tipul *permite referire*: Între o entitate și mulțimea celorlalte entități (identificatori) care pot utiliza în declararea lor entitatea respectivă. Și aceste relații pot să fie explicite (export) sau implicite. Exemplu: relația dintre un subprogram și entitățile programului în care el poate fi apelat (subprogramul însuși, unitatea de program imediat înconjurătoare, unele unități de program de pe același nivel cu subprogramul și unitățile de program interioare subprogramului. În principiu, din punctul de vedere al vizibilității, o entitate, A, a unui program poate avea acces la o altă entitate, B, doar dacă între ele există ambele tipuri de relații: A *solicită referire* la B și B *permite referire* pentru A. Mecanismul pentru controlul vizibilității identificatorilor dintr-un limbaj de programare poate fi definit ca mijlocul de specificare a celor două tipuri de relații. Diferențele între *solicitare* și *permisiune* sînt, evident, specifice limbajului de programare. De exemplu, în limbajul PASCAL "solicitarea" și "permisiunea" referirii sînt simetrice ca formă: entitățile *solicitate, permit* întotdeauna referirea și invers. În limbajele de programare mai noi, în care import - exportul sau numai unul dintre ele se specifică în mod explicit (cap. 1, §1.3.), apar unele diferențe între cele două tipuri de relații. În această categorie se încadrează limbajele de programare concurentă: CONCURRENT PASCAL, EDISON, MODULA-2, ADA, etc.

Un model grafic potrivit pentru reprezentarea vizibilității entităților dintr-un program este *graful vizibilității* [WO87]. Într-un astfel de graf, nodurile sînt entitățile programului iar arcele dintre noduri marchează cele două tipuri de relații posibile.

Definiția 5.1.: Un graf de vizibilitate G_V este un triplet de forma:

$G_V=\langle N,S,P\rangle$, unde:

- N este mulțimea nodurilor, reprezentînd entitățile (identificatorii) programului;
- S este perechea mulțimilor ordonate de noduri (A,B) aflate în relația: A *solicită referire* la B;
- P este mulțimea perechilor ordonate de noduri (A,B) aflate în relația: A *permite referire* din partea lui B.

Dat fiind faptul că relațiile de tipurile "solicită" sau "permite" sînt recursive, grafurile de vizibilitate pot conține bucle sau cicluri. Absența unui arc între două noduri, indică faptul că între nodurile respective nu există nici un fel de relație de vizibilitate.

Graful de vizibilitate poate fi separat în două subgrafuri deschise, corespunzătoare celor două tipuri de relații.

Definiția 5.2.: Fie un graf de vizibilitate $G_V=\langle N,S,P\rangle$, notațiile avînd semnificațiile din def. 5.1.

Perechea $G_V(S)=\langle N,S\rangle$ reprezintă subgraful corespunzător relației "solicită referire" (*graful de solicitare*).

Perechea $G_V(P)=\langle N,P\rangle$ reprezintă subgraful corespunzător relației "permite referire" (*graful de permisiune*).

Graful de vizibilitate poate să fie construit automat de compilator pornind fie de la textul sursă fie de la arborele sintactic, aplicînd regulile mecanismului pentru controlul vizibilității specific limbajului respectiv.

Considerînd:

$\mathcal{P}$ - mulțimea programelor scrise într-un limbaj de programare, reprezentate într-o formă oarecare (text sursă, arbore sintactic);

$\mathcal{G}$ - mulțimea grafurilor de vizibilitate;

se poate defini o funcție care realizează corespondența între cele două mulțimi, astfel:

Definiția 5.3.: Corespondența între reprezentarea $\mathcal{P}$ a unui program și graful de vizibilitate corespunzător $G_V \in \mathcal{G}$

$$f_V: \mathcal{P} \to \mathcal{G}$$

se numește *funcție de vizibilitate*.

Cu notațiile introduse anterior, un mecanism pentru controlul vizibilității identificatorilor poate fi descris prin funcția de vizibilitate corespunzătoare, f_V. O astfel de funcție are două componente, corespunzătoare relațiilor de tip "solicită" și "permite", astfel:

- *funcția de solicitare*, s_V, transpune repezentarea programului într-un graf de solicitare;
- *funcția de permisiune*, p_V, transpune reprezentarea programului într-un graf de permisiune.

Considerînd $P \in \mathcal{P}$, reprezentarea unui program, rezultă:

$$f_V(P) = s_V(P) \cup p_V(P),$$

semnificînd obținerea grafului de vizibilitate prin reuniunea subgrafurilor corespunzătoare celor două tipuri de relații.

Intreaga descriere a mecanismului de control al vizibilității pentru un limbaj de programare complet poate fi laborioasă și greu de realizat. De mare importanță practică este faptul că, cele două funcții, s_V și p_V se pot obține, la rîndul lor, prin reuniunea funcțiilor de solicitare și permisiune corespunzătoare diferitelor categorii de entități din limbajul de programare (constante, tipuri, variabile, subprograme, date abstracte, etc.), astfel:

$$s_V(P) = s_{1V}(P) \cup s_{2V}(P) \cup \ldots \cup s_{nV}(P)$$
$$p_V(P) = p_{1V}(P) \cup p_{2V}(P) \cup \ldots \cup p_{nV}(P)$$

Pentru extragerea funcției de vizibilitate, punctul de plecare îl constituie, desigur, domeniul ei de definiție, adică alegerea unei reprezentări adecvate a programelor. Pentru diferitele categorii de entități din limbajul de programare respectiv reprezentările pot să fie diferite. De exemplu, în cazul limbajului PASCAL, pentru a defini funcția parțială referitoare la vizibilitatea programelor, o reprezentare potrivită a programelor este așa numitul graf de suprapunere (nesting graph) [WO87]. Acest graf reflectă sugestiv relațiile dintre subprograme fără a fi însă la fel de oportun în cazul tipurilor sau variabilelor. In limbajele de programare care suportă descrieri de obiecte sub formă de date abstracte și operații specifice între astfel de obiecte (sincronizare, comunicare, schimb de mesaje), dificultatea principală în descrierea formală a mecanismului de vizibilitate constă în reprezentarea corectă și completă a relațiilor (complexe) dintre datele abstracte ale programului, relații guvernate, deseori, de indicații de import - export explicite.

Modelarea problemelor de vizibilitate în maniera de mai sus este potrivită mecanismelor de control cu caracter static, mecanisme încorporate în marea majoritate a limbajelor de programare actuale, inclusiv în cele foarte noi. Această tendință este probabil justificată de faptul că mecanismele statice prezintă un grad de securitate mai mare, inclusiv în timpul execuției programelor. Un exemplu elocvent în acest sens este evoluția dialectelor limbajului LISP: în timp ce variante mai vechi se bazează pe reguli dinamice de domeniu, o variantă nouă COMMON LISP [ST84], încorporează un mecanism static pentru controlul vizibilității.

5.3. Evaluarea mecanismelor pentru controlul vizibilității identificatorilor, din diferite limbaje de programare

Există, evident, mai multe criterii pe baza cărora pot să fie evaluate mecanismele pentru controlul vizibilității identificatorilor. In acest paragraf se va realiza o evaluare strict pe baza noțiunilor și definițiilor introduse în §5.2. In esență, evaluarea constă în analizarea felului în care mecanismele rezolvă cele două tipuri de relații: "solicită referire" și "permite referire".

O cerință minimă, pe care trebuie să o includă orice mecanism este aceea de a

permite, mai greu sau mai ușor orice relație de solicitare sau permisiune pe care o dorește programatorul. Limbajul PASCAL și limbajele de programare ulterioare lui satisfac această cerință spre deosebire de limbajele mai vechi (de ex. majoritatea variantelor limbajului FORTRAN) care, nesuportînd recursivitatea, nu permit nici un fel de relație. Calitatea unui mecanism pentru controlul vizibilității identificatorilor depinde însă în primul rînd, de felul în care "ascunde" în anumite puncte din program informațiile ce nu sînt necesare: permite doar relațiile de vizibilitate utile.

Se mai introduc următoarele definiții:

Definiția 5.4.: Un mecanism pentru controlul vizibilității este *precis* din punct de vedere al relației *solicită referire* dacă și numai dacă funcția corespunzătoare, s_V, este surjectivă.

Definiția 5.5.: Un mecanism pentru controlul vizibilității este *precis* din punct de vedere al relației *permite referire* dacă și numai dacă funcția corespunzătoare, p_V, este surjectivă.

Definiția 5.6.: Un mecanism pentru controlul vizibilității este *precis* dacă și numai dacă este precis din punctul de vedere al ambelor relații adică, funcția de vizibilitate:

$$f_V = s_V \cup p_V \text{ este surjectivă.}$$

Definiția 5.7.: Un mecanism pentru controlul vizibilității este *imprecis* dacă și numai dacă este imprecis din punctul de vedere al ambelor relații (nici s_V, nici p_V nu sînt surjective).

Pe baza acestor definiții, se vor analiza în continuare cîteva limbaje tipice.

a) *PASCAL.* Se ia în considerare următoarea situație: două subprograme A și B, invizibile unul pentru celălalt, trbuie să apeleze exclusiv un al treilea program C (invizibil din orice altă unitate de program). Acestei situații îi corespunde un graf de vizibilitate $G_V \in \mathcal{G}$ cît și cele două subgrafuri $G_V(S)$ - de solicitare și $G_V(P)$ - de permisiune. Se observă ușor că situația dată nu poate fi descrisă exact printr-un program PASCAL. Rezultă că:

1) $\exists G_V(S) \in \mathcal{G}$ pentru care $\in P \in \mathcal{P}$ astfel încît $s_V(P) = G_V(S)$ și

2) $\exists G_V(P) \in \mathcal{G}$ pentru care $\in P \in \mathcal{P}$ astfel încît $p_V(P) = G_V(P)$

adică nici s_V, nici p_V nu sînt surjective.

Concluzia: PASCAL este *imprecis* din punct de vedere al vizibilității.

Observații:

1) Se poate arăta că în alte limbaje de programare (EDISON, CONCURRENT PASCAL) situația de mai sus se rezolvă fără dificultate.

2) Dintre limbajele tratate în cap. 1, SMALLTALK (§1.2.) și SIMULA 67 (§1.3.1) au aceeași poziție ca și PASCAL față de precizia mecanismului de vizibilitate.

b) *EDISON.* In ceea ce privește relațiile de tipul "permite referire", limbajul încorporează, în plus față de PASCAL, indicațiile explicite de export din module. Neindicîndu-se destinația, exportul se face fără discernămînt, în întreg domeniul procedurii înconjurătoare modulului fiind, deci, imprecis. Relațiile de tipul "solicită referire" se bazează, la fel ca și în PASCAL, exclusiv pe suprapunerea blocurilor unităților de program (proceduri și module).

Concluzia: EDISON este *imprecis* din punct de vedere al vizibilității.

Observații:

1) Comparativ cu limbajul PASCAL, se poate aprecia totuși că EDISON este *mai precis* în ceea ce privește relațiile de tipul "permite referire".

2) In aceeași poziție ca și EDISON față de precizia mecanismului de vizibilitate sînt toate limbajele de programare în care se indică doar entitățile exportate din diferite unități de program, fără a se preciza destinația exportului și fără indicarea explicită sau implicită a importului.

c) *CONCURRENT PASCAL.* Pentru specificarea relațiilor de tipul "permite referire", limbajul nu se deosebește decît prin formă de EDISON. In schimb, declararea componentelor sistem (procese, monitoare și clase) ca variabile în interiorul altor componente sistem precum și transmiterea monitoarelor ca parametri

între procese determină ca relațiile de tipul "solicită referire" să fie precis indicate.

Concluzia: CONCURRENT PASCAL este *precis* din punct de vedere al relației *solicită referire* și *imprecis* din punct de vedere al relației *permite referire*.

Observație: In aceeași poziție ca și CONCURRENT PASCAL față de precizia mecanismului de vizibilitate sînt toate limbajele bazate pe date abstracte în care se indică importul, iar pentru export se indică doar entitățile (nu și destinația). Exemple: EUCLID, MODULA, MODULA-2, ADA (care, datorită unor particularități va fi totuși tratat separat la punctul d)).

d) *ADA*. In acest limbaj, o condiție minimă a corectitudinii relațiilor de vizibilitate este următoarea: o relație de tip *solicită referire* implică obligatoriu relația de tip *permite referire* inversă. Aceasta înseamnă că o entitate *solicitată*, *permite* întotdeauna referirea. Rezultă următoarea definiție [WO87, HI80]:

Definiția 5.8.: Un graf de vizibilitate $G_V=\langle N,S,P\rangle$ este potrivit pentru ADA dacă și numai dacă $\forall$ (A,B)$\in$S, (B,A)$\in$P.

In plus față de limbajul PASCAL, mecanismul de vizibilitate incorporat în limbajul ADA conține elemente rezultate din definirea și utilizarea formelor specifice de date abstracte: pachete și task-uri (§ 1.3.2.). Rezultă un control mai riguros al relațiilor de tipul *permite referire* decît în PASCAL, în special prin posibilitatea de "ascundere" pentru exterior a unor informații definite într-un pachet. Totuși exportul se realizează fără discernămînt (nu se indică destinația) ceea ce înseamnă că relațiile de tipul *permite referire* nu sînt precise (identic cu punctul c)).

In privința relațiilor de tipul *solicită referire*, utilizarea clauzei **with** [AN83] permite specificarea exactă a importului în fiecare modul.

Concluzia: ADA este *precis* din punct de vedere al relației *solicită referire* și *imprecis* din punct de vedere al relației *permite referire*.

In lucrările [WO85, WO87] se prezintă o extensie a limbajului ADA denumită PIC/ADA, *precisă* din punctul de vedere al vizibilității. Mecanismul pentru controlul vizibilității din această extensie a fost completat cu două clauze diferite pentru specificarea explicită atît a importului (relațiile de tip *solicită referire*) cît și a

Tabelul 5.1.

LIMBAJUL	Precizia din punct de vedere al relației		CONCLUZIA
	solicită referire	permite referire	
PASCAL	imprecis	imprecis	limbajul este imprecis
SMALLTALK	imprecis	imprecis	limbajul este imprecis
SIMULA−67	imprecis	imprecis	limbajul este imprecis
EDISON	imprecis	imprecis	limbajul este imprecis
CONCURRENT PASCAL	precis	imprecis	
EUCLID	precis	imprecis	
MODULA	precis	imprecis	
MODULA−2	precis	imprecis	
ADA	precis	imprecis	
PIC/ADA	precis	precis	limbajul este precis
GYPSY	imprecis	precis	

exportului (relațiile de tip *permite referire*). In plus, luînd ca model limbajul de programare GYPSY (punctul e)), în cazul exportului se indică explicit și destinația.

e) *GYPSY*. Prezentarea acestui limbaj în analiza de față este justificată de următoarea particularitate a sa (apărută în premieră): prin așa numita *listă de acces*

[GC79] se poate specifica explicit și exact destinația exportului unei entități. In acest fel, relațiile de tip *permite acces* sînt rezolvate precis. In schimb, limbajul nu are nici o facilitate pentru indicarea importului.

Concluzia: GYPSY este *precis* din punctul de vedere al relației *permite referire* și *imprecis* din punct de vedere al relației *solicită referire*.

Rezultatele analizei de mai sus sînt prezentate sintetic în tabelul 5.1.

5.4. Analiza de domeniu prin metoda actualizărilor dinamice

Algoritmul de analiză descris în continuare a fost proiectat pentru a fi utilizat în compilatoarele PASCAL și PASCAL CONCURENT pentru calculatoarele din gama FELIX C [CE81d, CI80c]. In esență, el se bazează pe faptul că un identificator este accesibil în tabela de simboluri, împreună cu toate atributele sale, pe toată întinderea domeniului său și dispare din evidența analizorului cînd s-a ieșit din acest domeniu. In cazul redefinirii unui identificator, se înlocuiesc informațiile legate de identificatorul suspendat cu cele referitoare la noua stare; în momentul în care identificatorul redevine activ, vechile informații se restaurează. Zonele de date utilizate sînt descrise în PASCAL în figura 5.1.

```
type tipintrari=record
                  intrareinf:indexinf;
                  nivel:tipnivel
                end;
     tipinf=record ... end;
     tipsalv=record
               intraresalv:codlen;
               infsalv:tipintrari
             end;
     tipcomp=record
               numeintrare:codlen;
               numeinf:indexinf;
               inlantuire:indexcomp
             end;
var intrari:array [codlen] of tipintrari;
    info:array [indexinf] of tipinf;
    salvari:array [indexsalv] of tipsalv;
    componente:array [indexcomp] of tipcomp;
    virfsalv,salvvechi:indexsalv;
```

Fig. 5.1. Descrierea datelor utilizate în analiza de domeniu prin metoda actualizărilor dinamice

In tabela "info" se regăsesc, în ordinea apariției lor, toate entitățile programului, însoțite de atributele corespunzătoare. Entități diferite, desemnate în blocuri diferite prin același identificator, ocupă în "info" rînduri diferite. In limbajul PASCAL, în care orice entitate există doar în blocul în care a fost declarată, liniile tabelului "info" pot fi componente alocate dinamic, relocabile la încheierea analizei blocului respectiv.

Tabela "intrări" conține informații numai despre identificatorii vizibili în punctul din program în care se găsește analiza. Redefinirea unui identificator se reflectă prin modificarea liniei respective.

Mecanismul de analiză utilizează următoarele operații [CI83, HR77]:

a) La definirea fiecărui identificator se verifică dacă linia corespunzătoare din tabela "intrări" (indicele este codul lexical) conține sau nu un identificator definit pe același nivel (intrări[cl].nivel=nivelcurent); cazul afirmativ indică o definire multiplă și se semnalează ca eroare. Urmează transferarea liniei respective în stiva "salvări".

chiar dacă este vidă (identificatorul este la prima definire) și completarea ei cu atributele identificatorului nou introdus (procedura "salvează" din fig. 5.2.).

```
procedure salveaza(cl:codlen;index:indexinf);
begin
  if nivelcurent<>nivelglobal then
    begin
      virfsalv:=virfsalv+1;
      with salvari[virfsalv] do
        begin
          intraresalv:=cl;
          infsalv:=intrari[cl]
        end
    end;
  with intrari[cl] do
    begin
      intrareinf:=index;
      nivel:=nivelcurent
    end
end;
```

Fig. 5.2. Textul procedurii "salvează"

b) In momentul încheierii analizei unui bloc (revenirea pe nivelul anterior), se restaurează liniile din intrări care au fost salvate (procedura "scade_nivel" din fig. 5.3.). In cazul în care linia a fost vidă, restaurarea are efect de ștergere.

```
procedure scade_nivel;
var i:indexsalv;
begin
  ...
  for i:=virfsalv downto salvvechi do
    with salvari[i] do
      intrari[intraresalv]:=infsalv;
  virfsalv:=salvvechi-1;
  nivelcurent:=nivelcurent-1
end;
```

Fig. 5.3. Textul procedurii "scade_nivel"

c) La referirea unui identificator, prezența lui în "intrări" este echivalentă cu utilizarea curentă în cadrul domeniului său și, pe baza indicelui "intrări[cl].intrareinf" se poate ajunge în tabela "info" pentru alte verificări care țin de natura sau tipul entității respective.

Detalii și exemple privind aceste operații precum și alte operații specifice limbajelor PASCAL sau CONCURRENT PASCAL/FELIX C sînt tratate în [CI78b, CI83].

O problemă distinctă, cu unele particularități față de maniera expusă anterior este aceea referitoare la identificatorii cîmpurilor unui articol (atît PASCAL cît și CONCURRENT PASCAL) și la entitățile exportate (variabile, funcții și proceduri) dintr-un tip sistem (CONCURRENT PASCAL). Pentru uniformitatea metodei s-a considerat că tipurile **record**, **process**, **monitor** și **class** crează un nou nivel. Identificatorii cîmpurilor unui articol și identificatorii entităților exportate, definiți pe astfel de nivele, trebuie să fie vizibili pe nivele exterioare și anume în domeniul variabilei de care aparțin. In acest domeniu, identificatorii luați izolat sînt necunoscuți (invizibili) și nu pot fi referiți doar prin numele lor. Conform regulilor limbajului PASCAL [CE85a] referirea acestor entități trebuie însoțită de calificarea

lor prin numele variabilei (articol, proces, monitor sau clasă) corespunzătoare, într-unul din următoarele două moduri:

1) cu instrucțiunea **with**;

2) prin selectare directă (cu punct).

Aceste două modalități ale limbajului trebuie să se refere distinct în analiza de domeniu. Principala structură de date suplimentară este tabela "componente" (fig. 5.1.). Această tabelă conține, sub forma unei liste înlănțuite, identificatorii cîmpurilor unui articol, respectiv identificatorii entităților exportate dintr-un proces, monitor sau clasă. Adresa primului identificator dintr-o astfel de listă se păstrează în tabela "info", în înregistrarea corespunzătoare entității care a creat nivelul de care aparține lista.

Tabela "componente" se completează pe parcursul analizării nivelului pe care sînt definiți identificatorii respectivi, concomitent cu completarea tabelului "intrări". La terminarea analizării unui tip articol, proces, monitor sau clasă s-a încheiat nivelul respectiv și entitățile definite în el se șterg din "intrări" dar se păstrează în "componente".

Instrucțiunea **with** crează, de asemenea, un nou nivel. Din acest motiv, o calificare după modelul 1) de mai sus se materializează prin transferarea în "intrări" a listei din "componente" corespunzătoare variabilei menționate în instrucțiunea **with**. Pe întregul nivel, identificatorii transferați se consideră vizibili și pot fi referiți în conformitate cu natura și tipul lor.

În cazul în care calificarea se realizează după modelul 2), analiza de domeniu se execută astfel: se identifică în tabela "componente" lista entităților corespunzătoare variabilei cu care se face calificarea; se verifică dacă identificatorul calificat (componenta selectată) este sau nu prezent în această listă. Date suplimentare despre natura și starea componentei se iau direct din "info", prin intermediul indicelui "numeinf".

5.5. Analiza de domeniu prin metoda înlănțuirii identificatorilor pe nivele

Algoritmul descris în continuare a fost aplicat în două din compilatoarele pentru limbajul EDISON realizate la I.P.T.V. Timișoara. Problematica abordată nu diferă, în principiu, de cea de la metoda precedentă însă modalitatea de rezolvare a analizei de domeniu este cu totul diferită [CE84e, WI85].

Structura de date principală utilizată la analiză este tabela "tab" (fig. 5.4.), ale cărei linii corespund identificatorilor din program. La declararea unui identificator, se completează linia corespunzătoare lui cu atributele caracteristice entității declarate. În descrierea datelor din fig. 5.4. s-au reținut doar strict informațiile necesare analizei de domeniu.

```
type alfa=array [1..maxid] of char;
     indextab=1..maxtab;
var tab:array [indextab] of record
                              ident:alfa;
                              leg:indextab;
                              exp:integer
                            end;
    display:array [0..nivmax] of indextab;
    niv,nrmod,nrmode:integer;
```

Fig. 5.4. Descrierea datelor utilizate în analiza de domeniu prin metoda înlănțuirii identificatorilor pe nivele

Spre deosebire de metoda prezentată în §5.4., identificatorii sînt prezenți în permanență în tabelă din momentul definirii și pînă la încheierea analizei. Identificatorii definiți pe același nivel se înlănțuie între ei (cîmpul "leg").

Pentru desfășurarea analizei în conformitate cu regulile de domeniu ale limbajului

(§5.1.), sînt necesare următoarele operații:

a) la compilarea unei definiții sau declarații, în interiorul unui bloc de nivel "n", înaintea introducerii identificatorului în tabela "tab", se va verifica dacă el nu este deja prezent, datorită unei definiții sau declarații anterioare de pe același nivel;

b) la compilarea referinței unui identificator, într-un bloc de nivel "n", linia din tabelă corespunzătoare identificatorului referit trebuie să fi fost creată anterior.

Accesul la începutul listei înlănțuite corespunzatoare, în "tab", unui anumit nivel, este asigurat de tabela "display" (fig. 5.4). Aceasta conține, la un moment dat al analizei, liniile din tabela "tab" corespunzătoare ultimului identificator declarat în blocurile de nivel "n" (cel în curs de analizat), respectiv "n-1", "n-2", ... ,"1". Ultimul identificator din lanț (primul în ordinea introducerii în "tab"), are ca legătură linia 0.

Posibilitatea de a exporta identificatori de pe un nivel pe altul, determină activități suplimentare în cadrul algoritmului de analiză, și anume:

i) identificatorii exportați dintr-un modul de nivel "n" sînt introduși în lista corespunzătoare nivelului "n-1" imediat înconjurător;

ii) înaintea introducerii în tabelă a unui identificator neexportat "x" definit în interiorul unui modul, se verifică următoarele:

- dacă "x" este absent în lista corespunzătoare nivelului curent "n";
- dacă "x" este absent în lista corespunzătoare nivelului "n-1" ca entitate exportată din modulul curent de nivel "n";

iii) înaintea introducerii în tabelă a unui identificator exportat "x" definit în interiorul unui modul, se verifică următoarele:

- dacă "x" este absent în lista corespunzătoare nivelului curent "n";
- dacă "x" este absent în lista corespunzătoare nivelului "n-1";
- dacă blocul de nivel "n-1" este, de asemenea, modul, trebuie verificat în plus faptul că "n" este absent în lista corespunzătoare nivelului "n-2" ca entitate exportată din modulul de nivel "n-1".

Aceste verificări necesită numerotarea secvențială a modulelor programului și înregistrarea numărului respectiv în tabela "tab", într-un cîmp ("exp") specific fiecărei entități exportate, după următoarea regulă:

0 - identificator neexportat;

n - identificator exportat din modulul cu numărul "n".

Pe parcursul analizei, blocului curent și celui înconjurător le corespund două variabile "urmod" și respectiv "urmode" (fig. 5.4.) care pot să aibă una din următoarele valori:

0 - blocul nu este modul;

n - blocul este modulul cu numărul "n".

Principalele subprograme ale analizorului de domeniu care lucrează după principiile de mai sus sînt:

- *Funcția "loc"* (fig. 5.5): furnizează intrarea în tabela "tab" corespunzătoare declarației identificatorului "id" (transmis ca parametru); identificatorul este căutat, în ordine, în lista corespunzătoare nivelului curent "niv" iar, dacă nu este găsit, și în listele corespunzătoare nivelelor următoare: "niv-1", "niv-2", ... , "1"; în cazul în care, pînă la sfîrșit, identificatorul nu este găsit, se semnalează eroare și se returnează valoarea 0.

- *Procedura "enterpr"* (fig.5.6): introduce identificatorul "id" (transmis ca parametru) în tabela "tab"; este utilizată pe parcursul compilării declarațiilor într-o procedură în care nu există posibilitatea exportului de identificatori.

- *Procedura "entermod"* (fig. 5.7.): introduce identificatorul "id" (transmis ca parametru) în tabela "tab"; este utilizată pe parcursul compilării declarațiilor dintr-un modul; i se transmit ca parametri, în afara lui "id", "nrmod", "nrmode" și valoarea logică "export", indicînd dacă entitatea declarată este sau nu exportată.

Cele două proceduri apelează funcția "locație" (fig. 5.8.) care furnizează intrarea în tabela "tab" corespunzătoare identificatorului "id", dacă acesta este prezent în lista nivelului transmis ca parametru. În caz contrar se furnizează valoarea 0.

```
function loc(id:alfa):indextab;
var n,k:integer;
begin
  tab[0].ident:=id;
  n:=niv;
  repeat
    k:=display[n];
    while tab[k].ident<>id do
      k:=tab[k].leg;
    n:=n-1
  until (n<0)or(k<>0);
  if k=0 then eroare('identificator nedeclarat');
  loc:=k
end;
```

Fig. 5.5. Textul funcției "loc"

```
procedure enterpr(id:alfa);
begin
  if t=maxtab then stop else
    begin
      tab[0].ident:=id;
      if locatie(id,niv)<>0 then
        eroare('dubla definire') else
        begin
          t:=t+1;
          with tab[t] do
            begin
              ident:=id;
              leg:=display[niv];
              exp:=0
            end;
          display[niv]:=t
        end
    end
end;
```

Fig. 5.6. Textul procedurii "enterpr"

O analiză mai în detaliu a funcționării acestui algoritm este prezentată în [EC85a]. De asemenea, în lucrarea [CE84e] se analizează o paralelă între algoritmul din acest paragraf și cel tratat în paragraful anterior. Un algoritm aplicabil pentru analiza de domeniu a limbajului de programare EDISON dar substanțial mai complex și superior în ceea ce privește atît viteza (datorită utilizării tehnicii hashing) cît și memoria ocupată (prin eliberarea spațiilor utilizate la părăsirea unui bloc) este prezentat în [EC88b].

5.6. Mecanism pentru tratarea în compilare și execuție a parametrilor funcții și proceduri

Mecanismul prezentat în continuare a fost proiectat și ulterior a fost implementat în compilatorul PASCAL/FELIX C [CI81b, CE79]. In momentul proiectării acestui compilator, materialele de referință privind definirea limbajului de programare PASCAL erau lucrările [WI71, JW74] în care descrierea listelor de parametri formali corespunde diagramei sintactice din fig. 5.9.

```
procedure entermod(id:alfa;export:boolean;nrmod,nrmode:integer);
var k:integer;
begin
  if t=maxtab then stop else
    begin
      tab[0].ident:=id;
      if locatie(id,niv)<>0 then
        eroare('dubla definire') else
        begin
          k:=locatie(id,niv-1);
          if not export then
            if (k<>0)and(tab[k].exp=nrmod) then
              eroare('dubla definire') else
              begin
                t:=t+1;
                with tab[t] do
                  begin
                    ident:=id;
                    leg:=display[niv];
                    exp=0
                  end;
                display[niv]:=t
              end
            else if k<>0 then
              eroare('dubla definire') else
              begin
                if nrmode<>0 then k:=locatie(id,niv-2);
                if (k<>0)and(tab[k].exp=nrmode) then
                  eroare('dubla definire, pe nivele diferite') else
                  begin
                    t:=t+1;
                    with tab[t] do
                      begin
                        ident:=id;
                        leg:=display[niv-1];
                        exp:=nrmod
                      end;
                    display[niv-1]:=t
                  end
              end
        end
    end
end;
```

Fig. 5.7. Textul procedurii "entermod"

```
function locatie(id:alfa,nivel:integer):indextab;
var k:integer;
begin
  k:=display[nivel];
  while tab[k].ident<>id do
    k:=tab[k].leg;
  locatie:=k
end;
```

Fig. 5.8. Textul funcției "locație"

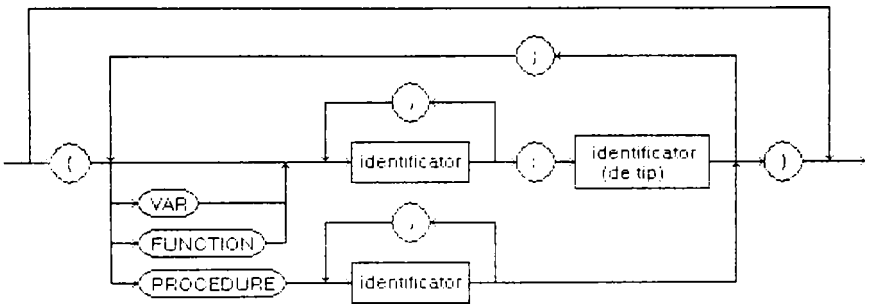


Fig. 5.9. Definirea sintactică a listei de parametri formali în varianta inițială a limbajului PASCAL

Deși introducerea parametrilor funcții și proceduri conferă un plus de generalitate limbajului, această facilitate, în forma sintactică din fig. 5.9, ingreunează foarte mult aspectele din compilare legate de analiza și semnalarea erorilor din listele de argumente. Dificultatea principală rezultă din faptul că procedurile și funcțiile parametri (atît formali cît și actuali) nu sînt însoțite de liste de parametri proprii. Necunoscînd numărul și tipul parametrilor proprii, subprogramele parametri nu sînt complet definite. Din această cauză, în compilare nu se pot verifica riguros aspectele semantice esențiale legate de apelul acestor subprograme, și anume:

- corespondența și compatibilitatea subprogramelor argumente cu cele parametri formali;
- corespondența (ca număr și tipuri) între parametrii formali și cei actuali, la apelul unui subprogram parametru formal.

Standardizarea limbajului PASCAL [AN79] a rezolvat în mod satisfăcător o parte dintre aceste probleme prin introducerea, la declararea unei proceduri sau funcții, a listelor de parametri formali proprii subprogramelor care sînt la rîndul lor parametri formali. În dialectele de PASCAL anterioare standardului, printre care și PASCAL/FELIX C, s-au adoptat diferite soluții pentru rezolvarea corectă și completă a problemelor de mai sus. Soluția prezentată în continuare se caracterizează prin aceea că transferă în execuție verificările care nu pot fi efectuate în faza de compilare.

O dificultate suplimentară, specifică realizării compilatorului PASCAL/FELIX C în limbaj de asamblare (ASSIRIS) este recursivitatea subprogramelor de analiză a apelurilor de proceduri și funcții: un subprogram argument poate avea la rîndul său ca parametri, subprograme.

Avînd în vedere cele de mai sus, o eventuală analiză completă a parametrilor funcții și proceduri în faza de compilare presupune, fie renunțarea la acele analize pentru care nu există informații suficiente, fie întregirea declarației subprogramului parametru formal în momentul primei lui utilizări, prin asimilarea parametrilor lui formali cu aceste prime argumente. Astfel, la următoarele apeluri, subprogramul poate fi considerat complet definit și verificăile se pot efectua prin comparație. Ambele variante sînt aproximative, nu conduc la semnalarea tuturor erorilor și, în cazuri particulare, semnalarea erorilor poate genera confuzii pentru programator.

Avînd în vedere ideea exploatată și în compilatorul FORTRAN/FELIX C, la prelucrarea opțiunii de compilare PCK [BA74], soluția prezentată în continuare lasă pentru faza de execuție verificarea compatibilității dintre argumentele subprogramului parametru formal și parametrii formali ai subprogramului argument.

5.6.1. Particularități încorporate la faza de compilare

a) Compilarea declarațiilor de subprograme argumente. Pentru a nu transfera în

execuție verificarea listelor de argumente a tuturor subprogramelor, indiferent dacă acestea sînt sau nu parametri. a apărut necesitatea indicării în program a acelor subprograme care pot figura ca argumente. In acest scop s-a propus introducerea la declararea subprogramelor a cuvîntului cheie **parm** inaintea cuvintelor cheie **procedure** sau **function**, după modelul următor [CE80]:

parm procedure nume(listă de parametri formali);

. . . . . .

begin . . . . . **end**;

Analiza acestui gen de subprogram coincide cu cea a procedurilor sau funcțiilor obișnuite (fără indicația **parm**) iar la sinteza codului obiect corespunzător se generează în plus apelul subprogramului de verificare în execuție a compatibilității listelor de parametri (formali și actuali), urmat de informațiile necesare despre numărul și tipul parametrilor formali (fig. 5.10.).

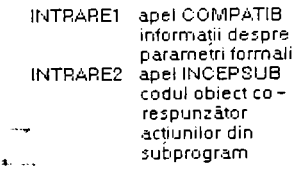


Fig. 5.10. Structura codului obiect generat pentru un **parm** subprogram

INTRARE1 și INTRARE2 sînt puncte de intrare în subprogram corespunzătoare apelului ca argument și, respectiv, corespunzătoare unui apel direct, obișnuit. Prezența acestor două puncte de intrare face deci posibilă și apelarea directă a subprogramului.

Utilizarea unei **parm procedure** (**function**) ca argument, va determina depunerea în stiva de date a programului, în locația destinată argumentului respectiv a adresei asociate etichetei INTRARE1, iar accesul la subprogram se va realiza prin indirectare cu această adresă.

b) *Compilarea apelurilor de subprograme parametri formali.* Deoarece argumentele parametrilor funcții sau proceduri nu au corespondenți direcți parametri formali, compilatorul va ocoli verificările de compatibilitate (ca număr și tipuri) între argumente și parametri pe care le face pentru lista unui subprogram apelat direct. Informațiile despre argumentele subprogramului parametru formal se vor transfera în faza de analiză semantică. In această fază, în locul din codul obiect corespunzător apelului subprogramului, se va genera cod obiect pentru următoarele operații (fig. 5.11.):

1) Crearea în stiva de date a programului a zonei argumentelor subprogramului, formată din:

- argumentele propriu-zise (valoare sau adresă);
- informații despre argumente (categorie, tip, lungime, etc.).

2) Apelul subprogramului argument de la eticheta INTRARE1, conform celor precizate la punctul a).

Observație: Modul de lucru ales poate fi utilizat și pentru apelul procedurilor standard READ și WRITE, avînd un număr variabil de parametri de diferite tipuri și lungimi. Pentru ca aceste apeluri să treacă prin compilare este suficient ca, la inițializarea tabelelor compilatorului, atributele asociate procedurilor READ și WRITE să indice că acestea sînt parametri proceduri ale programului principal. In acest caz apelul lor va fi permis pe tot parcursul programului și nu se va verifica nici numărul argumentelor, nici tipul lor. Subprogramele corespunzătoare, în execuție, acestor proceduri standard vor trebui să interpreteze o zonă de argumente ca cea din fig. 5.11. și să execute operațiile de intrare/ieșire indicate.

c) *Sinteza mecanismului.* Mecanismul prezentat constă din următoarele operații principale:

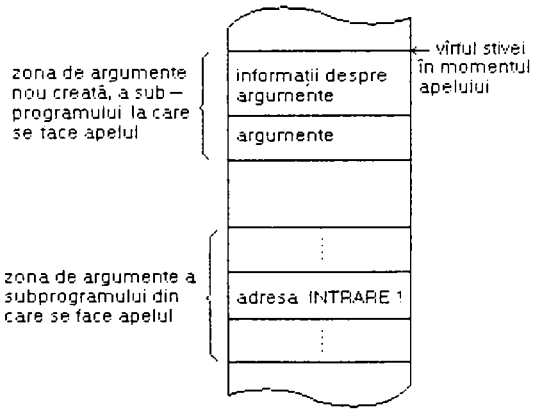


Fig. 5.11. Structura stivei în momentul execuției apelului unui subprogram parametru formal

1) La compilarea declarației unui subprogram care poate fi transmis ca argument (marcată prin cuvîntul cheie **parm**), se crează un punct suplimentar de intrare în subprogram (INTRARE1) la care se generează apelul subprogramului COMPATIB. Acest subprogram va realiza în faza de execuție o serie de verificări de compatibilitate amînate din compilare.

2) La compilarea unei liste de argumente care conține un subprogram se generează depunerea în stiva de date, în locația corespunzătoare argumentului respectiv, a adresei INTRARE1 asociată subprogramului sau, dacă subprogramul nu este declarat cu **parm**, se semnalează eroare.

3) La compilarea apelului unui parametru formal procedură sau funcție, se generează cod obiect pentru crearea în stiva de date a unei zone de argumente ce conține și atributele din compilare ale argumentelor respective ignorînd, în această fază, verificarea numărului și tipurilor argumentelor.

5.6.2. Mecanismul în faza de execuție. Subprogramele COMPATIB și INCEPSUB

Subprogramele COMPATIB și INCEPSUB, al căror apel se generează la compilare (§5.6.1, punctul a)), sînt încorporate în codul obiect al programului în faza de editare a legăturilor.

Subprogramul COMPATIB interpretează și compară zona de informații despre parametrii formali, aflată în codul obiect imediat după apel (fig. 5.10.) și respectiv zona de informații despre argumente, care se găsește în vîrful stivei (fig. 5.11.). In final, COMPATIB decide dacă apelul subprogramului argument s-a efectuat corect și elimină din vîrful stivei zona de informații despre propriile sale argumente.

Subprogramul INCEPSUB execută operațiile obligatorii la intrarea într-un subprogram: rezervarea în stivă a înregistrării de activare, poziționarea registrelor generale, etc. El nu este specific unei **parm procedure (function)**.

Capitolul 6

REALIZAREA CONCEPTELOR DE CONCURENTA IN COMPILATORUL PENTRU LIMBAJUL DE PROGRAMARE PASCAL CONCURENT/FELIX C

6.1. Generalităţi. Organizarea datelor compilatorului.

Avînd ca model limbajele PASCAL [WI71] şi CONCURRENT PASCAL [BH75c], precum şi limbajele de nivel înalt FORTRAN şi COBOL, şi ţinînd cont de posibilităţile limbajului de asamblare ASSIRIS, la Centrul de Calcul al I.P.T.V. Timişoara s-a implementat, prin realizarea compilatorului şi a unui executiv adecvat, limbajul de programare CONCURRENT PASCAL pe calculatoarele din gama FELIX C. Noua variantă a limbajului se numeşte PASCAL CONCURENT. Intrucît FELIX C (256, 512, 1024) sînt calculatoare de putere medie, înzestrate cu un sistem de operare care satisface la această oră necesităţile majorităţii aplicaţiilor obişnuite, s-a considerat util ca noul compilator să genereze la ieşire module BT, care urmează să fie editate de editorul de legături al sistemului şi nu programe autonome, cu rol de sistem de operare. In acest fel, în versiunea românească a limbajului PASCAL CONCURENT nu se vor scrie sisteme de operare, aşa cum a recomandat P. B. Hansen pentru versiunea iniţială [BH75a], ci subsisteme de natură conversaţională sau în timp real, executabile într-o partiţie fixă sub sistemul de operare al calculatorului. Programul obiect concurent rezultat, are posibilitatea să încorporeze module BT din orice alt limbaj (ASSIRIS, FORTRAN, COBOL), asigurîndu-se astfel nu numai o legătură simplă cu alte limbaje secvenţiale, ci chiar modalitatea de utilizare a unor programe sau părţi de programe testate şi funcţionale, dar realizate în alte limbaje.

Faptul că PASCAL CONCURENT are la bază limbajul PASCAL explică numeroasele similitudini între ele. In cele ce urmează se vor trata numai aspectele specifice limbajului PASCAL CONCURENT, legate de realizarea conceptului de programare concurentă. Aceste aspecte vor fi raportate la PASCAL şi prezentate în acest limbaj sub forma unor fragmente de program care îndeplinesc rolul unor algoritmi de principiu, fără a avea corespondenţă imediată în compilatorul propriu-zis, scris în ASSIRIS.

Din punctul de vedere al structurii segmentate şi pe faze, al funcţiilor pe care le îndeplinesc fazele de compilare, nu există nici o deosebire între compilatorul PASCAL CONCURENT şi cel pentru limbajul PASCAL [CI84b] (cap. 7.). Totuşi deosebirile dintre PASCAL CONCURENT şi PASCAL, scoase în evidenţă în cap. 1. şi cap. 3. se reflectă corespunzător în compilator determinînd analize şi generări de cod specifice, dintre care unele vor fi punctate în capitolul următor.

In acelaşi timp, realizarea unui limbaj concurent pornind de la PASCAL implică şi proiectarea unui nucleu (executiv) adecvat. Nucleul înglobează funcţiile de control şi sincronizare ale proceselor, rezolvarea intrărilor/ieşirilor, etc., probleme ce vor fi prezentate în cap. 8.

In fig. 6.1. se prezintă, în limbajul PASCAL, structura principalelor tabele ale compilatorului. Această structură ar putea fi reală în cazul în care partea din procesul de compilare care urmează după analiza sintactică s-ar efectua într-o singură trecere şi, din acest motiv, nu corespunde decît în principiu compilatorului existent. Complexitatea excesivă a organizării acestor tabele reprezintă de altfel un motiv în plus în favoarea compilării în mai multe treceri, care a fost adoptată (cap. 7).

Principala tabelă utilizată de compilator este "informaţii" (de tipul "tipinf"). Fiecărei entităţi definite în programul concurent i se asociază un rînd în această tabelă, în care se memorează atributele deduse sau calculate pentru entitatea respectivă. Există şi posibilitatea ca unei entităţi să-i corespundă două înregistrări în "informaţii" în cazul în care ea este interpretată de compilator sub două aspecte

```
type tipfel=(constindex,constreal,sircaractere,variabila,parametru,
             cimp,scalar,compsistem,sirasteptare,rutina,multime,
             program,tablou,articol,vartemp,nedefinit,tipconst);
     tipcodnou=0..codnoumax;
     indexinf=0..infmax;
     indexcomp=0..compmax;
     modintrare=(clasa1,monitor1,proces1,proced1,procedext1,
                 functie1,functieext1,program1,articol1);
     modiesire=(constscurta2,constlunga2,proced2,program2,procesext2,
                clasaext2,monitorext2,proces2,clasa2,monitor2,
                standard2,nedefinit2);
     deplasament=integer;
     tipcompozitie=(activ,pasiv,queue);
     tipcategorie=(cnedefinita,cvaloare,crutina,ctip);
     modapel=(direct,indirect,adresa,expresie);
     tipcontext=(rezfunc,varext,var,paramvar,varuniv,paramconst,ccimp,
                 constuniv,expr,constanta,paramsalvat,constwith,varwith);
     tipnivel=0..nivelmax;
     tipacces=(general,extern,intern,incomplet,nerezolvat,calificat,
               functional,nedefinit);
     tipintrari=record
                  intrareinf:indexinf;
                  nivel:tipnivel;
                  acces:tipacces
                end;
     codlex=0..codlexmax;
     tipclasa=(varcl,rutcl,consticl,constrcl,constscl,defcl,nedefcl,
               constfcl,progcl,eticcasecl,valfunccl);
     tipcomp=record
               numeintrare:codlex;
               numeinf:indexinf;
               inlantuire:indexcomp
             end;
     indexstiva=0..stivamax;
     tipinf=record
              codnou:tipcodnou;
              case categorie:tipcategorie of
                cvaloare:(context:tipcontext;
                          apel:modapel;
                          vdepl:deplasament;
                          vmod:modiesire;
                          case fel:tipfel of
                            constindex:(tipconst:tipcodnou;
                                        valiconst:integer);
                            realconst:(realdepl:integer);
                            sircaractere:(lungimesir,deplsir:integer);
                            variabila:(tipvar:indexinf);
                            parametru:(tipparam,paramurmator:indexinf);
                            vartemp:(tipvtemp:tipcodnou);
                            cimp:(tipcimp:indexinf));
                crutina:(rmod:modiesire);
                         nrrutina:integer;
                         dimparam,dimvar,dimstiva:deplasament;
                         case fel:tipfel of
```

Fig. 6.1. Structura principalelor date ale compilatorului

```
                        rutina:(paramrut:indexinf;
                                tiprut:tipcodnou);
                        program:(paramprog:indexinf;
                                 interfata:indexcomp));
              ctip:(dim:deplasament;
                    compozitie:tipcompozitie;
                    case fel:tipfel of
                      scalar:(tipdomeniu:tipcodnou);
                      compsistem:(rutinit:indexinf;
                                  ext:indexcomp;
                                  cmod:modiesire;
                                  lvar:deplasament);
                      tablou:(tipindice:tipcodnou;
                              tipelement:indexinf);
                      articol:(cimpuri:indexcomp);
                      tipconst:(min,max:integer))
          end;
     tipstiva=record
                case clasa:tipclasa of
                  varcl:(vtip:indexinf);
                  progcl:(paramprog:indexinf);
                  rutcl:(paramrutin:indexinf);
                  varfunccl:(tipfunc:tipcodnou);
                  consticl:(tipconsti:tipcodnou;
                            valconst:integer);
                  constrcl:(constrdepl:integer);
                  constscl:(constslung,constsdepl:integer);
                  eticcasecl:(etic,index:integer);
                  defcl:(definf:indexinf;
                         defintrare:codlex)
              end;
     tipsalv=record
               intraresalv:codlex;
               infsalv:tipintrari
             end;
     indexsalv=0..salvmax;
var intrari:array [codlex] of tipintrari;
    informatii:array [indexinf] of tipinf;
    componente:array [indexcomp] of tipcomp;
    salvari:array [indexsalv] of tipsalv;
    modcurent:0..11;
    necomp:set of modiesire;
    acestnivel:tipnivel;
    stiva:array [indexstiva] of tipstiva;
    virfstiva:indexstiva;
    intrnedef,acestparam,primulparam:indexinf;
    felactiv:set of tipfel;
    apeladr,univers:set of tipcontext;
    nivelglobal,nivelsistem:tipnivel;
    v,lista:indexcomp;
    clnedef:codlex;
    accesext:set of tipacces;
    alocstiva,nrparam,nrvar,nerez:integer;
    nevar:set of modintrare;
```

Fig. 6.1. (continuare)

diferite (variabila contor din **for**, variabilele temporare din instrucțiunile **with**, procesele, monitoarele și clasele care sînt în același timp tipuri și rutine, etc.).

Identificarea entității se realizează prin cîmpul "codnou", care preia rolul deținut pînă în faza3 de codul lexical. Necesitatea acordării unui nou cod în această fază se explică prin posibilitatea apariției unor entități noi (ca cele amintite mai sus) și prin dispariția unui mare număr de entități, cum sînt constantele întregi și logice, identificatorii din tipurile enumerare, etc. [HR77, CI79b].

După cum se observă în fig. 6.1., structura unei înregistrări din "informații" depinde de categoria din care face parte entitatea iar, în cadrul categoriei, de felul ei. Din acest motiv, caracteristicile memorate sînt dintre cele mai diverse: contextul în care apare entitatea, cu rolul de apel, deplasamentul calculat în stiva de date din execuție, valoarea, lungimea sau tipul entității, etc.

Accesul la "informații" este asigurat prin tabela "intrări"; indexul în "intrări" este codul lexical. Mai multe entități pot fi reprezentate pe nivele diferite, eventual chiar interioare unul altuia, prin același identificator. Corespondența identificator - cod lexical fiind biunivocă, aceste entități vor fi caracterizate prin același cod lexical, care va puncta un singur rînd în tabela "intrări". Această situație permite redefinirea sau redeclararea unui identificator, cu alt sens decît cel inițial, pe alt nivel. Compilatorul trebuie deci să asigure prezența unei entități în tabela "intrări" doar pe parcursul compilării nivelului ei iar, pe de altă parte, la redefinirea identificatorului pe un nivel interior, trebuie să fie în măsură să înlocuiască vechea entitate cu cea nouă, păstrînd în același timp datele despre cea veche pentru a fi refăcute la închiderea nivelului interior. In acest scop se utilizează tabela suplimentară "salvări" [CC81].

Completarea unui rînd în "intrări" este întotdeauna precedată de trecerea rîndului respectiv în "salvări", chiar dacă pe el n-a fost completat nimic. La sfîrșitul unui nivel se vor pune pe starea inițială toate rîndurile din "intrări" completate în acel nivel. In cazul în care inițial rîndul a fost gol, această refacere este echivalentă cu o ștergere. Mecanismul prezentat mai sus asigură, de asemenea, prezența entității în tabela "intrări", doar pe parcursul compilării domeniului său și nu este cazul să se execute verificări suplimentare privind vizibilitatea (rezolvarea este similară celei prezentate în §5.4.). Necesitatea salvărilor și refacerilor justifică și existența a două tabele ("intrări" și "informații") pentru caracterizarea completă a entităților din program, dintre care tabela "informații" este fixă, în sensul că un rînd odată completat nu se mai modifică pînă la sfîrșitul compilării.

Structura tabelei "intrări" se poate urmări în fig. 6.1. (tipul "tipintrări"). Un rînd din "salvări" conține, în plus, și codul lexical al entității salvate, necesar pentru refacerea rîndului corespunzător din "intrări".

Tabela "componente" conține, sub formă de liste de înregistrări, cîmpurile unui articol definit în program sau entitățile (proceduri, funcții și variabile) externe unui nivel. Fiecare înregistrare are în componența sa codul lexical și indexul în tabela "informații" corespunzător entității la care se referă.

Necesitatea formării și păstrării acestor liste se explică prin aceea că atît cîmpurile cît și procedurile, funcțiile și variabilele externe se utilizează în afara domeniului în care au fost declarate [CE85a], în acel moment din compilare nemaifiind prezente în tabela "intrări". Capul listei (de tipul "indexcomp"), se păstrează în tabela "informații", în rîndul asociat entității ale cărei componente figurează în listă, și prin intermediul căreia se vor califica aceste componente (fig. 6.1.).

Algoritmii exemplificați în continuare se bazează pe existența unei stive (tabela "stiva"). In fiecare rînd al stivei se introduc date referitoare la o entitate, sub formă de indici în tabelele "intrări" și "informații". Cu caracter tranzitoriu, stiva poate conține și date de felul celor memorate în "informații". Conținutul stivei se referă întotdeauna la o singură construcție PASCAL CONCURENT, sfîrșitul compilării unei fraze independente determinînd reducerea ei completă [CI80c].

Observație: In descrierile de date din fig. 6.1. nu s-a utilizat facilitatea de alocare dinamică a memoriei din limbajul PASCAL întrucit aceasta nu există în ASSIRIS, limbaj în care este scris compilatorul real.

```
procedure initializare;
begin
  necomp:=[proced2,procesext2,clasaext2,monitorext2,program2,nedefinit2];
  felactiv:=[coada,compsistem];
  apeladr:=[paramvar,varuniv];
  univers:=[varuniv,constuniv];
  accesext:=[extern,nerezolvat];
  nevar:=[clasa1,monitor1,proces1,functie1,functieexti]
end;

procedure crestenivel(mod:modintrare);
begin
  if acestnivel>=nivelmax then abort
    else acestnivel:=acestnivel+1;
  salvnivel;
  if modcurent in necomp then
    if mod<>articol1 then
      begin
        eroare(incuibare_interzisa);
        modcurent:=clasa2
      end;
  case mod of
    clasa1:modcurent:=clasa2;
    monitor1:modcurent:=monitor2;
    proces1:modcurent:=proces2;
    proced1,functie1:modcurent:=proced2;
    procedext,functieext:case modcurent of
                           clasa2:modcurent:=clasaext2;
                           monitor2:modcurent:=monitorext2;
                           proces2:modcurent:=procesext2
                         end;
    program1:modcurent:=program2;
    articol1:modcurent:=nedefinit2
  end;
  initnivel
end;

procedure scadenivel;
begin
  refnivel;
  acestnivel:=acestnivel-1
end;
```

Fig. 6.2. Inițializarea compilatorului și gestionarea nivelelor

In fig. 6.2. se prezintă partea din rutina de inițializare a compilatorului care interesează în exemplele următoare, precum și procedurile apelate la începutul și la sfîrșitul compilării unui nivel ("crestenivel" și respectiv "scadenivel").

Funcțiile procedurii "crestenivel" sînt următoarele [CI80c]:

- stabilește așa-numitul "modcurent", care depinde de definiția sau declarația ce determină nivelul și, uneori, de "modcurent"-ul anterior; acest "modcurent" este utilizat de compilator pentru a decide corecta declarare a parametrilor și variabilelor;

- salvează, într-un tabel suplimentar, datele specifice nivelului întrerupt: indexul de ocupare în "salvări", capul listei în curs de construcție în tabela "componente" ("listă"), "modcurent"-ul nivelului întrerupt, pointerul de alocare a stivei în execuție ("alocstiva"), etc.; în acest scop apelează procedura "salvnivel", care nu este explicitată în textul PASCAL din fig. 6.2.;
- apelînd "initnivel" inițializează, pentru noul nivel, entitățile salvate anterior;
- execută o primă verificare a modului de suprapunere între nivele.

Procedura "scadenivel" are rolul de a restabili nivelul anterior, utilizînd datele salvate de "creștenivel".

6.2. Compilarea tipurilor și variabilelor active

Sirurile de așteptare, procesele, monitoarele și clasele sînt componente active ale unui program PASCAL CONCURENT [CE81c], întrucît prezența și gestionarea lor provoacă și rezolvă totodată efectul de "concurență". Din acest motiv tipurile sistem **process**, **monitor**, **class** precum și tipul **queue** reprezintă tipurile active ale limbajului PASCAL CONCURENT. Variabilele de tip activ vor fi la rîndul lor variabile active. Specificul concurent al limbajului se manifestă în primul rînd în modul în care se utilizează tipurile și variabilele active.

Caracterul special al compilării tipurilor sistem derivă din dublul aspect sub care sînt privite procesele, monitoarele și clasele: descrierea de structuri de date și un set de acțiuni asociat. Din această cauză unui tip sistem i se rezervă două înregistrări în tabela "informații", un "codnou" diferit, ca și cînd tipul sistem s-ar diversifica în două entități diferite. Prima înregistrare caracterizează tipul sistem ca tip al limbajului, permițînd declararea unor variabile de acest tip, iar a doua îl caracterizează ca rutină (procedura inițială), capabilă să califice procedurile și funcțiile sale externe.

```
procedure ctipactiv;
var k:indexinf;
begin
  k:=stiva[virfstiva-1].definf;
  with informatii[k] do
    if cmod=proces2 then
      begin
        dim:=cuvint;
        lvar:=0
      end else
      begin
        dim:=alocstiva+cuvint+
             informatii[stiva[virfstiva].definf.dimparam];
        lvar:=alocstiva
      end
end;
```

Fig. 6.3. Calculul dimensiunii unui tip sistem

In fig. 6.3. se indică modul de calcul al dimensiunii ("dim") unui tip sistem, dimensiune care va fi apoi atribuită tuturor variabilelor de tipul respectiv, fiind utilizată la alocarea stivei pentru variabile.

Vîrful stivei de compilare ("vîrfstivă") corespunde rutinei sistem asociată tipului sistem memorat la "vîrfstivă"-1. Tipurile și variabilele procese au dimensiunea fixă: "dimcuvint", întrucît zona de memorie necesară unui proces nu se rezervă în locul din stivă corespunzător procesului ca variabilă, ci într-o zonă separată [CE85d]. In momentul efectuării acestui calcul, "alocstiva" indică dimensiunea zonei de variabile din rutina sistem. In dimensiunea finală a tipurilor și variabilelor monitoare și clase

intră pe lîngă această zonă și zona de parametri ("dimparam"). Cuvîntul rezervat în plus, nu are semnificație decît la monitoare, fiind utilizat pentru memorarea adresei porții monitorului. Uniformizarea calculului deplasamentelor finale ale variabilelor și parametrilor în faza 5 (cap. 7.), presupune rezervarea acestui cuvînt la toate tipurile sistem.

Spre deosebire de procese, monitoarele și clasele există ca variabile doar pe parcursul executării nivelului din care fac parte. Din această cauză variabilele monitoare și clase, avid dimensiunea calculata după modelul din fig. 6.3., se rezervă în stiva de date în locul corespunzător declarării lor.

Partea de declarare a variabilelor, ca și cea de definire a tipurilor de altfel, formează zona descriptivă a unui program PASCAL CONCURENT. Acestei zone nu-i corespunde cod obiect executabil, compilatorul mărginindu-se la a face o serie de verificări și calcule a căror rezultate se memorează în tabela "informații" și se utilizează la generarea codului obiect pentru partea de instrucțiuni a programului.

Deși se declară după aceleași reguli sintactice ca și variabilele obișnuite, cele active sînt supuse unor restricții suplimentare, specifice limbajului PASCAL CONCURENT și strîns legate de implementarea concurenței între procese și de funcția de sincronizare a monitoarelor [CI80b].

In fig. 6.4. se prezintă o sinteză a funcțiilor compilatorului pentru analiza declarațiilor de variabile, din fazele 3 și 4 (cap. 7.), aceasta fiind concentrată mai ales în faza 4 [CI79c]. "nr" indică numărul de variabile din sublista de variabile de același tip, tratate la un singur apel al procedurii "listavar". Vîrful stivei de compilare corespunde tipului variabilelor din listă, iar de la "vîrfstiva"-1 la "vîrfstiva"-"nr" se găsesc indicii asociați variabilelor propriu-zise. Funcțiile compilatorului concentrate în procedura "listavar" sînt următoarele [CI80c, CI79c]:

a) Verificarea concordanței dintre tipul variabilei active și locul declarării ei, astfel:

- o variabilă de tip activ, nu poate fi declarată decît într-un monitor, clasă sau proces (inclusiv procesul inițial);
- o variabilă de tip șir de așteptare ("queue") sau care are în componența sa un șir de așteptare, nu poate să apară decît într-un monitor;
- procesele nu se declară decît în procesul inițial.

b) Stabilirea dimensiunii variabilei respective după dimensiunea tipului corespunzător ("aceastădim").

c) Calcularea deplasamentului variabilei în zona de variabile ("vdepl") ținînd cont de dimensiunea variabilelor anterioare. Acest deplasament nu este definitiv. In faza 5 el va fi corectat și translatat și va deveni pozitiv (utilizabil în cuvîntul instrucțiune al calculatorului FELIX C-256). Stabilirea lui în această formă negativă permite însă organizarea zonei de date pentru un bloc de așa manieră încît să fie simplă reprezentarea ei în execuție (principiul stivei).

d) Gestionarea dimensiunii totale a zonei de variabile pentru un bloc ("alocstiva") care va deveni în final o caracteristică a blocului respectiv și va fi memorată în tabela "informații".

e) Completarea, în "informații", a înregistrării corespunzătoare variabilei, cu informațiile deduse sau calculate pentru variabila respectivă (atribute).

Compilarea proceselor, monitoarelor și claselor, ca rutine, presupune următoarele operații:

- analizarea listei parametrilor formali;
- etichetarea rutinei (procedura inițială) cu un număr de ordine corespunzător poziției sale în textul sursă; acest număr va fi asociat blocului de instrucțiuni și va identifica rutina pînă în etapa generării codului obiect cînd, referirea rutinei prin eticheta ei, se va înlocui cu saltul (**bal**,6) la adresa de început a blocului de instrucțiuni;
- completarea în tabela "informații" a dimensiunii zonei de parametri și a zonei de variabile a rutinei, cunoscute la sfîrșitul compilării listei de parametri și respectiv la

```
function definit:boolean;
begin
  definit:=stiva[virfstiva].clasa<>nedefcl
end;

function intrare:indexinf;
begin
  intrare:=stiva[virfstiva].definf
end;

procedure listavar;
var i,nr:integer;
    aceastadim:deplasament;
    vtip,k:indexinf;
begin
  if definit then vtip:=intrare
    else vtip:=intrnedef;
  k:=intrare;
  with informatii[k] do
    begin
      if fel in felactiv then
        begin
          if modcurent in necomp then eroare(activerr);
          if compozitie=queue then
            if modcurent<>monitor2 then eroare(queueerr);
          if fel=compsistem then
            if cmod=proces2 then
              if acestnivel<>nivelglobal then eroare(proceserr);
        end;
      aceastadim:=dim
    end;
  citeste(nr);
  for i:=nr downto 1 do
    begin
      k:=stiva[virfstiva-i].definf;
      with informatii[k] do
        begin
          categorie:=cvaloare;
          vmod:=modcurent;
          context:=var;
          fel:=variabila;
          tipvar:=vtip;
          alocstiva:=alocstiva+aceastadim;
          vdepl:=-alocstiva
        end
    end;
  virfstiva:=virfstiva-nr-1
end;
```

Fig. 6.4. Analiza declarațiilor de variabile

începutul compilării blocului de instrucțiuni; aceste date se transmit în codul obiect generat, pînă în execuție și se utilizează la gestionarea stivei de variabile;

- analizarea semantică a instrucțiunilor din procedura inițială și generarea codului obiect corespunzător.

Interpretarea tipurilor sistem ca rutine conferă posibilitatea și, în același timp, necesitatea înzestrării lor cu liste de parametri. Acești parametri vor lua valoarea argumentelor precizate la inițializarea variabilelor sistem corespunzătoare. Din acest punct de vedere, în compilator, specificul concurent al limbajului se manifestă mai pregnant în subprogramele de tratare a parametrilor formali (fig. 6.5.) [CC81].

```
procedure verifslp;
var k:indexint;
    modi:modintrare;
begin
  k:=intrare;
  with informatii[k] do
    begin
      citeste(modi);
      case modi of
        monitor1,proces1,clasa1:if fel=compsistem then
                                  if cmod=monitor2 then {corect} else
                                    if (cmod=clasa2)and(modi=clasa1)
                                      then {corect} else
                                      eroare(paramerr1)
                                  else eroare(paramerr2);
        proced1,functie1:{corect};
        procedext1,functieext1:if queue=compozitie
                               then eroare(paramerr3);
        program1:if fel in felactiv then eroare(paramerr4)
      end
    end
end;

procedure listaparam;
var c:tipcontext;
    modi:modintrare;
    sf:boolean;
begin
  repeat
    citesteslp;
    punetip;
    citeste(c);
    citeste(modi);
    if c in apeladr then
      if modi in nevar then eroare(paramerr_5);
    if c in univers then
      if informatii[intrare].fel in felactiv then eroare(paramerr_6);
    verifslp;
    sublp(c)
  until sf;
  sfirsitlp
end;
```

Fig. 6.5. Tratarea listelor de parametri formali

O listă de parametri se consideră ca fiind formată din mai multe subliste ("slp") alcătuite din parametri de același tip. Compilarea unei liste de parametri constă din repetarea pentru fiecare sublistă a următoarelor operații (procedura "listaparam"):

a) Se citește fiecare parametru din sublistă și se rezervă rindul corespunzător din "informații"; în același timp parametrul se depune și în stiva ("citeșteslp").

b) La încheierea sublistei de parametri se apelează "punetip" care verifică dacă

```
procedure sublp(c:tipcontext);
var i,nr:integer;aceastadim:deplasament;
    k,ptip:indexinf;apelc:modapel;
begin
  if definit then ptip:=intrare
    else ptip:=intrnedef;
  citeste(nr);
  nrparam:=nrparam+nr;
  k:=intrare;
  with informatii[k] do
    if (c in apeladr)or(dim>cuvint) then aceastadim:=cuvint
      else aceastadim:=dim;
  for i=nr downto 1 do
    with stiva[virfstiva-i] do
      if clasa=defcl then
        begin
          with informatii[definf] do
            begin
              categorie:=cvaloare;
              context:=c;
              calculeaza(apelc);
              apel:=apelc;
              vdepl:=aceastadim;
              fel:=parametru;
              tipparam:=ptip;
              if primulparam=nil then primulparam:=definf
                else informatii[acestparam].paramurmator:=definf;
              acestparam:=definf;
              paramurmator:=nil
            end;
          intrari[defintrare].acces:=intern
        end;
      virfstiva:=virfstiva-1
end;

procedure sfirsitlp;
var vdim:deplasament;
    i:integer;k:indexinf;
begin
  alocstiva:=cuvint;
  for i:=0 to nrparam-1 do
    begin
      k:=stiva[virfstiva-i].definf;
      with informatii[k] do
        begin
          vdim:=vdepl;
          vdepl:=alocstiva;
          alocstiva:=alocstiva+vdim;
          vmod:=modcurent
        end
    end;
  alocstiva:=alocstiva-cuvint;
  virfstiva:=virfstiva-nrparam
end;
```

Fig. 6.6. Analiza parametrilor formali în cadrul unei liste

tipul referit a fost definit anterior și, în caz afirmativ îl depune în "stiva".

c) Verifică direct, sau apelînd "verifslp", corecta utilizare a tipurilor active în declararea parametrilor formali. Aceasta trebuie să țină cont de următoarele restricții:

- din necesitatea realizării funcției de sincronizare a monitoarelor, parametrii proceselor, monitoarelor și claselor pot fi de orice tip pasiv precum și de tipurile active "monitor" și "queue". De asemenea o clasă poate avea parametri de tip "class";
- tipul "queue" nu poate fi utilizat în listele de parametri ale procedurilor și funcțiilor externe; în consecință, compilatorul nu permite ca un program concurent să execute operații asupra șirurilor de așteptare, decît în locul declarării lor (în monitoare);
- tipurile active nu pot fi declarate **univ** și nu pot apare în declararea programelor secvențiale;
- procesele, monitoarele, clasele și funcțiile de orice fel nu pot avea parametri formali variabili, neavînd astfel nici posibilitatea de a modifica variabile declarate în alte blocuri; în acest mod se asigură protecția datelor de la un bloc la altul.

d) Apelează procedura "sublp" (fig. 6.6.) care execută la rîndul ei următoarele operații:

- gestionează numărul de parametri din listă;
- stabilește dimensiunea fiecărui parametru din sublistă, în funcție de modul de apel și de dimensiunea tipului corespunzător;
- completează înregistrările din tabela "informații" care corespund parametrilor din sublistă, înlănțuind aceste rînduri între ele; înlănțuirea este necesară în vederea stabilirii compatibilității de număr și tip a parametrilor formali cu cei actuali, la lansarea proceselor, monitoarelor și claselor, prin instrucțiunea **init** [CI79d].
- completează cîmpul "acces", în tabela "intrări";
- reduce, din stivă, rîndul corespunzător tipului parametrilor din sublistă.

La sfîrșitul compilării listei de parametri, se apelează "sfîrșitlp" pentru calcularea deplasamentului fiecărui parametru în lista din care face parte ("vdepl"). In acest calcul se ține cont de dimensiunea parametrilor anteriori. In final, rezultă dimensiunea totală a zonei de parametri ("alocstiva"), care este o caracteristică a rutinei sistem și se memorează în tabela "informații".

6.3. Compilarea declarațiilor externe

In limbajul PASCAL CONCURENT se pot declara proceduri, funcții și variabile externe. Modul de utilizare în program al entităților externe este sinonim cu cel al cîmpurilor unui articol, rolul articolului fiind îndeplinit de variabila proces, monitor sau clasă în care s-au declarat entitățile respective.

Pentru a uniformiza, în compilare, tratarea entităților externe cu aceea a cîmpurilor, se utilizează tabela "componente", care conține atît liste de cîmpuri ale aceluiași articol cît și liste de proceduri, funcții și variabile externe ale aceluiași proces, monitor sau clasă. O astfel de listă se formează pe parcursul compilării tipului sistem respectiv și servește pentru verificarea corectei utilizări a entităților externe în restul programului PASCAL CONCURENT.

Compilarea procedurilor externe se execută în linii mari în același mod cu cel precizat anterior pentru procedurile inițiale ale proceselor, monitoarelor și claselor, cu indicația suplimentară că li se asociază în plus o înregistrare în tabela "componente".

Tratarea declarațiilor de variabile externe este sintetizată în fig. 6.7.

Procedura "înlănțuie" crează și completează înregistrarea din tabela "componente" corespunzătoare variabilei. Se remarcă de asemenea faptul că nu pot fi declarate externe variabile de tip activ. Din necesitatea protejării datelor din monitoare și procese, nu este permisă declararea variabilelor externe decît în clase care, nu pot fi

```
procedure inlantuie(e:indexinf;i:codlex);
begin
  v:=v+1;
  with componente[v] do
    begin
      numeintrare:=i;
      numeinf:=e;
      inlantuire:=lista;
      lista:=v
    end
end;

procedure listavare;
var i:integer;
begin
  citestelve;
  for i:=0 to nrvar-1 do
    with stiva[virfstiva-i] do
      inlantuie(definf,defintrare);
  punetip;
  with informatii[intrare] do
    if (fel in felactiv)or(modcurent<>clasa2) then eroare(entryerr);
  listavar
end;
```

Fig. 6.7. Tratarea declarațiilor de variabile externe

utilizate în multiprogramare și nu se pune problema protecției împotriva unui acces simultan. În continuare, compilarea se execută în același mod ca cel indicat în subcapitolul 6.2., apelîndu-se procedura "listavar" (fig. 6.4.).

6.4. Compilarea declarațiilor de programe secvențiale

O facilitate importantă care trebuie implementată într-un program concurent este posibilitatea de a recunoaște și de a lansa programe secvențiale. În PASCAL CONCURENT, declararea și apelul programelor secvențiale se face după modelul procedurilor, similitudine care se păstrează și în compilator. Spre deosebire însă de declararea procedurilor, la declararea programelor secvențiale programatorul are posibilitatea să indice o listă de referințe externe ale programului secvențial, reprezentată de proceduri și funcții externe din procesul în care apare declararea programului și în care se va executa lansarea lui. Această listă este interfața programului secvențial cu cel concurent și trebuie să se regăsească în prefixul programului secvențial [BH75b]. Interfața crează posibilitatea apelării unor funcții concurente (de exemplu tratarea intrărilor/ieșirilor) din programul secvențial lansat [HR77].

Pentru fiecare identificator din lista de referințe externe, compilatorul apelează procedura "interfața" (fig. 6.8.). În cazul în care rutina indicată prin acest nume este definită, se verifică dacă este o rutină externă într-un tip sistem. Dacă rutina nu este definită, i se rezervă un rînd în tabela "informații" ("rezervăinf") și se completează rîndul corespunzător codului lexical ("cl") din tabela "intrări", după ce în prealabil s-a salvat conținutul lui anterior ("scintrări"). Se asociază acestui nume accesul "nerezolvat" și, în același timp, se contorizează rutina ca nerezolvată ("nerez"). La sfîrșitul compilării programului, "nerez" trebuie să fie 0. Acesta este singurul mod prin care un nume dintr-un program PASCAL CONCURENT poate fi

```
rocedure interfata;
ar cl:codlex;
   iinf:indexinf;
egin
 citeste(cl);
 if cl<>clndef then
   with intrari[cl] do
     if (acces<>nedefinit)and(nivel=nivelsistem) then
       if acces in accesext then inlantuie(intrareinf,cl)
         else eroare(intererr)
       else
         begin
           rezervainf(iinf);
           inlantuie(iinf,cl);
           scintrari(cl,iinf,nerezolvat);
           nerez:=nerez+1
         end
nd;
```

Fig. 6.8. Tratarea interfeței programelor secvențiale

eferit înainte de a fi definit.

Rutinele din interfață sînt înlănțuite în tabela "componente", într-un șir diferit de celea din care fac parte ca declarații externe. Acest șir este asociat programului ecvențial și va fi utilizat de compilator pentru crearea unei tabele de adrese a roceedurilor și funcțiilor referite. Tabela de adrese se crează în stiva de date din xecuție de către codul obiect generat la compilarea apelului programului secvențial. n această tabelă, rutinele se regăsesc în ordinea indicată în lista de referințe xterne. La execuția programului secvențial, identificarea unei anumite rutine se face unoscînd adresa de început a tabelei și numărul ei de ordine. Se permite astfel evenirea în procesul concurent pentru anumite funcții speciale.

```
type tipidef=0..bdefmax;
     tipiref=0..brefmax;
     tipetic=0..eticmax;
     tipadr=0..adrmax;
     tipref=record
              adrref:tipadr;
              eticref:tipetic
            end;
var indexref:tipiref;
    blocdef:array [tipidef] of tipadr;
    blocref:array [tipiref] of tipref;
    adrcod:tipadr;
```

Fig. 6.9. Structuri pentru înregistrarea adreselor nerezolvate

Intrucît rutinele din interfață pot fi referite înainte de a fi definite, în etapa generării codului obiect pentru crearea tabelei de adrese (faza 5) nu există certitudinea că se cunosc toate adresele de definire [AM77]. Din acest motiv s-a adoptat un mecanism simplu pentru transmiterea adreselor nerezolvate în faza 6 sub forma unui tablou denumit, în fig. 6.9., "blocref". Înregistrarea "adrref" reprezintă adresa din codul obiect generat la care a rămas deplasamentul nerezolvat, iar "eticref" memorează numărul rutinei referite.

Funcția "adrrut" (fig. 6.10.) prezintă principiul după care se crează o înregistrare în tabela de adrese a rutinelor din interfață, ținîndu-se cont de eventualele referiri

```
function adrrut(cl:codlex):tipadr;
begin
  adrrut:=1610612736;
  with intrari[cl] do
    with informatii[indexinf] do
      if categorie<>crutina then eroare(ruterr)
        else
          if acces=nerezolvat then
            if indexref>=irmax then abort
              else
                begin
                  indexref:=succ(indexref);
                  with blocref[indexref] do
                    begin
                      adrref:=adrcod+2;
                      eticref:=nrrutina
                    end
                end
            else
              if acces=extern then
                else eroare(acceserr)
end;
```

Fig. 6.10. Completarea unei înregistrări în tabela de adrese a rutinelor de interfață

"înainte". Adresele de definire ale rutinelor pentru care s-a generat anterior cod obiect, se găsesc în "blocdef".

"adrcod" este contorul codului obiect generat. Inițializarea adresei "adrrut" se face în scopul transformării ei în adresă de tip **BA**, bazabilă cu registrul 14 [BA74].

Capitolul 7

COMPILATORUL PASCAL CONCURENT PENTRU CALCULATORUL FELIX C

7.1. Structura şi funcţionarea compilatorului

Compilatorul PASCAL CONCURENT/FELIX C traduce programele sursă scrise în limbajul PASCAL CONCURENT, în format BT editabil de către editorul de legături al sistemului de operare SIRIS-3. Compilatorul efectuează mai multe tipuri de analize şi semnalează erorile de compilare corespunzătoare. Este împărţit în 6 faze. Fiecare fază reprezintă o singură trecere (baleiere) a programului sursă compilat sau a codului intermediar generat de faza anterioară şi generează codul intermediar pentru faza următoare. Compilatorul este scris în limbaj de asamblare (ASSIRIS). Structura lui segmentată este prezentată în fig. 7.1.

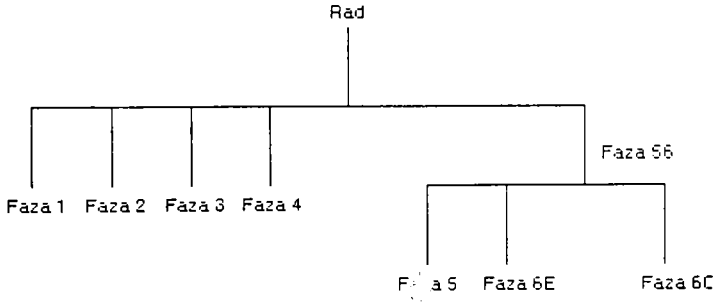


Fig. 7.1. Structura pe segmente şi faze a compilatorului pentru limbajul PASCAL CONCURENT (PASCAL)/FELIX C

Funcţiile compilatorului, pe faze, sînt următoarele [CI84b]:

Faza 1: *Analizorul lexical* - se baleiază textul sursă caracter cu caracter, se separă atomii textului convertind simbolurile, identificatorii, constantele numerice şi alfanumerice într-un şir de numere întregi care reprezintă codurile lexicale.

Faza 2: *Analizorul sintactic* - se verifică programul din punct de vedere sintactic, efectuînd o analiză de tip predictiv, de sus în jos, cu ajutorul unor proceduri recursive, corespunzătoare fiecărei variabile intermediare a gramaticii.

Faza 3: *Analizorul de domeniu* - se verifică dreptul de utilizare al variabilelor, tipurilor, constantelor şi rutinelor în punctul din program unde acestea sînt referite.

Faza 4: *Analizorul de declaraţii* - se verifică dacă variabilele sînt utilizate conform tipului declarat pentru ele; se calculează dimensiunea fiecărui tip utilizator şi deplasamentul variabilelor şi parametrilor în stiva de date.

Faza 5: *Analizorul de instrucţiuni* - se verifică corectitudinea utilizării instrucţiunilor se semnalează eventualele incompatibilităţi între entităţile dintr-o aceeaşi instrucţiune PASCAL şi se generează cod obiect pentru calculatorul FELIX C.

Faza 6: Este alcătuită din două părţi:

Faza 6E - se listează erorile de compilare;

Faza 6C - se rezolvă adresele rămase nerezolvate din faza 5, utilizînd în acest scop nişte tabele din segmentul de date faza 56 şi se organizează codul obiect generat de faza 5, în format BT.

In continuare sînt descrise aspectele mai importante din fazele 2-6 ale compilatorului. Nu se revine cu amănunte asupra fazei de analiză lexicală, aceasta fiind rezolvată după procedee clasice. De asemenea, analiza de domeniu a fost prezentată în principiu în capitolul 5 (§5.4.). Compilatorul PASCAL/FELIX C are aceeaşi structură şi, în linii mari, aceeaşi funcţionare.

7.2. Analizorul sintactic

Analiza sintactică din compilator este de tip predictiv [LE76] și se efectuează conform următorului procedeu: variabilelor intermediare ale gramaticii limbajului PASCAL CONCURENT, fixate pe baza descrierii în forma BNF, le corespunde cîte o rutină a analizorului sintactic. Această rutină tratează derivarea variabilei intermediare, în conformitate cu relația ei de transformare.

In această idee, analizorul sintactic [AU77] se compune dintr-un program principal și din subprogramele atașate fiecărei variabile intermediare; subprogramele se apelează între ele în conformitate cu relațiile de transformare ale gramaticii precum și în funcție de simbolurile din textul analizat. In cazul unei instrucțiuni succesiunea apelurilor dintre subprogramele analizorului sintactic constituie deci arborele sintactic al instrucțiunii analizate.

In scopul de a da claritate și o cît mai mare concizîune analizei sintactice, variabilele terminale ale gramaticii (deci simboluri putînd apărea în textul analizat) au fost grupate în clase: o clasă cuprinde acele simboluri care sînt de prevăzut a apărea ca simbol următor a textului de analizat în eventualitatea în care acesta este corect. Clasele nu sînt mulțimi disjuncte, un același simbol putînd apărea în mai multe dintre ele. Există 66 asemenea simboluri.

Utilizarea claselor se face în cursul analizei sintactice în felul următor: rutinele de analiză sintactică lucrează cu o zonă de memorie reprezentînd "clasa curentă". Această clasă curentă se îmbogățește cu una sau mai multe clase de la un apel de subrutină la altul și scade corespunzător în momentul revenirii. Dacă simbolul următor din textul sursă nu aparține clasei curente, s-a sesizat o eroare; în caz contrar, analiza continuă conform relațiilor de transformare ale gramaticii.

Clasa curentă a fost concepută ca o zonă de 66 octeți (număr egal cu cel al tuturor terminalelor gramaticii). Dacă simbolul este în clasa curentă, octetul corespunzător lui conține codul X'FF', iar în caz contrar codul X'00'. poziția simbolului în clasa curentă este dată de chiar valoarea sa, valorile acestor simboluri fiind cuprinse între 0 și 65. Printr-o asemenea structurare a clasei curente, adăugarea, respectiv scăderea de clase se poate face rapid.

Gramatica de descriere a limbajului PASCAL fiind recursivă, rutinele atașate variabilelor intermediare se apelează unele pe celelalte în mod recursiv; din acest motiv a fost necesar să se ia măsuri pentru a asigura reentranța acestor rutine.

Recursivitatea în cadrul gramaticii apare în principal datorită următoarelor posibilități ale limbajului:

a) Posibilitatea de a defini tipuri de variabile oricît de complicate prin definiții **type** succesive

b) Posibilitatea utilizării foarte libere a instrucțiunilor ciclice și condiționale (ex. **if** în **if**).

c) Recursivitatea din gramatica expresiilor aritmetice și logice.

Reentranța este asigurată prin aceea că datele proprii ale subprogramelor apelante; precum și adresa de revenire se salvează în momentul apelului la alt subprogram. Aceste operații se execută de o rutină de gestiune a apelurilor recursive, denumită RECURSIV. Avînd în vedere că o eventuală adăugare de clase la clasa curentă are loc în același moment, rutina RECURSIV execută și operațiile de completare a clasei curente [CI78b].

Apelul unui subprogram de analiză prin intermediul rutinei RECURSIV, cu transmiterea informațiilor necesare pentru efectuarea operațiilor menționate mai sus au loc prin următoarea secvență:

```
BAL,8       RECURSIV
DATA,4,4    BA'CL1', BA'CL2',..., BA'CLn'
DATA        L1
DATA,3      RA'DATE1'
```

```
.
DATA        Lm
DATA,3      RA'DATEm'
DATA,4,4    RA'SP1',X'C00nD00m'
```

unde CL1,...., CLn - sînt adresele claselor de adăugat clasei curente;

L1,...., Lm - sînt lungimile celor m zone de date de salvat;

DATE1,...., DATEm - sînt adresele zonelor de date de salvat;

SP1 - este adresa punctului de intrare în subprogramul apelat;

X'C00nD00m' - este un cuvînt indicînd numărul de clase adăugate (n), respectiv numărul de zone de date salvate (m), informații necesare pentru restabilirea configurației anterioare apelului, în momentul revenirii în rutina apelantă. Restabilirea este efectuată tot de subprogramul recursiv.

Zonele de lucru ale subprogramului recursiv sînt 4 stive plasate în zona de date comună a analizorului sintactic și anume [CI78b]:

- stiva adreselor de revenire;
- stiva de salvare a zonelor de date;
- două stive pentru a reține simbolurile adăugate clasei curente în momentul apelului.

Zona de date comună este organizată în două părți descrise ca secțiuni fictive:

1) Secțiunea constantelor analizorului - cuprinzînd valorile codurilor utilizate pentru reprezentarea programului analizat în fișierul de intrare al fazei și pe fișierul de ieșire, codurile erorilor sesizate în această fază, lungimea și conținutul fiecărei clase.

2) Secțiunea variabilelor analizorului - cuprinzînd stivele rutinei RECURSIV și alte variabile comune subprogramelor analizorului sintactic.

Forma intermediară generată în urma analizei sintactice a fost structurată în ideea că ea va fi utilizată de fazele ulterioare în mod diferit, și anume:

a) Forma intermediară codificînd definițiile și declarațiile de etichete, constante, variabile, tipuri și proceduri va fi tratată de fazele imediat următoare: analiza de domeniu și analiza declarațiilor.

b) Forma intermediară codificînd instrucțiunile programului, va fi tratată doar de faza de analiză a instrucțiunilor (faza 5).

Luînd în considerare aceste aspecte, codul intermediar generat de analizorul sintactic cuprinde, pe lîngă atributele atașate entităților programului și o serie de operatori, introduși în text în punctele în care vor acționa fazele următoare.

7.3. Analizorul de declarații

Analiza declarațiilor reprezintă cea de a patra fază a compilării unui program PASCAL CONCURENT. Funcția ei principală este aceea de a verifica utilizarea corectă a tipurilor standard precum și a celor definite în program, la declararea variabilelor și a parametrilor formali [AH86, CI79c]. In acest scop, analizorul utilizează o tabelă de informații despre entitățile programului, denumită "inf" (similară cu tabela "informații" din cap. 6). Tabela "inf" se completează la definirea sau declararea unei entități, cu date citite din fișierul de intrare, precum și cu cele deduse sau calculate de analizor. Aceste date sînt apoi disponibile pe tot parcursul compilării și, în funcție de ele, analizorul va decide corecta referire a entităților din program în momentul utilizării lor.

Diversitatea definițiilor și declarațiilor limbajului PASCAL CONCURENT a condus la diversificarea adecvată a organizării liniilor curente din tabela "inf", după cum urmează [CI80d]:

a) un fel de organizare pentru cazul în care informațiile memorate se referă la cîmpuri variabile sau parametri formali;

b) un fel de organizare pentru cazul cînd entitatea declarată este o rutină;

c) un fel de organizare pentru identificatorii de tipuri.

In tabela "inf" se reţin informaţii referitoare la tipurile şi rutinele standard cît şi la parametrii acestor rutine. Aceste linii sînt completate la iniţializarea analizei şi sînt modificate doar cînd analizorul şi-a redefinit anumite cuvinte standard după propriile sale nevoi. Din acest motiv, în cursul analizei nu se face distincţie între entităţile standard şi cele definite sau declarate de utilizator. Analiza declaraţiilor se realizează în principal prin intermediul tabelei "inf" şi a unei stive de lucru "s". Stiva "s" conţine la un moment dat consrucţia PASCAL CONCURENT analizată sub forma unui şir de indici în tabela "inf"; ea se reduce pe parcurs, astfel încît la încheierea analizei unei construcţii independente stiva este goală.

Definirea tipurilor şi constantelor şi declararea variabilelor formează partea de descriere a unui program. Începînd cu faza a patra a compilării, ea dispare din codul intermediar. Informaţia referitoare la tipul unei variabile este înscrisă în punctele din codul intermediar corespunzător instrucţiunilor propriu zise, în care variabila respectivă este utilizată (referită).

Subprogramele de analiză a definiţiilor de tipuri îndeplinesc următoarele funcţii:

a) Calculul dimensiunii fiecărui tip definit în program, astfel:

- se acordă tipurilor "enumerare" şi "subdomeniu" dimensiunea standard de un cuvînt (4 octeţi);
- se acordă tipului "mulţime" dimensiunea standard de 32 octeţi, ceea ce limitează cardinalitatea unei variabile de acest tip la 256;
- se acordă tipului "tablou" dimensiunea calculată după formula:

$$D_T = D * \prod_{k=1}^{n} (M_K - m_K + 1)$$

unde: M_K, m_K sînt limitele, superioare şi respectiv inferioare ale indicelui tabloului;

D este dimensiunea unui element; în cazul general D poate rezulta dintr-un calcul de dimensiune anterior efectuat pentru tipul elementului de tablou;

- se acordă tipului "articol" dimensiunea:

$$D_A = \sum_{k=1}^{n} D_K$$

unde D_K este dimensiunea tipului cîmpului K din articol.

In cazul special al limbajului PASCAL CONCURENT, pentru calculul dimensiunii tipurilor, "proces", "monitor" şi "clasă" s-a ţinut cont de faptul că procesele se execută în paralel şi, odată lansate, se eliberează complet de procesul care a executat lansarea lor. Din acest motiv, rezervarea de memorie pentru variabilele de tip "proces" nu se face în zona de variabile a procesului iniţial (unde sînt declarate) ci într-o zonă complet separată şi protejată împotriva accesului dintr-un alt proces. Pentru uniformitate, se acordă totuşi tipului "proces" dimensiunea fixă de 4 octeţi care urmează a se rezerva în locul declarării variabilei "proces".

Monitoarele şi clasele nu au calitatea de mai sus. Dimensiunea unui tip "monitor" sau "clasă", care ar fi asociată fiecărei variabile de acest tip se calculează astfel:

$$D = D_V + D_P + D_C$$

unde: D_V este lungimea zonei de variabile a monitorului sau clasei (în octeţi);

D_P este lungimea listei de parametri a monitorului sau clasei (în octeţi);

D_C=4 octeţi necesari doar în cazul monitoarelor pentru memorarea adresei porţii (cap. 8.).

b) Semnalarea erorilor care rezultă din evaluarea dimensiunilor tipurilor.

c) Calculul deplasamentelor cîmpurilor în cadrul articolelor, ţinînd cont de dimensiunea cîmpurilor anterioare.

d) Completarea tabelei "inf" cu atributele citite, deduse sau calculate pe parcursul analizei, referitoare la fiecare identificator de tip. Tratarea parametrilor formali constă din următoarele operaţii [HR77]:

a) Stabilirea categoriei din care face parte parametrul formal, în funcţie şi de

tipul său.

b) Verificarea restricțiilor impuse la declararea parametrilor formali de tip "proces", "monitor", "clasă" și "locație de așteptare" [CI79e].

c) Calcularea deplasamentelor parametrilor formali în cadrul listei de parametri din care fac parte, ținînd cont de dimensiunea parametrilor anteriori.

d) Calcularea dimensiunii totale a listei de parametri.

e) Completarea în tabela "inf" a liniilor corespunzătoare parametrilor formali și rutinelor propriu-zise.

Analiza declarării rutinelor trebuie să țină cont de posibilitatea referirii "înainte" a procedurilor și funcțiilor declarate cu **forward** (PASCAL) sau a celor externe (**entry**) din procese PASCAL CONCURENT [CI84b]. În aceste cazuri, rezervarea liniei din "inf" corespunzătoare rutinei se face la declararea parțială (**forward**) sau la referire (**entry**), iar completarea propriu-zisă, la declararea ei definitivă. Erorile ce mai pot să apară în această etapă a analizei sînt doar cele care rezultă din utilizarea eronată a tipului funcției.

Fiecare rutină primește un număr de ordine (etichetă), funcție de poziția sa în textul sursă. Acest număr va fi asociat numărului de instrucțiuni corespunzător rutinei și va identifica rutina în fazele de compilare următoare. În final, referirea rutinei prin eticheta ei se va înlocui cu saltul cu revenire (BAL,6) la adresa de început a blocului de instrucțiuni [CI80d].

La încheierea tratării listei de parametri se cunoaște și se completează în tabela "inf" dimensiunea zonei de parametri a rutinei, necesară în timpul execuției programului pentru reducerea corespunzătoare a stivei de date, la sfîrșitul apelului rutinei precum și pentru a avea acces la informațiile aflate în stivă imediat sub zona de parametri.

Intîlnirea cuvîntului cheie **begin** corespunzător blocului rutinei coincide cu sfîrșitul tratării variabilelor. Dimensiunea variabilelor va fi de asemenea memorată în "inf" și se va transmite și în fișierul de ieșire la fiecare apel, fiind utilizată tot în execuție pentru rezervarea zonei de stivă necesară.

Subprogramele de tratare a declarațiilor de variabile îndeplinesc următoarele funcții:

a) verifică concordanța dintre tipul variabilei și locul declarării ei;

b) stabilesc dimensiunea variablei respective pe baza dimensiunii tipului corespunzător;

c) calculează dimensiunea variabilelor în stiva de variabile, ținînd cont de dimensiunile variabilelor anterioare;

d) gestionează dimensiunea totală a zonei de variabile pentru un bloc;

e) completează în tabela "inf" linia care corespunde variabilei cu atributele citite, deduse sau calculate.

În concluzie, partea din faza 4 de tratare a tipurilor și declarațiilor de variabile nu generează cod intermediar pentru faza următoare. Subprogramele de analiza blocului de instrucțiuni, utilizînd tabela "inf", scriu în fișierul de ieșire, la referirea entităților respective toate informațiile necesare pentru efectuarea analizei semantice și pentru generarea codului obiect. Deci, caracteristicile unei variabile sînt multipilicate în fișierul de ieșire de un număr de ori egal cu numărul de referiri făcute la variabila respectivă.

7.4. Analizorul semantic

Funcția de bază a analizorului semantic este aceea de a verifica și semnala incompatibilitățile care pot apărea între entitățile din cadrul aceleiași instrucțiuni [CI80d]. Astfel sînt analizate: compatibilitatea parametrilor formali cu cei actuali la apelul rutinelor, compatibilitate între operanzi și operatori în expresiile limbajului, etc. Deoarece instrucțiunile se prezintă analizorului semantic codificate în formă poloneză extinsă [CI78b], s-a adoptat o metodă de analiză ce utilizează o stivă: în

fiecare rînd al stivei se introduc informațiile refritoare la un operand (constante, variabile, parametri formali și actuali, monitoare, procese, clase, etichete, etc.), conținute în fișierul de intrare al fazei și provenind din fazele anterioare de analiză [CI79c].

Structura unui rînd al stivei depinde de tipul operandului. Elementele între care au loc verificări ale compatibilității se găsesc întotdeauna în vîrful stivei.

In tratarea instrucțiunilor analizorul execută următoarele operații [CI80d]:

a) verifică compatibilitatea operanzilor în cadrul aceleiași expresii;

b) verifică compatibilitatea membrului stîng cu membrul drept al unei asignări;

c) verifică concordanța între operatori și tipul operației;

d) determină tipul operanzilor (globali, locali, constante lungi sau scurte), pentru ca ulterior să se cunoască cu ce registre de bază este adresabilă entitatea;

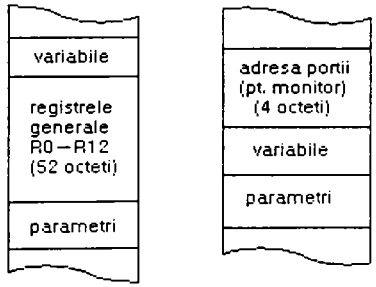
e) verifică compatibilitatea dintre parametri actuali și cei formali.

In cursul analizei semantice și înainte de generarea codului obiect se calculează majoritatea adreselor entităților din program. Acest calcul se face ținînd cont de structura programului obiect, și anume:

- cod obiect executabil;
- zonă conținînd valorile constantelor programului;
- o stivă de date a programului.

Entitățile programului sînt grupate funcție de modalitatea prin care se realizează adresarea în codul obiect, în următoarele categorii:

- *constantele scurte* (întregi, caractere, booleene) apar prin însăși valoarea lor și sînt accesibile prin instrucțiuni imediate;
- *constantele lungi* (șiruri de caractere, numere reale) sînt concentrate la sfîrșitul codului obiect, iar accesul la ele este asigurat prin bazare cu registrul 9;
- *parametrii și variabilele* se memorează în stiva de date, în zone diferite (fig. 7.2.), după cum sînt de tip global sau local și în funcție de dinamica stivei în momentul apelului [CI80d]. Accesul la aceste zone se face prin bazare cu registrele 12, respectiv 11.



a) variabile locale b) variabile globale

Fig. 7.2. Localizarea variabilelor și parametrilor în stiva de date

Organizarea prezentată determină corectarea deplasamentelor calculate în faza 4, după cum urmează:

a) pentru entitățile locale se ține cont de cei 52 octeți reținuți pentru salvarea registrelor generale la trecerea în nivelul local;

b) pentru entitățile globale se ține cont de cei 4 octeți reținuți în vederea memorării adresei porții monitoarelor [CC81].

Un caz aparte îl constituie calculul adresei cîmpurilor și variabilelor cu indici, datorită existenței a două deplasamente: deplasamentul articolului sau tabloului în stiva de date și deplasamentul cîmpului în cadrul articolului respectiv deplasamentul elementului de tablou în zona tablou. Pentru accesul la astfel de entități se

utilizează facilitatea de indexare. Cel de-al doilea deplasament este calculat și plasat în vîrful stivei. Mecanismul implementat pentru calculul adresei rezolvă și cazuri mai complicate cum ar fi: articol în articol, tablou de articole, cîmp de tip tablou, etc.

7.5. Generatorul de cod obiect

Generarea codului obiect se desfășoară în paralel cu efectuarea verificărilor semantice și constă în generarea instrucțiunilor cod-mașină pentru calculatorul FELIX C-256. In această etapă rămîn nerezolvate doar adresele entităților referite înainte de a fi definite (etichete, rutine). Informațiile necesare rezolvării acestor adrese se transmit în faza 6 prin segmentul de date comune Faza56 (fig. 7.1.). In continuare se prezintă cîteva aspecte specifice generării codului obiect pentru programe PASCAL [CI80d]. Astfel:

a) Tratarea apelurilor de procedură sau funcție comportă următoarele operații:
- generarea depunerii parametrilor actuali în stiva de date;
- generarea operației de apel;
- memorarea adresei de apel ca nerezolvată în cazul în care referirea rutinei se face "înainte".

Apelul funcțiilor și procedurilor standard este realizat prin generarea apelului subprogramului corespunzător din BSS [CI75].

b) Instrucțiunea **case** se implementează sub forma unei tabele de salturi la instrucțiunea sau setul de instrucțiuni corespunzător fiecărei mărci. Restul instrucțiunilor condiționale sînt despărțite în etichete și salturi încă din faza 2, ceea ce permite tratarea lor uniformă.

c) Referitor la complexitatea generării codului obiect pentru expresii se amintesc posibilitățile limbajului PASCAL CONCURENT de a opera asupra mulțimilor (tipul **set**) și de a compara și asigura structuri printr-o singură instrucțiune sursă.

7.6. Listarea erorilor de compilare și organizarea codului obiect în format BT

Listarea erorilor de compilare se realizează în segmentul Faza6E. In acest scop se utilizează tabela "taberr" din segmentul de date Faza56 (fig. 7.1.) în care analizorul semantic a selectat și a introdus erorile de compilare semnalate în cele 5 faze de analiză. Selectarea erorilor înseamnă reținerea în "taberr" a unei singure erori pentru un rînd sursă întrucît, cu o mare probabilitate, prezența mai multor erori pe același rînd se datorește aceleiași erori din textul sursă.

In funcție de faza în care a apărut eroarea și în funcție de codul erorii, se selectează și se tipărește textul adecvat acelei erori.

In cazul în care programul conține erori de compilare, se interzice trecerea la faza de prelucrare următoare (editarea legăturilor).

Absența erorilor de compilare determină apelul segmentului Faza6C, care rezolvă adresele nerezolvate din codul obiect și îl organizează în format BT.

In vederea rezolvării adreselor de rutine referite înainte de a fi definite se utilizează tablourile "bloctab" și "blocref" furnizate de generatorul de cod prin intermediul segmentului de date Faza56 (fig. 7.3.).

"bloctab" este o tabelă avînd intrarea de 2 octeți în care se înscriu, pe parcursul generării codului obiect adresele de apel ale rutinelor. Acestea sînt apoi accesibile folosind ca indice valoarea 2*(nrbloc-1), unde "nrbloc" este numărul de ordine al rutinei.

Tabela "blocref" are intrarea de 4 octeți, primii 2 conținînd adresa la care s-a făcut referirea la o procedură sau funcție, iar ultimii 2 conțin numărul de ordine al blocului referit. Completarea tabelei "blocref" se face în ordinea întîlnirii apelurilor.

Prin baleierea tabelei "blocref" și utilizînd tabela "bloctab" se pot rezolva toate adresele de rutine rămase nerezolvate în faza anterioară.

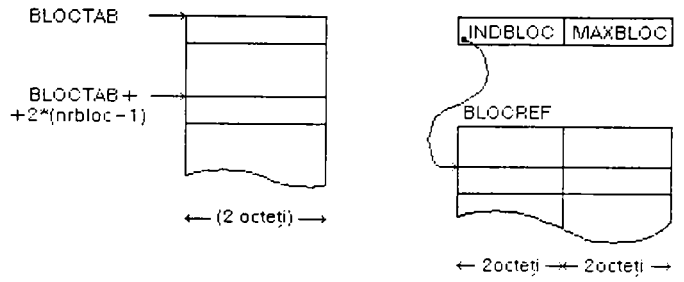


Fig. 7.3. Structuri de date pentru rezolvarea apelurilor de rutine rămase nerezolvate la generarea de cod

Pentru completarea adreselor de salt înainte se utilizează tabelele "salttab" și "saltref" cu conținut și semnificație asemănătoare tabelelor "bloctab" și "blocref" (fig. 7.4.).

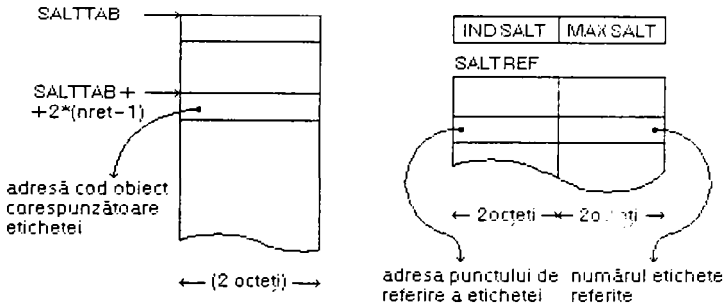


Fig. 7.4. Structuri de date pentru rezolvarea salturilor rămase nerezolvate la generarea de cod

Organizarea codului obiect în format BT presupune generarea articolelor de tip STB, REF, DEF, TXT, EOB precum și completarea fișierelor sistem FILEDIT (*5) și REPEDIT (*6) în conformitate cu cerințele editorului de legături [CI75]

7.7. Sistemul de compilare interactivă a programelor PASCAL și PASCAL CONCURENT

Compilatorul pentru limbajul PASCAL CONCURENT prezentat în paragrafele anterioare ca și cel pentru limbajul PASCAL/FELIX C de altfel, au fost proiectate pentru regimul de lucru în loturi de programe. In perioada 1982-1984, pe baza celor două compilatoare disponibile, la CCE al IPTVT s-a realizat un sistem pentru compilarea și punerea la punct în regim conversațional a programelor PASCAL și PASCAL CONCURENT [CE82, CI82].

Compilatoarele existente sînt reentrante, ceea ce a facilitat în mare măsură realizarea versiunilor conversaționale. Obiectivele principale care s-au luat în considerare la proiectarea sistemului sînt:

- executarea de compilări, în paralel, de la mai multe terminale;
- semnalarea erorilor de compilare la terminale cu posibilitatea, în anumite cazuri. a corectărilor imediate în vederea continuării compilării;
- introducerea și corectarea interactivă a programelor sursă.

Cele două compilatoare se comportă în mod identic din punct de vedere al problemelor tratate în continuare. Forma și funcționarea lor au impus anumite

limitări în adoptarea soluțiilor pentru realizarea sistemului conversațional.

7.7.1. Structura unui sistem de compilare cu monoacces

Pornim de la cerințele exprimate anterior sistemul, privit din punct de vedere al unui singur utilizator, are structura prezentată în fig. 7.5.

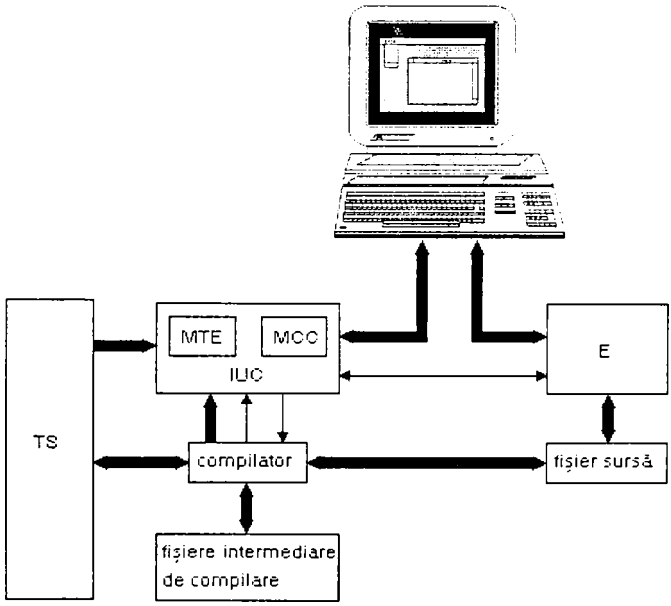


Fig. 7.5. Structura sistemului cu monoacces

Utilizatorului de la terminal i se oferă servicii de editare a programelor prin dialog direct cu editorul (E) și de compilare prin intermediul interfeței utilizator-compilator (IUC), care cuprinde următoarele două module:

- modulul de tratare erori (MTE), care îndeplinește sarcinile legate de cercetarea interactivă a erorilor de compilare;
- modulul de control al compilării (MCC), care inițiază la cerere compilările, ține evidența fazelor de compilare și asigură reluările după tratarea unei erori.

7.7.2. Testarea erorilor de compilare

Eficiența utilizării unui limbaj de programare este strîns legată de asistența pe care o acordă compilatorul pentru detectarea și corectarea erorilor [AU77]. In cele ce urmează, această problemă va fi privită strict din punctul de vedere al proiectării sistemului de compilare interactivă, neurmărindu-se în mod special tratarea erorilor pornind de la clasificarea lor după tipul de analiză (lexicală, sintactică, semantică, etc.).

In compilatoarele existente funcționează moduri de depistare a erorilor și politici de reluare a compilării specifice fiecărei analize [CI80c]. Erorile se codifică și se cumulează în fișierele intermediare urmînd a fi semnalate programatorului (listate la imprimantă) la încheierea compilării.

Pentru abordarea tratării erorilor în mod conversațional s-au reținut ca puncte de plecare următoarele aspecte:

- informarea programatorului la depistarea fiecărei erori prin furnizarea unui

mesaj clar și complet;

- oferirea tuturor variantelor de remediere posibile pentru erorile corectabile în locul în care au apărut, cu continuarea compilării din acel punct, dacă s-a acceptat corecția;

- introducerea posibilității ca programatorul să decidă oprirea compilării sau continuarea ei, în cazul în care eroarea nu este corectabilă pe loc.

In vederea realizării dezideratelor de mai sus s-au avut în vedere o serie de restricții, impuse fie de natura limbajelor în discuție, fie de modul de realizare al compilatoarelor existente.

Structura pe blocuri a programelor PASCAL și PASCAL CONCURENT conduce la extinderea efectului unor corecții aparent locale, în sus în program, dar nu în afara blocului respectiv. Alte categorii de erori (ex. identificator nedefinit), implică modificări în părți de program compilate anterior care nu sînt în mod obligatoriu în blocuri în care a apărut eroarea. Din punctul de vedere al corecțiilor posibile s-au luat în considerare următoarele categorii de erori:

a) - erori corectabile imediat (în locuri în care au apărut) cu continuarea compilării din același punct;

b) - erori corectabile imediat și reluarea compilării de la începutul blocului;

c) - erori pentru care se impune corectarea ultimei construcții sintactice și reluarea compilării cu această construcție;

d) - erori corectabile în altă parte din program și reluarea compilării de la începutul blocului în care s-a făcut corecția (în caz particular de la începutul programului).

Asupra acestei clasificări au acționat limitările impuse de realizarea unor modificări minime în compilatoarele existente. Astfel, organizarea analizelor în 6 faze separate, deși are avantajul posibilității de a relua compilarea cu începutul fazei în curs, ridică dificultatea de a pune în concordanță corecțiile în fișierele intermediare cu cele în fișierul sursă, dat de programator. Aceasta implică reluarea compilării de la început în anumite cazuri, pornind de la noul fișier sursă, corectat separat de programator. Analiza sintactică realizată (de sus în jos cu descendenți recursivi) pornește de la sintaxa originală a limbajelor și nu de la una transformată în ideea depanării interactive a programelor [SE80]. Din această cauză s-a renunțat la a oferii corectarea pe loc a ultimei construcții sintactice (cat. c).

Eficiența depanării programelor depinde de claritatea mesajelor furnizate, ce trebuie să includă referiri la simbolurile care au determinat eroarea, simboluri ce pot fi corectate de programator. Acest deziderat conduce la transmiterea tablei de simboluri TS (fig. 7.5.) între faze și la organizarea ei în așa manieră încît să se realizeze accesul atît pe baza codului lexical (furnizarea simbolului), cît și pe baza numelui simbolului (furnizarea codului lexical). Considerente de economie de memorie au determinat reținerea TS într-un fișier cu acces direct.

Dificultatea repunerii unor faze de compilare pe starea corespunzătoare începutului unui bloc a condus la renunțarea reluării compilării cu începutul blocului respectiv (se va relua, după caz, faza de compilare sau compilarea de început a programului).

Din cele de mai sus rezultă următorul mod de tratare a erorilor:

i) Compilatorul detectează eroarea și o transmite modulului de tratare a erorilor.

ii) Modulul de tratare intră în dialog cu programatorul afișînd eroarea simbolic (utilizînd TS). In funcție de natura și gravitatea erorii se oferă alternative de corectare ori posibilitatea optării între întreruperea sau continuarea compilării.

iii) Corecțiile imediate pot fi realizate și în fișierul sursă, separat, prin intermediul Editorului. In cazul corecțiilor acceptate, compilarea se reia din punctul respectiv sau de la începutul programului. Această din urmă reluare se poate face după natura erorii, fie cu începutul fazei în curs, fie cu începutul compilării.

iiii) In celelalte cazuri se continuă compilarea sau nu, în funcție de decizia programatorului.

7.7.3. Editarea programelor sursă.

Editorul de text asigură toate operațiile legate de introducerea și modificarea programelor sursă, prin intermediul următoarelor funcții principale:

- introducerea de programe sursă de la terminal; crearea unui nou fișier sursă pornind de la mai multe fișiere existente;
- consultarea unui fișier sursă;
- modificarea unui fișier sursă, independent de compilare, la cererea programatorului;
- corectarea la cerere a unui fișier, în urma sesizării unei erori de compilare.

Modificările în programe se realizează la nivel de linie sursă, sau de caracter în cadrul unei linii. Avînd în vedere orientarea spre limbajele PASCAL și PASCAL CONCURENT a sistemului, se au în vedere și funcții de editare orientate pe sintaxa limbajelor. In acest caz editorul tratează unități reprezentînd construcții gramatical PASCAL sau PASCAL CONCURENT.

7.7.4. Extinderea sistemului pentru regimul de lucru cu multiacces.

Pentru a extinde sistemul prezentat, astfel încît el să deservească în paralel mai mulți utilizatori, la sistemul din fig. 7.5 s-a adăugat un program supervizor. Acesta asigură evoluția corectă a acțiunilor paralele de editare și compilare, îndeplinind următoarele funcții:

- Inițializarea sesiunii de lucru a fiecărui utilizator, ceea ce presupune luarea în evidența și alocarea de memorie. In continuare, acțiunile de compilare sau editare inițiale de la terminale se tratează la nivelul supervizorului ca activități (procese) paralele;
- Gestionarea unității centrale (UC) prin divizare de timp între procese. Astfel, fiecărui proces i se alocă pe rînd UC pentru o anumită cuantă de timp, procesele trecînd prin stări succesive de activitate și de așteptare;
- Evidența stărilor momentane ale proceselor. Un proces execută la un moment dat compilări sau editări. O compilare se desfășoară într-o anumită fază și decurge normal sau se află în stare de testare interactivă a unei erori. Procesul aflat în oricare din aceste stări poate fi în curs de execuție sau în așteptarea alocării UC;
- Gestionarea și protecția resurselor sistemului (fișiere de lucru, fișiere sursă, zone de date);
- Scoaterea din evidență a procesului și eliberarea resurselor asociate acestuia la încheierea unei sesiuni de lucru la un terminal;
- Impreună cu modul de control al compilării (fig.7.5.) supervizorul asigură sincronizarea activităților de compilare din procese paralele, după o politică realizată pe baza următoarelor considerente.

Pentru a limita necesarul de memorie a sistemului s-a luat decizia ca la un moment dat în memoria centrală, alături de editor și modulele sistemului să fie prezentă o singură fază a compilatorului.

In scopul evitării unui trafic exagerat memorie-centrală disc al segmentelor compilatorului, compilările paralele se vor executa astfel: fazele de compilare se activează în ordine, trecerea de la una la cealaltă făcîndu-se în momentul în care nici o activitate paralelă nu mai solicită compilări în faza respectivă. In scopul evitării unor așteptări îndelungate procesele care în urma detectării unei erori au trecut la corectări, ies din competiția pentru compilare în ciclul respectiv, urmînd ca să fie servite la prima trecere, ulterioară încheierii corecturilor, prin faza din care se reia compilarea.

Capitolul 8

NUCLEUL LIMBAJULUI PASCAL CONCURENT/FELIX C. DETALII DE REALIZARE

8.1. Structura şi funcţiile nucleului

Implementarea limbajului PASCAL CONCURENT pe calculatoarele din gama FELIX C a implicat proiectarea a două părţi distincte şi anume:

a) compilatorul;

b) nucleul limbajului.

Intrucît compilatorul a constituit obiectul unor capitole anterioare (cap. 5, §5.4.; cap. 6; cap. 7) în acest capitol se vor face cîteva precizări privind tehnicile pe baza cărora a fost realizat nucleul [HR77, CI79e, CI84a].

Nucleul limbajului PASCAL CONCURENT este o colecţie de subprograme, gîndită în mod unitar, care are ca scop controlul şi sincronizarea proceselor, rezolvarea intrărilor/ieşirilor precum şi alte funcţii, unele de natură concurentă, altele secvenţiale, care nu au fost tratate direct în codul obiect. Partea de nucleu necesară unui anumit program PASCAL CONCURENT, se ataşează acestuia în faza de editare a legăturilor pe baza referinţelor nerezolvate ale programului compilat [CE83a].

Parametrii de apel ai subprogramelor de nucleu se transmit prin codul obiect generat, imediat după instrucţiunea de apel corespunzătoare.

Modularitatea sub care se prezintă nucleul, face ca eventuala schimbare a politicii de gestiune a proceselor sau resurselor să nu afecteze decît o mică parte a lui, după cum nu va afecta nici compilatorul propriu-zis. In acest mod, cu efort minim, se pot implementa diferite tehnici de tratare a competiţiei la resurse, de gestiune a şirului de aşteptare, etc., ţinînd cont de categoria de aplicaţii căreia îi este destinat limbajul.

Subprogramele de nucleu îndeplinesc următoarele funcţii principale [CI80b]:

- gestionează o parte din resursele proprii sistemului de calcul (memorie, UC);
- gestionează şi protejează resursele descrise în program şi modelate prin monitoare;
- asigură multiprogramarea proceselor în cadrul aceleiaşi partiţii;
- gestionează şirurile de aşteptare interne şi, parţial, pe cele ale utilizatorului (constitute cu locaţii de tip **queue**).

In continuare, nu se face o trecere în revistă a subprogramelor de nucleu, ci a principiilor care au stat la baza elaborării lor, principii care implementează noţiunile de proces, monitor şi clasă aşa cum au fost definite de P. B. Hansen [BH75c]. Problemele specifice gestiunii timpului real au fost tratate în cap. 4 şi nu se revine asupra lor.

8.2. Evidenţa proceselor

Aşa după cum s-a arătat (cap. 3, §3.2.2.), în limbajul PASCAL CONCURENT, tipul proces descrie un set de date proprii, acţiuni asupra acestor date grupate, eventual, în subprograme şi un număr de drepturi de acces la alte componenete ale programului. El nu poate opera asupra datelor proprii altui proces, dar, mai multe procese, pot utiliza structuri de date şi resurse comune, prin intermediul monitoarelor.

Variabilele de tip **process** sînt taskuri ce rulează în paralel, disputîndu-şi între ele resursele sistemului de calcul şi ale programului concurent.

In mecanismul de gestiune implementat, procesul este implementat prin tranzacţia sa (fig. 8.1.) înglobează caracteristici momentane dintre cele mai diverse ale procesului respectiv. Semnificaţia cîmpurilor conţinute este următoarea:

- *adrleg*: adresa de legătură a procesului în diferite şiruri de aşteptare.
- *index*: numărul de ordine al procesului. Se acordă în ordinea iniţializării

```
type ptranz=^tiptranz;
     manevra=array [1..2] of integer;
     tipsalv=array [0..14] of integer;
     tipindex=0..procmax,
     tiprez=(corect,incorect,intrerupt);
     tiptranz=record
                adrleg:ptranz;
                index:tipindex;
                apelmonit:integer;
                pd,pdmax:integer;
                timpuctotal,timpuc,timpabs:integer;
                prioritate:integer;
                rezultat:tiprez;
                zonasalv:tipsalv;
                adrintrerupere:integer;
                zonamanevra:manevra
              end;
var  tranzact:ptranz;
```

Fig. 8.1. Structura tranzacției unui proces

(lansării) lor; procesul inițial are "index"=1.

- *apelmonit*: are la început valoarea zero. Se incrementează cu unu la fiecare apel al unei rutine externe de monitor și se decrementează la revenire. Indică numărul de monitoare apelate la un moment dat, de procesul respectiv.

- *pd*: pointerul de alocare dinamică. Reprezintă, în fiecare moment, adresa de memorie de la care se va face următoarea alocare dinamică (prin apelul procedurii "new") pentru programele PASCAL (secvențiale) apelate sau lansate din procesul respectiv [CE83a, CI81c, CE85a]. După apelul procedurii "new", "pd" crește cu lungimea zonei alocate.

- *pdmax*: valoarea maximă pe care o poate lua "pd". Pe parcursul executării procesului concurent, se modifică între anumite limite. În momentul în care "pd">"pdmax", programul secvențial (apelat sau lansat) sau concurent, în activitate se termină cu depășire de memorie.

- *timpuctotal*: cumulează timpul UC consumat de un proces de la inițializarea sa și pînă în momentul respectiv. Este număr întreg exprimînd unități de 512 μs.

- *timpuc*: timpul unitate centrală consumat de proces de la ultima acordare a unei cuante de 0.125 secunde.

- *timpabs*: zona de manevră pentru calcule de timp;

- *prioritate*: prioritatea procesului (-1,0,1,2)

- *rezultat*: starea în care s-a terminat programul secvențial apelat sau lansat de procesul respectiv;

- *zonasalv*: zonă pentru salvarea registrelor generale (R0 - R14), la întreruperea procesului;

- *adrintrerupere*: adresa din codul obiect la care procesul a fost întrerupt ultima dată.

Limbajul PASCAL CONCURENT incorporează două subprograme standard care permit accesul programatorului la tranzacția unui proces, și anume:

1) *Funcția attribute(x)* este o funcție de tip întreg; furnizează valoarea cîmpului din tranzacția procesului care a executat apelul (activ), aflat la deplasamentul "x" (întreg) față de începutul ei. Prin intermediul acestei funcții, toate cîmpurile din tranzacție pot să fie consultate, fără însă a fi modificate.

2) *Procedura setheap(x)* forțează "pd" pe valoarea transmisă în argumentul "x" (întreg). Împreună cu attribute(4), este utilă pentru eliberarea unor zone alocate dinamic de către un program PASCAL (secvențial) apelat sau lansat, în momentul în

care acesta şi-a încheiat activitatea.

8.3. Activarea şi dezactivarea proceselor. Priorităţi.

Printr-un program PASCAL CONCURENT se indică regulile conform cărora procesele concurente coexistă în sistemul de calcul. In acest scop stau la dispoziţie tehnicile sincronizării automate şi programate prezentate în [CE85a]. In limitele permise de sincronizarea realizată în program, acţionează un set de operaţii de sincronizare, invizibile pentru programator şi independente de el. Acestea au ca scop realizarea unei multiprogramări cît mai eficiente a proceselor pe setul de procesoare fizice disponibile pe baza următoarelor principii:

- creşterea gradului de paralelism în funcţionarea procesoarelor (pe calculatorul FELIX C se urmăreşte desfăşurarea paralelă a operaţiilor de I/E din anumite procese concomitent cu executarea instrucţiunilor în unitatea centrală pentru alte procese);
- asigurarea accesului la unitatea centrală într-un anumit interval de timp a tuturor proceselor disponibile.

In continuare se prezintă principalele elemente pe baza cărora se realizează gestionarea proceselor programului concurent.

a) Stările în care se poate găsi un proces la un moment dat sînt următoarele:

1) *Activ* (running) - atunci cînd unitatea centrală execută acţiuni proprii procesului respectiv (inclusiv rutinele din alte tipuri sistem, apelate).

2) *Pregătit pentru activare* (ready) - în cazul în care procesul aşteaptă pentru a primi unitatea centrală, ocupată de un proces de prioritate mai mare sau de aceeaşi prioritate dar lansat anterior.

3) *In aşteptare* (waiting) - atunci cînd procesul aşteaptă producerea unui eveniment exterior lui; stările de aşteptare în care poate ajunge un proces se împart în mai multe categorii, astfel:

- aşteptare la un monitor ocupat (prin apelul unei rutine externe dintr-un monitor ocupat);
- aşteptare într-o locaţie de tip **queue** (prin "delay");
- aşteptare a întreruperii de ceas (prin "wait");
- aşteptare a întreruperii de la consola centrală (prin "io");
- aşteptare pentru terminarea unei operaţii de I/E (prin lansarea unei astfel de operaţii).

In momentul producerii evenimentului pentru care un proces este în aşteptare el va deveni activ, dacă nu există un alt proces activ sau pregătit pentru activare cu prioritate mai mare sau egală cu a lui. In caz contrar va trece în starea "pregătit pentru activare".

b) Pe parcursul rulării unui program PASCAL CONCURENT procesele sînt caracterizate printr-un număr de prioritate acordat automat după cum urmează [CI84a]:

-1: din momentul apelării procedurii "wait" şi pînă la reactivarea procesului;

0: cînd procesul execută acţiuni dintr-un monitor sau dintr-o clasă activată direct sau indirect dintr-un monitor;

1: dacă procesul a terminat o operaţie de I/E şi nu sînt îndeplinite condiţiile de la prioritatea 0;

2: la lansarea procesului, la revenirea dintr-o rutină externă de monitor (dacă nu mai sînt îndeplinite condiţiile de la prioritatea 0), la depăşirea unei cuante de timp admise (consumată în unitatea centrală).

Prioritatea cea mai înaltă este -1. Urmează în ordine 0, 1, 2. Un proces mai puţin prioritar poate fi întrerupt pentru a lansa altul mai prioritar.

c) La activarea sa, unui proces i se acordă o cuantă de timp UC de aproximativ 0.125 secunde. In momentul depăşirii acestei cuante procesul primeşte prioritatea 2 şi poate fi întrerupt pentru a lansa un alt proces "pregătit pentru activare"

(divizarea timpului între procese).

In concluzie, un proces activ rămîne în această stare pînă la producerea unuia din următoarele evenimente:

- depășirea cuantei de timp acordate;
- trecerea într-una din stările de așteptare;
- întreruperea de către un proces mai prioritar care se găsește în starea "pregătit pentru activare".

In momentul întreruperii unui proces, va fi activat procesul cu prioritatea și vechimea cea mai mare, dintre cele "pregătite pentru activare". Acestei stări îi corespunde un șir de așteptare format din 3 liste FIFO, corespunzătoare priorităților 0, 1 și respectiv 2. Introducerea unui proces în acest șir (subprogramul "intproc"), se face în coada listei corespunzătoare priorității sale, iar extragerea (subprogramul "selectproc") se realizează din capul listei nevide, cea mai prioritară.

8.4. Lansarea proceselor

Lansarea unui proces este cerută de programator, în procesul inițial, prin instrucțiunea **init** [CE85a], specifică limbajului PASCAL CONCURENT. La compilarea acestei instrucțiuni, se generează apelul subprogramului de nucleu "initproc", care execută următoarele operații (fig. 8.2.):

```
procedure init_proc(lparam,lvar,lstiva,adrlansare:integer);
var pdc,lungime:integer;
    tranz:ptranz;
begin
  salvproc;
  gestimp;
  initproc(tranzact);
  reztranz(tranz);
  lungime:=lparam+lvar+lstiva+cuvint;
  if lungime>lmax then eroare('depasire memorie');
  rezervazp(lungime,pdc);
  with tranz^ do
    begin
      index:=indexurm;
      indexurm:=indexurm+1;
      pd:=pdc;
      adrleg:=inl;
      timpuctotal:=0;
      timpuc:=0;
      timpabs:=0;
      apelmonit:=0;
      prioritate:=2;
      adrintrerupere:=adrlansare
    end;
  intproc(tranz);
  selectproc
end; {initproc}
```

Fig. 8.2. Lansarea unui proces

- trece în șirul corespunzător stării "pregătit pentru activare" procesul inițial, activ pînă la acest moment (apel "intproc") după ce, în prealabil, s-au salvat în tranzacția sa registrele generale (apel "salvproc") și s-au efectuat calcule de timp aferente (apel "gestimp");
- rezervă (apel "reztranz") și completează tranzacția unui proces;
- rezervă spațiul de memorie necesar procesului în stiva de date și poziționează

corespunzător registrele de bază (apel "rezervazp");

- în cazul în care nu există alt proces "pregătit pentru activare" mai prioritar, lansează noul proces de la prima instrucțiune executabilă ("adrlansare"), prin apelul subprogramului "selectproc".

"tranzact" indică procesul activ (în cazul nostru, procesul inițial), iar "indexurm" contorizează procesele lansate anterior.

8.5. Implementarea conceptului de monitor

Monitorul reprezintă o structură de date utilizată în comun de mai multe procese, precum și toate operațiile pe care aceste procese le pot executa asupra structurii de date respective, operații implementate în program ca proceduri de monitor (cap. 3, §3.4.2.).

Funcția de sincronizare pe care o realizează monitoarele se obține prin aceea că accesul la datele lor se poate face doar prin intermediul procedurilor monitorului, executate strict numai una la un moment dat. În acest scop, monitorului i se asociază, la inițializare, o înregistrare numită "poartă" în care se înregistrează ocuparea lui (fig. 8.3.).

```
type tippoarta=record
                 ocupat:boolean;
                 primul,ultimul:ptranz
               end;
procedure int(var poarta:tippoarta;tranz:ptranz);
begin
  with poarta do
    if primul=nil then
      begin
        primul:=tranz;
        ultimul:=tranz;
      end
    else
      begin
        ultimul^.adrleg:=tranz;
        ultimul:=tranz
      end
end; {int}
procedure cererem(poarta:tippoarta);
begin
  with tranzact do
    begin
      apelmonit:=apelmonit+1;
      prioritate:=0;
    end;
  with poarta do
    if ocupat then
      begin
        salvproc;
        gestimp;
        int(poarta,tranzact);
        selectproc
      end
    else ocupat:=true
end; {cererem}
```

Fig. 8.3. Tratarea solicitării accesului la monitor

Procesele care solicită acces la un monitor ocupat (apel "cererem") sînt înlănțuite între ele (apel "int") prin "adrleg" într-o listă FIFO al cărei cap ("primul") se memorează, de asemenea, în "poartă" (fig. 8.3.).

Eliberarea unui monitor de către un proces (apel "eliberarem"), este urmată de către ocuparea lui cu primul proces din listă. Dacă lista este goală ("primul"=nil), atunci monitorul este declarat liber (fig. 8.4.).

```
function ext(var poarta:tippoarta):ptranz;
var p:ptranz;
begin
  with poarta do
    begin
      if primul=nil then eroare('sir vid');
      p:=primul;
      primul:=p^.adrleg
    end;
  ext:=p
end; {ext}
procedure eliberarem(var poarta:tippoarta);
var p:ptranz;
begin
  with poarta do
    if primul=nil then ocupat:=false
    else
      begin
        p:=ext(poarta);
        intproc(p)
      end;
  with tranzact do
    begin
      apelmonit:=pred(apelmonit);
      if apelmonit=0 then
        begin
          prioritate:=2;
          salvproc;
          gestimp;
          intproc(tranzact);
          selectproc
        end
    end
end; {eliberarem}
```

Fig 8.4. Tratarea ieșirii unui proces dintr-un monitor

Extragerea primului proces din șirul de așteptare la monitor este descrisă în funcția "ext" (fig. 8.4.).

Spre deosebire de monitoare, clasele reprezintă date și acțiuni specifice unui anumit proces, monitor sau unei alte clase și nu pot fi partajate între mai multe procese [CE85a].

Restricțiile de utilizare ale claselor sînt verificate în faza de compilare, unde se semnalează și eventualele erori. Organizarea anumitor acțiuni din program sub formă de clasă permite o mai bună structurare a programului și scurtarea lui (în cazul unor acțiuni repetabile).

8.6. Gestionarea partiției unui program PASCAL CONCURENT

Pentru stabilirea organizării partiției unui program PASCAL CONCURENT, s-a

ținut seama de următoarele aspecte: pe de-o parte programul este în cod obiec (IMT), executabil sub sistemul de operare al calculatorului [CI75] iar, pe de alt parte, zona de memorie de la sfîrșitul programului pînă la sfîrșitul partiției trebui gestionată dinamic, în funcție de necesitățile exprimate prin program. Astfel rezultat structura de partiție prezentată în fig. 8.5.

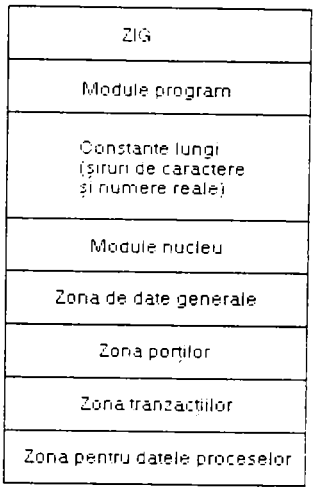


Fig. 8.5. Structura partiției unui program PASCAL CONCURENT

Zona de date generale are o dimensiune redusă și conține date și tabele de lungime fixă, utilizate în nucleu: diferite constante, tabele pentru gestiunea intrărilor/ieșirilor, etc.

Alocarea memoriei necesare unui proces se face la inițializarea procesului, astfel:

- pentru tranzacția procesului, în zona tranzacțiilor;
- pentru variabilele și parametrii procesului, precum și pentru stiva de lucru, în zona proceselor.

In ceea ce privește spațiul de memorie utilizat, procesele sînt independente și protejate între ele și sînt independente de procesul inițial care le-a lansat. Singura posibilitate de comunicare între procese rămîn parametrii de apel și eventual zonele de date din monitoarele utilizate în comun.

In PASCAL CONCURENT procesele nu se pot desființa. Din punct de vedere al programării procesul este de obicei un ciclu infinit (**cycle**). De aceea nu se pune problema unei gestiuni speciale a zonelor alocate pentru diferite procese, în vederea reutilizării lor.

Apelul unui monitor sau a unuia din subprogramele sale externe permite accesul la patru zone de memorie distincte, și anume:

1) *Poarta monitorului* (3 cuvinte), alocată în zona porților la inițializarea monitorului, cu semnificația prezentată în §8.5.

2) *Zona globală a monitorului*, rezervată în locul din stivă corespunzător declarării monitorului ca variabilă (fig. 8.6.), avînd dimensiunea dată de relația:

DZGM = LVM + LPM + AP

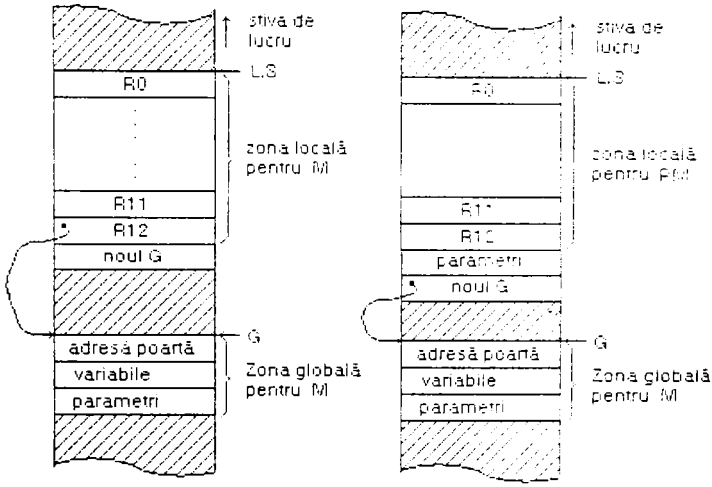
unde: LVM - lungimea zonei variabilelor monitorului;
LPM - lungimea zonei parametrilor monitorului;
AP - un cuvînt rezervat la începutul zonei pentru adresa porții.

Accesul la variabilele sau parametrii globali se obține prin registrul G-R12 (fig. 8.6.)

3) *Zona locală a monitorului sau a rutinei de monitor, apelată*, rezervată în vîrful stivei de date ("s") din momentul apelului, cu următoarea dimensiune:

a) monitoare: DZLM=13 cuvinte (pentru salvarea registrelor generale);

b) proceduri de monitor: DZLPM = LVPM + LPPM + 13 cuvinte, unde:
LVPM - lungimea zonei de variabile a rutinei apelate;
LPPM - lungimea zonei de parametri a rutinei apelate.



a) monitoare (M) b) proceduri de monitoare (PM)

Fig. 8.6. Structura stivei de date la apelul unui monitor sau a unuia din subprogramele sale externe

Accesul la aceste zone se realizează prin bazare cu registrul L-R11 (fig. 8.6.)

4) *stiva de lucru*, care se rezervă în continuarea zonei locale și a cărei dimensiune se calculează în faza de compilare și se transmite în codul obiect, ca parametru.

Vîrful stivei este indicat de registrul S-R10 (fig. 8.6.).

8.7. Implementarea comunicării între module și programe

8.7.1. Apelul subprogramelor scrise în alte limbaje

In PASCAL și PASCAL CONCURENT se pot apela (sub)programe, compilate separat, scrise în PASCAL sau alte limbaje. In programul apelant numele punctelor de intrare apelate trebuie declarate în *directive de compilare REF* [CE85a], avînd următoarea sintaxă:

*REF nume1, nume2, ..., numen.

Intr-un program PASCAL sau PASCAL CONCURENT pot exista mai multe directive REF, plasate oriunde în program, cu condiția ca ele să preceadă directivele de descriere a fișierelor.

Punctelor de intrare declarate, compilatorul le asociază cîte un număr de ordine, care indică poziția numelui respectiv în directivele REF, considerate una în continuarea celeilalte. Subprogramele vor fi încorporate în programul obiect în faza de editare a legăturilor.

Apelul efectiv din PASCAL sau PASCAL CONCURENT al subprogramelor scrise în FORTRAN, COBOL sau ASSIRIS se realizează prin intermediul *procedurii standard "call(x, y)"*. Valoarea parametrului "x" indică numărul parametrilor transmiși către subprogram, iar valoarea parametrului "y" indică numărul punctului de intrare apelat (conform celor precizate anterior).

Apelul procedurii standard "call" trebuie efectuat într-o primă procedură. Parametrii actuali ai acestei proceduri se transmit subprogramului apelat prin "call": parametrii trebuie să fie variabili (de ieșire), de orice tip, sau constanți (de intrare)

de tip articol, tablou sau fișier, pentru ca transmiterea lor să se facă prin adresă [CE85a].

Operațiile pe care le execută subprogramul de nucleu corespunzător procedurii standard "call" sînt:

- construirea tabelei de adrese a celor "x" parametri;
- încărcarea adresei tabelei de parametri în registrul 2;
- salt cu revenire (BAL,8) la punctul de intrare numărul "y" din directivele REF cumulate;
- la revenire, asigură refacerea contextului din momentul apelului și execută reîntoarcerea în programul apelant.

8.7.2. Comunicarea între programe PASCAL. Facilitatea de compilare separată. Segmentarea.

Una din deficiențele majore ale limbajului PASCAL în forma sa inițială [JW74], inclusiv a standardului [AN79] este aceea că nu include facilități de compilare separată. Implementări concrete ale limbajului au încercat să suplinească această deficiență adoptînd în acest sens, instrumente specifice, dintre cele mai diverse.

Un program scris în PASCAL/FELIX C poate face parte, ca un modul, dintr-un sistem de programe cu o anumită structură arborescentă. Astfel, diferite activități se separă în programe compilate independent care se apelează unele pe altele. Unul sau mai multe programe pot fi grupate în segmente, în maniera proprie calculatoarelor din gama FELIX C, prin intermediul cartelelor de comandă SEG, ENTSG, TREE, etc. Comunicarea între diferitele programe PASCAL sau între programe PASCAL CONCURENT și PASCAL (§8.7.3) se realizează prin intermediul prefixului și a listelor de parametri ale programelor respective.

Prefixul este format dintr-o succesiune de definiții de tipuri și constante, declarații de proceduri și funcții, conținînd doar titlul acestora și declarații de programe apelate (fig. 8.7.).

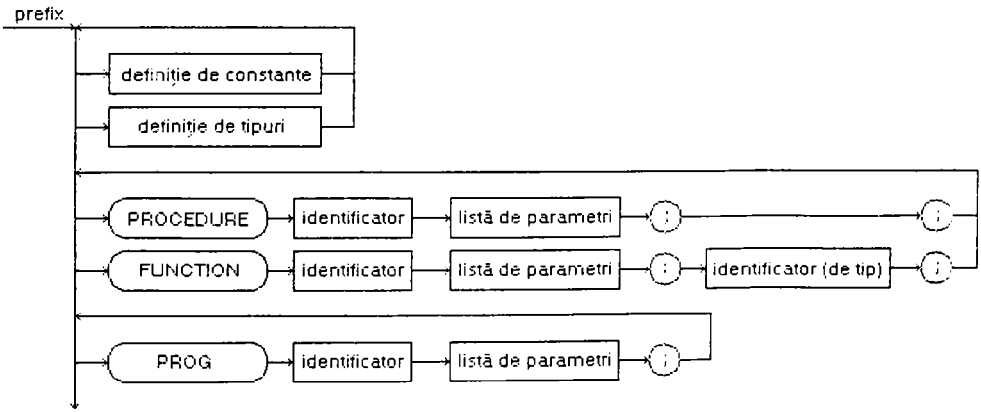


Fig. 8.7. Diagrama de sintaxă a prefixului unui program PASCAL

Prefixul, dacă există, trebuie să preceadă declarația **program** a programului PASCAL (antetul programului). Identificatorii definiți sau declarați în prefix sînt globali și vizibili în tot programul PASCAL. Prin urmare ei pot fi utilizați, cu sensul asociat, în orice punct din program, cu excepția blocurilor în care au fost redefiniți. Procedurile și funcțiile introduse în prefix permit comunicarea cu programul PASCAL CONCURENT din care s-a apelat ori s-a lansat programul PASCAL (§8.7.3). Declarațiile formale de programe din prefix, evidențiate prin cuvîntul cheie **prog**, servesc apelării altor programe PASCAL din programul curent.

Prin intermediul unei *directive DEF* [CE85a], programului PASCAL i se asociază un nume care va servi la referirea lui din exterior, astfel:

```
*DEF nume
```

Numele indicat ca definiție externă este punctul de intrare în program și va fi precizat și ca punct de intrare în segment (într-o cartelă de comandă ENTSG).

Pentru ca nucleul să poată realiza apelul unui program PASCAL din alt program PASCAL, în cele două programe se vor include următoarele operații:

- printr-o directivă DEF programul apelat va primi un nume în vederea referirii lui din exterior; același nume va apărea într-o directivă REF (§8.7.1.) în programul apelant;
- în prefixul programului apelant se declară formal programul apelat, printr-o declarație **prog**, conform sintaxei din fig. 8.7.;
- apelul propriu-zis se indică printr-o instrucțiune de apel, în maniera cunoscută pentru proceduri.

Ultimul parametru formal din declarația **prog** va fi în mod obligatoriu constant, de tip întreg. Numărul transmis ca argument pentru acest parametru, indică programul apelat. El reprezintă numărul de ordine al numelui programului în lista argumentelor directivelor REF cumulate (analog cu procedura standard "call", §8.7.1.). Restul parametrilor servesc transmiterii de date între programe și pot să fie variabili (de ieșire) sau constanți (de intrare). Ei vor corespunde, prin urmare (ca număr și tip), cu parametrii formali din antetul (declarația **program**) a programului apelat.

Variabilele declarate într-un program PASCAL apelat există doar pe parcursul executării acestuia. Zona de memorie ocupată de ele este eliberată automat de nucleu, la revenire. În schimb, variabilele alocate dinamic există și după încheierea activității programului care le-a creat prin apelul procedurii standard "new" (dacă nu s-a cerut explicit eliberarea zonelor de memorie respective prin "dispose"). Variabilele dinamice pot constitui deci un mijloc de comunicare prin transmiterea de date între programe PASCAL (apelant și apelat).

8.7.3. Apelul și lansarea programelor secvențiale dintr-un program PASCAL CONCURENT

În vederea proiectării unor sisteme concurente complexe pe baza unor metodologii și facilități accesibile în diferite limbaje, PASCAL CONCURENT/FELIX C - versiunea 2 [CE83a] introduce posibilitatea apelării sau lansării din programul concurent a unor programe secvențiale realizate în alte limbaje: PASCAL, ASSIRIS, FORTRAN, COBOL. Această modalitate se adaugă aceleia de a apela subprograme scrise în alte limbaje, care a fost prezentată în §8.7.1.

In PASCAL CONCURENT se pot *apela* numai programe PASCAL. Această operație este similară cu cea descrisă în §8.7.2. Programele apelate se compilează separat între ele și pot fi situate în același segment sau în segmente diferite. Intreg sistemul de programe rezultat în urma editării legăturilor este considerat ca formînd un unic program concurent. El va fi compus dintr-un singur program PASCAL CONCURENT plasat obligatoriu în segmentul rădăcină, ca primul modul al acestuia și dintr-un număr oarecare de programe PASCAL, apelate fie din programul concurent, fie unele din altele. In momentul în care se execută acțiuni dintr-un program apelat, se consideră activ procesul ce a efectuat apelul. Intrucît procesele își desfășoară activitatea în paralel, este interzis ca două sau mai multe procese să apeleze concomitent programe situate în segmente paralele și care s-ar acoperi între ele. De asemenea, este interzis să se apeleze simultan, din mai multe procese, același program PASCAL care operează asupra unor resurse partajate (de exemplu fișiere). Aceste inconveniente se rezolvă efectuînd apelurile respective în monitoare care le transformă din apeluri concomitente în succesive.

Fiecare program apelat va primi un nume printr-o directivă de compilare DEF

(§8.7.2). In programul apelant (PASCAL CONCURENT sau PASCAL), numele programelor apelate se declară în directive de compilare REF (§8.7.1.).

In afară de apelare, în PASCAL CONCURENT se pot *lansa* programe obiect (IMT) realizate în orice limbaj secvențial (PASCAL, ASSIRIS, FORTRAN, COBOL). Programele care se vor lansa sînt bibliotecate independent într-o bibliotecă de format IMT. Spre deosebire de apeluri, nu există nici o legătură prealabilă (realizată la editarea legăturilor) între aceste programe și programul concurent care efectuează lansările. Acesta ia cunoștință de identitatea programului lansat și a bibliotecii în care se află doar în momentul execuției. Pentru operația propriu-zisă de lansare, în nucleu se execută următoarele activități:

1) Se încarcă programul secvențial în zona de memorie corespunzătoare procesului care a solicitat lansarea. Procesul va avea acces exclusiv la programul respectiv. În cazul în care același program se va lansa și dintr-un alt proces, acesta din urmă va opera asupra altui exemplar al programului, încărcat în altă parte a memoriei. Pentru programul lansat se construiește o "partiție" proprie, în cadrul partiției programului concurent. Astfel se permite ca programe secvențiale corecte să fie rulate prin lansare în condiții identice executării lor independente. Erorile detectate în faza de încărcare a programului secvențial se semnalează în programul concurent (fig. 8.9. - cîmpul "er") și operația de încărcare este abandonată. Întrucît se revine în programul concurent, la instrucțiunea următoare celei prin care s-a cerut lansarea, există posibilitatea de a cerceta prin program cauza erorii și de a relua eventual operația.

2) Se activează programul secvențial încărcat de la prima instrucțiune executabilă, după care procesul respectiv efectuează acțiunile din programul secvențial în paralel cu activitatea celorlalte procese concurente. O importanță aparte o are în acest context punerea în așteptare a unui proces pe durata în care un program secvențial lansat de către el execută o operație de I/E și activarea altui proces dintre cele "pregătite pentru activare". Erorile survenite pe parcursul executării programelor lansate se semnalează în maniera proprie limbajelor în care acestea au fost scrise. În același timp, terminarea cu eroare se indică și în cîmpul "er" (fig. 8.9).

3) În momentul terminării (corect sau cu eroare) unui program lansat, zona de memorie afectată acestuia se eliberează și se revine în programul concurent la instrucțiunea următoare celei de lansare, de unde se continuă activitatea procesului respectiv.

8.7.4. Interfața cu utilizatorul

Programele secvențiale apelate sau lansate dintr-un program PASCAL CONCURENT trebuie declarate formal (printr-o declarație **program**) în blocul tipului sistem în care se execută apelul sau lansarea, conform figurii 8.8.

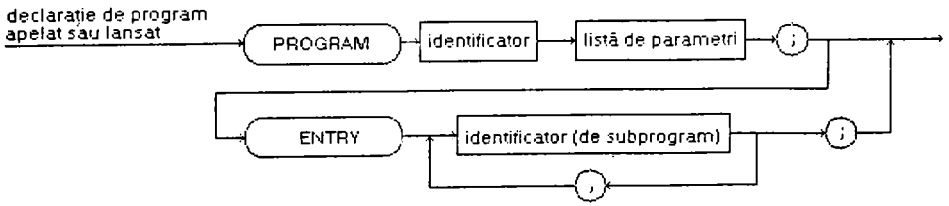


Fig. 8.8. Sintaxa declarației de programe secvențiale apelate sau lansate

La declarare se precizează un număr "n" (n>=1) parametri formali. Primii n-1 parametri servesc la transmiterea de date între programul concurent și programele PASCAL (secvențiale) apelate sau lansate. Ei trebuie să corespundă ca număr, ordine

și tip cu parametrii formali declarați în antetul programului secvențial respectiv. Ultimul parametru formal, numit *de comandă*, a fost prevăzut pentru identificarea programului care trebuie apelat sau lansat. În cele două cazuri el va fi declarat în felul următor:

a) la apel: parametru constant de tip întreg;

b) la lansare: parametru de tip articol, cu structura indicată în fig. 8.9.

După cum rezultă din fig. 8.8., la declararea programelor secvențiale se poate indica o listă de rutine externe (**entry**), care definește *interfața* între programul concurent și programul PASCAL (secvențial) apelat sau lansat.

Apelul sau lansarea propriu-zisă se execută în maniera în care se apelează procedurile. Se indică numele programului din declarația sa formală și un număr de "n" (n>=1) parametri actuali. Primii n-1 parametri actuali sînt variabile sau constante prin care se realizează schimbul de informații între programul concurent și programele PASCAL apelate sau lansate. Ultimul parametru actual va fi prevăzut astfel:

a) la apel: un număr întreg care precizează numărul de ordine al numelui programului apelat în lista cumulată a directivelor REF: este analog cu al doilea parametru la apelul procedurii standard "call" (§8.7.1.) și cu ultimul parametru la comunicarea între programe PASCAL (§8.7.2.);

b) la lansare: o variabilă de tip articol, cu structura parametrului formal corespunzător (fig. 8.9).

Zonele de date ale programelor PASCAL apelate sau lansate se rezervă împreună cu codul obiect al programelor lansate, în zona de memorie corespunzătoare procesului care solicită operația. Din acest motiv, la definirea unor astfel de tipuri proces, programatorul trebuie să indice necesarul suplimentar de memorie, în următoarea manieră:

type p=**process**(...);+ci

"ci" este o constantă întreagă care indică un număr de cuvinte calculator (4*ci octeți) ce vor fi alocate suplimentar procesului respectiv, în scopul precizat anterior (0=<ci=<65535). Necesarul suplimentar de memorie depinde de operația efectuată (apel sau lansare) și de limbajul în care este realizat programul lansat astfel:

1) *programe PASCAL apelate*: "ci" va corespunde sumei dimensiunilor zonelor de variabile alocate static și dinamic, plus o zonă de manevră pentru efectuarea unor operații standard;

2) *programe PASCAL lansate:* la necesarul de la punctul 1) se adaugă lungimea codului obiect al programului respectiv;

3) *programe ASSIRIS, FORTRAN sau COBOL, lansate*: necesarul suplimentar de memorie corespunde lungimii codului obiect, plus eventual o cantitate de memorie suplimentară rezervată în continuarea codului obiect ("lg" - fig. 8.9.).

În fig. 8.9. se prezintă structura completă a parametrului de comandă. O parte din informațiile furnizate prin el sînt utilizate pentru identificarea bibliotecii și a programului ce urmează să fie lansat. O altă parte este reprezentată de date necesare executării programului. Pentru lansarea de programe realizate în limbaje diferite de PASCAL, lista de argumente cuprinde doar acest parametru.

```
type comanda=record
                dv:array [1..4] of char;
                vs,ln:array [1..8] of char;
                gn,vn:integer;
                fn:array [1..8] of char;
                er,un,lg,key:integer;
                option:array [1..n] of char;
                stop:array [1..4] of char
              end
```

Fig. 8.9. Structura parametrului de comandă pentru operația de lansare

Semnificația cîmpurilor este următoarea:

- dw: indică suportul magnetic al bibliotecii sau numai tipul suportului("dot"în[CI75]).
- ws: este identificatorul de volum al suportului bibliotecii; este luat în considerare atunci cînd "do" indică doar tipul suportului.
- ln: reprezintă numele bibliotecii.
- gn: indică numărul de generație al bibliotecii.
- fn: reprezintă numele programului ce urmează să fie încărcat și lansat.
- er: prin acest cîmp se retransmite numărul erorii care a apărut la încărcarea, lansarea sau executarea programului; în principiu, er=0 înseamnă program încărcat corect, lansat și terminat normal.
- vn: este numărul de punere la zi al programului din bibliotecă.
- lg: indică o cantitate suplimentară de spațiu de memorie, în cuvinte calculator (4+lg octeți) care se va rezerva în continuare codului obiect, în cadrul "partiției" construite pentru programul lansat; se va utiliza, de exemplu, la lansarea unor programe cu proceduri de sortare.
- key: reprezintă cheia de exploatare a programului lansat.
- option: pe lungimea indicată în primul octet, acest șir de caractere se transferă în zona de opțiuni a programului lansat.
- stop: conține șirul de caractere 'STOP', care indică sfîrșitul informațiilor din articol.

Cîmpurile "dw", "ws", "ln", "gn", "vn", "fn", "er" și "stop" sînt obligatorii; celelalte cîmpuri sînt opționale. Cu excepția lui "stop", cîmpurile se identifică în cadrul articolului prin poziție. Din această cauză, rezervarea unui cîmp opțional implică rezervarea tuturor cîmpurilor din fața lui. Informațiile conținute în cîmpurile tablouri de caractere, trebuie să fie cadrate la stînga. Detalii privind semnificația acestor cîmpuri, precum și exemple privind apelul și lansarea de programe secvențiale di PASCAL CONCURENT, se găsesc în [CI75, CE83a, CE85a].

La declararea formală a programelor PASCAL (secvențiale) ce urmează să fie apelate sau lansate se poate indica o listă (**entry** - fig. 8.8.) de subprograme externe ale tipului sistem în care se execută apelul sau lansarea. Aceste subprograme pot fi apelate direct din programul PASCAL respectiv, cu condiția ca ele să fie declarate strict în aceeași ordine (întrucît identificarea se face prin poziție) și în prefixul programului apelat sau lansat (§8.7.2.). Ele reprezintă deci o *interfață* între programul PASCAL CONCURENT și programele PASCAL apelate sau lansate. Intrepătrunderea între programe diferite, realizată prin acesată modalitate este deosebit de spectaculoasă și nu are corespondent în alte limbaje, pe calculatoarele din gama FELIX C.

In legătură cu utilizarea unor proceduri și funcții externe ca subprograme de interfață există următoarele particularități:

- este interzis să se utilizeze în acest scop rutinele externe dintr-un monitor;
- se pot utiliza proceduri și funcții externe dintr-un tip **process** ; acestea, de altfel, nici nu pot fi întrebuințate în alt scop;
- nu este obligatoriu ca identificatorii rutinelor enumerate în lista **entry** a unei declarații **program** (fig. 8.8) să fie definiți în punctul respectiv din program; ei trebuie să fie definiți pînă la încheierea tipului sistem curent;aceasta este singura excepție de la regulile de vizibilitate (cap. 5), pe care o permite limbajul PASCAL CONCURENT .

In concluzie, considerînd programul concurent ca fiind sistem sau subsistem de operare apelarea rutinelor de prefix reprezintă modalitatea prin care programele secvențiale pot cere activarea unor funcții speciale din sistemul concurent.

8.7.5. Funcțiile nucleului concurent privind operația de lansare a programelor secvențiale

Modalitatea de încărcare și lansare a programelor secvențiale adoptată în PASCAL CONCURENT diferă substanțial de aceea definită de P. B. Hansen [BH75b] și

implementată de A. C. Hartmann pe calculatorul PDP 11/45 [HR77]. Există diferențe majore, afectînd principiul de lucru și diferențele de formă, prin felul în care se solicită operația în program. Principalele deosebiri sînt generate de statutul diferit al programului PASCAL CONCURENT pe calculatorul FELIX C (rulează într-o partiție fixă, sub controlul sistemului de operare) și de modul diferit de prezentare a programelor obiect (IMT). Astfel, compilatorul lui Hartmann generează cod obiect virtual, într-un fișier secvențial obișnuit, executarea lui făcîndu-se cu o mașină virtuală adecvată. În aceste condiții, operația de încărcare a programului secvențial (exclusiv PASCAL), echivalentă cu citirea unui fișier secvențial, este lăsată în seama programatorului, iar mașina virtuală care execută programul concurent este capabilă ca, fără nici o completare, să execute și eventualele programe secvențiale. Pe calculatoarele din gama FELIX C, încărcarea programelor obiect și pregătirea sistemului de operare pentru a fi capabil să execute două sau chiar mai multe programe IMT în cadrul aceleiași partiții sînt operații complicate care trebuiau, obligatoriu, să fie preluate din nucleu. Această facilitate deosebită, apelabilă cu multă ușurință dintr-un limbaj de nivel înalt (PASCAL CONCURENT), schimbă de fapt principiul de gestiune a memoriei (cu partiții fixe), specific calculatorului FELIX C. Ea a reprezentat, alături de alte modificări mai puțin importante, saltul de trecere la versiunea 2 a limbajului PASCAL CONCURENT/FELIX C [CE83a, CE83b].

Nucleul trebuia să rezolve, în prima fază, încărcarea programului secvențial în stiva procesului activ, urmată de lansarea lui în execuție. Pentru a realiza multiprogramarea proceselor în cadrul partiției, nucleul trebuie să fie capabil să intercepteze apelurile macroinstrucțiunilor WAIT activate din modulele de acces aferente programului lansat. Interceptarea unui WAIT trebuie să fie urmată de activarea unui alt proces, dintre cele "pregătite pentru activare". De asemenea, sistemul trebuie să sesizeze terminarea unui program lansat.

O serie întreagă de probleme apar în legătură cu lansarea unui program care utilizează fișiere, atît la încărcare, cînd trebuie construite repertoriul SGF și fișierul FILCOM corespunzătoare [CI76], cît și la execuție [CI81c].

In cazul încărcării programelor segmentate, nucleul trebuie să identifice segmentele și să le poziționeze corect în memorie (segmente de program, segmente de date, de COMMON provenite dint-un program scris în FORTRAN, etc.). In fine, pentru a rezolva încărcarea programelor, nucleul trebuie să cunoască suportul pe care se găsește biblioteca utilizator, precum și toate informațiile relative la bibliotecă și la programul ce urmează să fie lansat. Aceste informații se furnizează din programul PASCAL CONCURENT în maniera prezentată în §8.7.4.

Avînd în vedere considerațiile de mai sus, nucleul a fost completat astfel:

- un modul încărcător;
- un modul pentru interceptarea și tratarea macroinstrucțiunilor WAIT, activate din programul lansat;
- o modalitate de sesizare a terminării operațiilor de I/E și relansare a proceselor întrerupte, de la instrucțiunea următoare;
- un modul pentru terminarea tratării programelor secvențiale lansate.

Încărcătorul tratează rădăcina programului (dacă acesta nu este segmentat - întregul program), construind repertoriul SGF în zona de memorie rezervată acestui program și adăugînd fișierului FILCOM existent în acel moment, informațiile necesare. Pentru a putea realiza corect și complet aceste obiective, este obligatoriu ca, la catalogarea în bibliotecă, cartelele de comandă ASSIGN, LABEL și FILE să preceadă cartela de comandă LINK. Programul IMT trebuie deci să conțină toate informațiile relative la fișiere. De asemenea, la editarea legăturilor este obligatorie opțiunea (implicită) FMS, care asigură incorporarea modulelor de acces.

Dacă toate aceste condiții sînt respectate, programele lansabile de către nucleu pot fi segmentate și pot utiliza fișiere, în limitele acceptate de sistemul de operare. In plus, zona DCZ corespunzătoare poziției [CI76] trebuie să fie suficient de mare pentru a permite adăugarea tuturor informațiilor suplimentare.

In momentul în care un proces solicită lansarea unui program secvențial, se activează încărcătorul. Acesta identifică biblioteca și programul în cauză, verifică dacă programul încape în stiva procesului respectiv și, în caz afirmativ, execută încărcarea propriu-zisă.

Prin interceptarea macroinstrucțiunilor WAIT activate din programul secvențial, după lansarea opeerațiilor de I/E, nucleul evită blocarea partiției. Într-un astfel de moment, se întrerupe procesul respectiv și se lansează procesul cel mai prioritar dintre cele "pregătite pentru activare". Achitarea operației de I/E este urmată de reactivarea procesului întrerupt. La terminarea unui program lansat, nucleul realizează refacerea fișierului FILCOM inițial și asigură continuarea procesului de la instrucțiunea următoare lansării.

Capitolul 9

MEDIU DE PROGRAMARE PORTABIL PENTRU LIMBAJUL EDISON PE MICROSISTEME DE CALCUL. VARIANTE DE IMPLEMENTARE

În perioada 1984-1989 la I.P.T.V. Timișoara s-a desfășurat o intensă cercetare științifică aplicativă avînd ca obiectiv implementarea limbajului de programare EDISON pe diferite tipuri de calculatoare, cu predilecție pe microsisteme. Într-o primă fază s-a realizat, cu caracter experimental și didactic, un compilator într-o singură trecere pentru un subset al limbajului EDISON. Această activitate a fost urmată de proiectarea și realizarea unui compilator original și performant, pentru întreg limbajul EDISON, scris în PASCAL/FELIX C. Acest prim compilator profesional generează la ieșire cod virtual și a fost organizat în trei treceri, astfel:

1) Analiză lexicală, analiză sintactică parțială (numai pentru definiții și declarații) și analiză de domeniu.

2) Analiză sintactică parțială (numai pentru instrucțiuni), analiză semantică și generare de cod virtual provizoriu.

3) Definitivarea și optimizarea codului virtual, simularea stivei din execuție pentru evaluarea spațiului de memorie necesar procedurilor și proceselor programului și afișarea erorilor de compilare.

Existența acestui compilator a permis abordarea realizării unui sistem de programare portabil pentru limbajul EDISON, după modelul celui prezentat de P. B. Hansen în [BH82]. Portabilitatea sistemului este obținută în principal prin scrierea chiar în EDISON a majorității componentelor (exceptînd executivul). Ele au fost compilate, inițial, pe calculatorul FELIX C-256 iar codul virtual rezultat a fost transportat pe microcalculatorul FELIX M-18, pe care s-a implementat întregul sistem, avînd ca suport banda magnetică. Executarea codului virtual pe noul calculator se realizează de către un program executiv, parte componentă a sistemului, scrisă în limbaj de asamblare.

9.1. Structura și funcțiile sistemului de programare EDISON

Structura sistemului este prezentată în fig. 9.1.

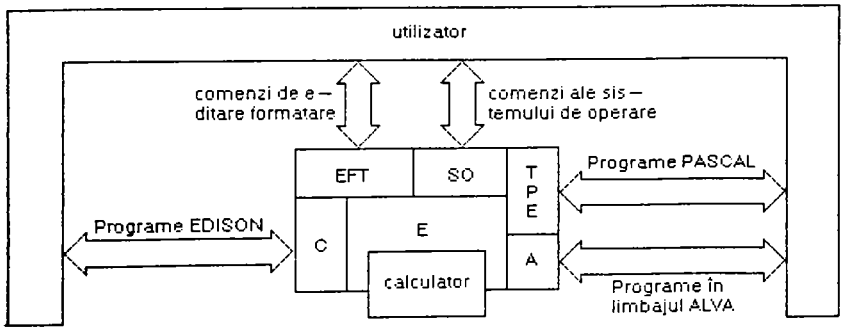


Fig. 9.1. Structura sistemului de programare EDISON

Funcțiile părților componente sînt [EC82]:

Executivul (E): implementînd mașina virtuală prezentată în §9.2., face legătura între calculatorul real și celelalte părți componente. Acestea au fost scrise în EDISON și au fost translatate anterior, de către compilator, în codul virtual

acceptat de executiv. Este alcătuit, în principiu, din două părți:

a) *Interpretatorul codului virtual*: interpretează și execută pe calculatorul real succesiunea de coduri virtuale de la intrarea sa.

b) *Nucleul*: realizează cîteva funcții speciale legate de structura și particularitățile sistemului de calcul, implementarea funcțiilor de programare concurentă, etc.

S-au realizat două executive scrise în limbaj de asamblare, unul pentru calculatoarele din gama FELIX C, ca suport al compilatorului scris pe acest tip de calculatoare și altul pentru familia de calculatoare FELIX M, ca suport al sistemului de programare portabil. Este tratat în detaliu în §9.3.

Compilatorul (C): este realizat după modelul din [BH82]. Programele compilate, translatate în cod virtual, rulează pe calculator prin intermediul aceluiași executiv, după schema simplificată din fig. 9.2.

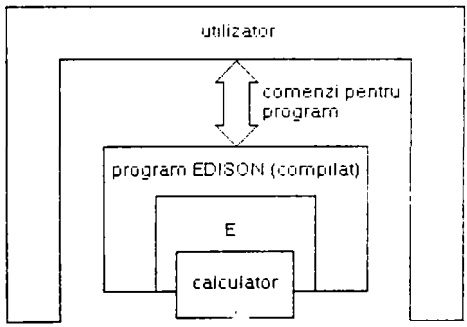


Fig. 9.2. Execuția unui program EDISON

Editorul - formator de texte (EFT): furnizează funcțiile necesare pentru crearea, actualizarea și, în general, gestionarea programelor sursă aflate în bibliotecile sistemului. In cooperare cu asamblorul, translatorul PASCAL-EDISON și compilatorul, contribuie la punerea la punct a programelor sursă. S-au realizat două astfel de editoare: unul scris în PASCAL, încorporînd funcții de editare mai generale și altul scris în EDISON și orientat pe sintaxa limbajului EDISON.

Sistemul de operare (SO): a fost realizat după modelul sistemului de operare Mono [BH82, RA84]. Principalele sale funcții sînt: efectuarea dialogului cu utilizatorul, gestionarea memoriilor externe și încărcarea programelor. Aceste funcții vor fi detaliate în §9.4.

Translatorul PASCAL-EDISON (TPE): permite translatarea în EDISON a unor programe scrise în PASCAL (cu anumite restricții). Se evită astfel rescrierea în EDISON a acestor programe. Programul rezutat este ulterior prelucrat de compilatorul sistemului de programare.

Asamblorul (A): este un program de traducere pentru un limbaj, ALVA, cu facilități de nivel scăzut, definit de P. B. Hansen în [BH82]. Este util la scrierea unor aplicații care reclamă astfel de facilități. De exemplu, executivul sistemului proiectat de Brinch Hansen este scris în ALVA.

Cu excepția executivului, celelalte părți componente sînt scrise chiar în EDISON. In consecință, trecerea sistemului de pe un calculator pe altul implică doar adaptarea sau rescrierea executivului și, eventual, mici intervenții în textul sistemului de operare. Codul virtual este independent de calculator. El a fost definit pornind de la caracteristicile limbajului de programare EDISON, ținîndu-se cont și de unele necesități de optimizare (§9.2.) care să confere performanțe ridicate la faza de execuție (necesar de memorie redus, viteză de execuție mare).

Realizat în această manieră, în cea mai mare parte a sistemului de programar

(peste 90%) se ignoră complet particularitățile sistemului de calcul, ceea ce conduce la un grad înalt de portabilitate.

9.2. Probleme specifice definirii și implementării unei mașini virtuale

9.2.1. Avantajele utilizării unei mașini virtuale pentru implementarea limbajelor de programare

Din punct de vedere al echilibrului între eficiența implementării unui limbaj de programare pe un anumit calculator și efortul depus pentru atingerea acestui scop se pun mai multe probleme cu importanță îndeosebi practică și anume [CE84d]:

- reducerea complexității programelor de translatare din limbajul sursă (LS) în limbajul mașină (LM);
- simplificarea și eficientizarea la maximum a activității de programare în limbajul respectiv;
- reducerea timpului de execuție a programelor chiar și în detrimentul timpului de translatare (compilare);
- asigurarea portabilității simple a limbajului ceea ce implică automat și portabilitatea programelor.

Utilizarea unei mașini virtuale (MV) ca interfață între compilator și calculator (mașina reală) răspunde în mod satisfăcător problemelor de mai sus dacă se rezolvă anumite cerințe. Dintre acestea, o parte țin de realizarea compilatorului (limbajul în care este scris, verificări complete și diagnostice de erori clare și oportune, optimizarea codului obiect, etc.) și nu constituie preocuparea acestui paragraf. Cerințele legate de MV, care au fost și rezolvate practic la implementarea limbajului EDISON [CE86b], pot fi la rîndul lor împărțite în două categorii:

- definirea optimă a setului de operații (instrucțiuni virtuale - IV) ce urmează să fie executate de MV atît din punct de vedere al sintaxei cît și al semanticii lor (§9.2.2.); aceste operații *codul virtual* (CV);
- stabilirea structurii și funcționării MV în strînsă corelare atît cu semantica și cu pragmatica limbajului implementat, cu complexitatea și specificul său (secvențial sau concurent), cît mai ales cu structura și posibilitățile sistemului de calcul (SC); în acest sens s-au analizat MV realizate de P. B. Hansen pentru PASCAL secvențial și concurent (8Koct) și respectiv EDISON (2Koct), precum și diferite variante de implementare a limbajelor concurente pe sisteme multiprocesor [HR77, BH82, EC83, EC86b].

Pentru implementarea limbajului EDISON pe un SC monoprocesor a rezultat structura de MV prezentată în fig. 9.3.

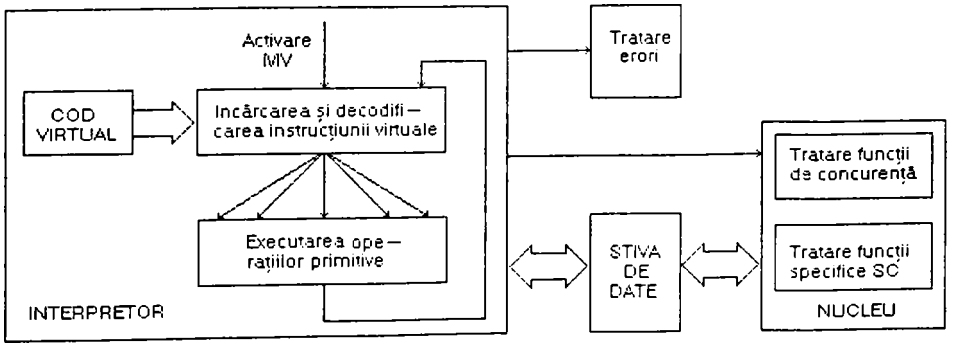


Fig. 9.3. Structura MV pentru implementarea limbajului EDISON

9.2.2 Probleme ale definirii unui cod virtual optim

La analiza în vederea definirii CV pentru implementarea unui limbaj de programare, trebuie să se țină cont de următoarele aspecte:

- Stabilirea IV și a semanticii lor în corelare cu semantica limbajului implementat: ce fel de date și structuri de date se acceptă; ce prelucrări se permit asupra acestor structuri; modul de lucru al instrucțiunilor limbajului; categorii de subprograme ce se pot defini și modul lor de apelare. Respectarea acestei cerințe conduce la reducerea complexității compilatorului și implicit a dimensiunii lui prin simplificarea sintezei CV pornind de la instrucțiunile sursă. Se reduce de asemenea și timpul de compilare.

- Alegerea dimensiunii, structurii și semanticii IV în concordanță și cu modul de lucru și posibilitățile mașinilor reale pe care urmează să se realizeze implementarea: lungimea cuvîntului calculator, posibilitățile LM, stivă de date gestionată prin hardware, categorii de echipamente periferice și modul de acces la ele. In acest fel se poate reduce substanțial dimensiunea MG și timpul de execuție al programelor.

- Uniformizarea la maximum a sintaxei IV pentru a simplifica analiza, decodificarea și executarea lor, cu aceleași consecințe ca mai sus.

In ultimă instanță, criteriile principale pentru definirea unui CV optim trebuie să fie reducerea timpului de execuție a programelor concomitent cu scurtarea lungimii CV.

Structura de principiu a unei IV este prezentată în fig. 9.4.

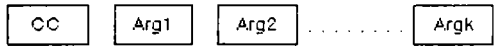


Fig. 9.4. Structura unei instrucțiuni virtuale

CC - cod de comandă; precizează operația ce trebuie efectuată.

Arg1, ..., Argk - argumente; în cazul unei MV ce lucrează cu stivă de date și prin care se vor transmite deci operanzii, argumentele precizează unele informații suplimentare necesare efectuării operației ca de exemplu: nivelul și deplasamentul unei variabile, lungimea operanzilor, deplasamentul pentru operații de salt, etc.

Intr-o variantă simplă CC și argumentele pot avea fiecare aceeași lungime, de exemplu lungimea cuvîntului calculator (2oct pentru multe micro și minisisteme). Deși în această variantă CV se manipulează ușor în interpretor, ea nu poate fi optimă întrucît plaja de valori posibilă pentru multe argumente este suficient de redusă pentru a fi reprezentată pe octet sau chiar pe mai puțin. Rezultă de aici două modalități de reducere a codului virtual:

- utilizarea de argumente de lungime diferită (1 sau 2 octeți, dupa caz);
- împachetarea pentru anumite instrucțiuni virtuale a CC cu argumentul corespunzător în același cuvînt calculator.

Cea mai eficientă metodă de scurtare a lungimii codului virtual generată pentru un program este aceea de a diversifica CC de așa manieră încît să dispară unul sau chiar două - trei argumente. Spre exemplu, utilizarea de CC diferite pentru a manipula operanzi de tipuri și lungimi diferite, de pe diferite nivele ale programului nu mai necesită prezența ca argumente a tipului, lungimii și nivelului operandului respectiv. Rezultă în acest caz cîte o familie de CC pentru un anumit tip de operație (încărcare, memorare, salt, operații logice și relaționale, operații aritmetice, apel, etc.) [TA79].

Diversificarea CC conduce automat și la scurtarea timpului de execuție a programelor prin eliminarea, rotată cu argumentele, și a unor teste și prelucrări referitoare la ele. Gruparea în interior a operațiilor primitive înrudite, ca puncte de intrare în același subprogram, cu părți distincte și părți comune diverselor operații, asigură ca dimensiunea interpretatorului să nu crească în mod semnificativ.

9.2.3. Aspecte specifice unei MV concurente

O MV pentru implementarea unui limbaj de programare concurentă trebuie să permită atît efectuarea operațiilor de prelucrare secvențială a datelor și structurilor de date definite în limbaj cît și realizarea următoarelor funcții specifice [CE84a]:

- gestiunea proceselor concurente și a resurselor logice din program;
- gestiunea surselor fizice ale sistemului de calcul.

Se obișnuiește ca aceste funcții să fie organizate în cadrul MV sub forma unui modul distinct numit *nucleu* (fig.9.3.), operațiile de nucleu fiind apelabile din cadrul virtual prin intermediul interpretatorului.

Forma și conținutul instrucțiunilor virtuale pentru realizarea și gestionarea concurenței proceselor depinde foarte mult de limbajul implementat. Diversitatea mare de metode de sincronizare și comunicare, de moduri de definire, creare, lansare și eventual distrugere a proceselor nu permite nici măcar discutarea în mod global a acestei probleme. Este posibil ca anumite operații de nucleu, pentru același limbaj, să difere de la o implementare la alta. In această categorie poate intra spre exemplu gestiunea proceselor și a memoriei interne și în mod obligatoriu gestiunea unităților periferice. Modul concret în care s-au abordat și s-au rezolvat aceste probleme în executivul sistemului de programare pentru limbajul EDISON, este prezentat în §9.3.

9.3. Nucleul executivului pentru sistemul de programare EDISON

9.3.1. Structura și funcțiile executivului și nucleului

Executivul reprezintă un set de programe care asigură anumite funcții necesare rulării pe o mașină dată a codului obiect rezultat în urma compilării textului sursă, scris într-un limbaj de nivel înalt. Pornind de la sarcinile pe care trebuie să le rezolve, executivul pentru un limbaj concurent se compune din [CE84a]:

- încărcătorul de programe;
- modulul de comunicare cu operatorul;
- interpretatorul codului virtual (în cazul în care compilatorul generează la ieșire un asemenea cod);
- nucleul.

Incărcătorul și modulul de comunicare cu operatorul îndeplinesc funcții pregătitoare pentru activarea codului obiect al programului.

Nucleul este acea parte a executivului care realizează interfața cu resursele fizice ale sistemului și asigură funcțiile care nu pot fi exprimate prin codurile mașinii virtuale definite pentru limbaj. Sarcinile ce pot reveni nucleului sînt următoarele [BH70]:

- planificarea proceselor și gestionarea unității centrale;
- sincronizarea proceselor;
- gestionarea memoriei;
- programarea perifericelor;
- tratarea întreruperilor;
- gestionarea timpului;

Nucleul reprezintă astfel o extindere a procesorului real (fizic), formînd împreună cu acesta un așa numit *procesor extins*, ale cărui funcții deservesc interpretatorul(fig. 9.5). Interpretatorul împreună cu procesorul extins poate fi privit ca un procesor virtual, care execută codul virtual reprezentînd programul concurent compilat.

In ultimii ani s-au cristalizat următoarele cerințe în ceea ce privește implementarea funcțiilor nucleului și relațiile "programator-limbaj-nucleu" în realizarea programelor concurente [JO79]:

- o parte cît mai mare a funcțiilor caracteristice sistemelor de operare să fie realizate de programator, în limbajul de nivel înalt (crește libertatea programatorului și transparența implementării limbajului);

- definirea limbajului să fie de așa manieră încît să permită o cît mai mare libertate în adoptarea politicilor de realizare a unui nucleu;
- reducerea la minim a nucleului, pentru a ușura implementarea (mărește portabilitatea limbajului).

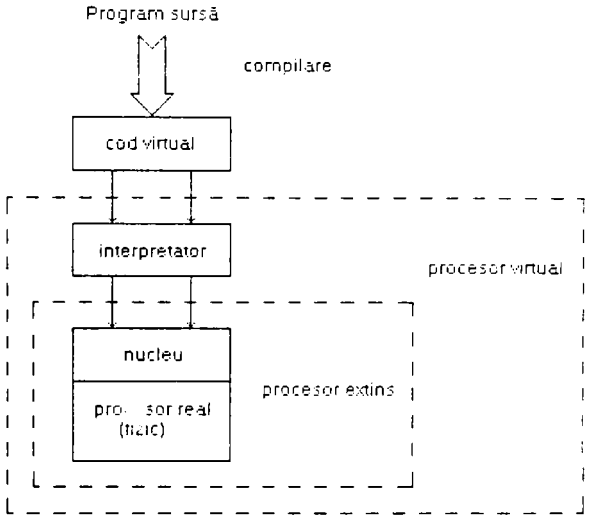


Fig. 9.5 Locul nucleului în cadrul executivului

Pornind de la aceste considerente se vor prezenta în continuare, în detaliu, funcțiile de bază ale nucleului pentru limbajul EDISON, implementat pe un sistem monoprocesor. Pentru a mări claritatea și generealitatea tratării nucleul este considerat ca fiind format din mai multe module (fig. 9.6) prezentate chiar în limbajul EDISON (se definește, în plus, un tip **pointer**).

```
proc nucleu
  const nmasc="cantitatea de memorie disponibila"
  enum stare_proces(activ,asteptare,term_corect,term_eroare)
  record bloc_procese(nr:int;stare:stare_proces;adresa:int;
                      registre:stare_registru;l1,l2:int)
  pointer acces_bloc(^bloc_proces)
  .....
  var vi:int;"indicator stiva pentru procedura principala"
      m:int,"cantitatea de memorie alocata curent"
      n:int;"numar procese create prin cobegin
  .....
  module "gestiune_memorie"
  .....
  module "gestiune_procese"
  .....
```

Fig 9.6. Structura nucleului descris ca program EDISON

9.3.2. Gestionarea memoriei

Varibilele accesibile la un moment dat dintr-un proces EDISON sînt:
- cele corespunzătoare contextului inițial [BH81b] al procesului; acestea se alocă în cursul rulării procedurii principale a programului și există în momentul executării unei instrucțiuni **cobegin** care dă naștere proceselor concurente; aceste variabile sînt

comune tutror proceselor;

- variabilele proprii unui proces: ele iau naștere la apelurile procedurilor dintr-un proces paralel și sînt accesibile în exclusivitate acestuia.

Prin urmare atît procedura principală cît și procesele paralele, create pe parcurs vor gestiona cîte o stivă de date.

Nucleului îi revine sarcina de a aloca fiecărui proces, la creare, o zonă de memorie (compactă) pentru stivă. După terminare tuturor proceselor paralele corespunzătoare unui **cobegin**, zonele de memorie proprii acestora se eliberează automat (§9.3.3), revenindu-se la vîrful stivei procedurii principale.

Cea mai simplă politică de stabilire a cantității de memorie ce îi revine unui proces este următoarea: împărțirea disponibilului de memorie din momentul executării instrucțiunii **cobegin**, în mod egal între procese (fig. 9.7).

```
*proc aloc(var proces:acces_bloc)
   var lim:int
   begin
      lim:=m+(mmax-vi)div n;
      proces^.l1:=m;
      proces^.l2:=lim;
      m:=lim
   end "aloc"
```

Fig. 9.7. Alocarea memoriei pentru un proces (varianta 1)

Această repartizare uniformă a memoriei, poate avea ca urmare o utilizare ineficientă a spațiului disponibil. O altă variantă este aceea în care cantitatea de memorie repartizată prin proces se transmite nucleul (fig. 9.8), pe baza unor informații obținute în faza de compilare. Ea rezultă din:

-constanta procesului, interpretată ca necesar de memorie pentru proces;

-calculul necesarului de memorie pentru un proces prin simulare, de către compilator; aceasta se poate efectua cu precizie pentru programe fără apeluri recursive de proceduri. În cazul prezenței unor apeluri recursive numărul maxim de recursiuni ar putea fi dat ca opțiune de compilare.

```
*proc aloc(var proces:acces_bloc;mem:int)
  begin
    if m+mem<=mmax do
      proces^.l1:=m;
      proces^.l2:=m+mem;
      m:=m+mem
    else true do ..... "eroare"
    end
  end "aloc"
```

Fig. 9.8. Alocarea memoriei pentru un proces (varianta 2)

9.3.3. Planificarea și sincronizarea proceselor și gestionarea unității centrale (UC)

Sincronizarea proceselor se realizează în limbaj prin instrucțiunea **when** (cap. 3, §3.4.1). Într-o implementare în care nu se tratează întreruperile externe, un proces poate fi întrerupt numai în cazul în care el execută o instrucțiune **when**, aceasta înseamnă că executarea unei zone critice ar putea fi întreruptă doar în cazul în care ar conține un alt **when**, ceea ce ar conduce automat la blocarea procesului. O astfel de suprapunere (nestiny) a executării instrucțiunilor **when** poate fi interzisă la faza de compilare. Cu această măsură, pe un sistem monoprocesor, zona critică descrisă printr-un **when** nu trebuie protejată în nucleu.

În cazul implementării cu întreruperi sau pe un sistem multiprocesor (§9.5), se

impune existența unui semafor unic pe prpogram care să asigure excluderea mutuală a fazelor critice, conform definiției instrucțiunii **when** (cap. 3, §3.4.1).

Pentru *planificarea proceselor și gestionarea UC* se poate utiliza următorul mecanism [CE8a]:

a) la debutul unei instrucțiuni **cobegin** se crează toate cele "n" procese descrise;

b) procesul "1" (primul în ordinea descrierii) devine activ (ocupă UC), iar celelalte "n-1" intră, în ordine, în șirul de așteptare la UC;

c) în momentul în care procesul activ execută o instrucțiune **when** cu toate condițiile false, va trece în coada șirului de așteptare la UC, urmînd a fi reluat cu începutul aceluiași **when**; tot odată UC se alocă primului proces din șirul de așteptare;

d) în cazul în care acțiunile unui proces s-au precizat, el părăsește definitiv UC, care urmează să se realoce ca la punctul precedent.

Mecanismul prezentat, proiectat pentru cazul unui sistem monoprocesor, este descris în modulul "*gestiune-procese* " (fig. 9.10).

```
module "gestiune_procese"
  const proces_max=...;"numarul maxim de procese"
        nedefinit=0
  array procese[1:proces_max](int)
  array blocuri[1:proces_max](acces_bloc)
  var proces_activ,contor_procese,primul,ultimul:int;
      sir_procese:procese;adrese_blocuri:blocuri
  *proc init_proces(constanta,adresa_cod:int)
    var tranzactie:acces_bloc
    begin
      rezerva(tranzactie);
      aloc(tranzactie,constanta);
      contor_procese:=contor_procese+1;
      tranzactie^.nr:=contor_procese;
      tranzactie^.adresa:=adresa_cod;
      adrese_blocuri[contor_procese]:=tranzactie;
      introduce(tranzactie^.nr);
      tranzactie.stare:=asteptare
    end "init_proces"
  *proc trece_asteptare(adresa_cod:int)
    var tranzactie:acces_bloc
    begin
      tranzactie:=adrese_blocuri[proces_activ];
      salv_reg(tranzactie^.registre);
      tranzactie^.adresa:=adresa_cod;
      introduce(tranzactie^:nr);
      tranzactie^:stare:=asteptare;
      proces_activ:=nedefinit
    end "trece_asteptare"
*proc lanseaza_proces(var adresa_cod:int)
  var tranzactie:acces_bloc;nr_proces:int
  begin
    if contor_procese=1 do nr_proces:=1;m:=vi
    else true do extrge(nr_proces) end;
    tranzactie:=adrese_blocuri[nr_proces];
    tranzactie^.stare:=activ;
    proces_activ:=tranzactie^.nr;
    adresa_cod:=tranzactie^.adresa;
    ref_reg(tranzactie^.registre)
  end "lanseaza_proces"
```

Fig. 9.10. Planificarea proceselor și gestionarea UC

```
proc init_program
  begin ... end
*proc term_proces
  var tranzactie_acces_bloc
  begin
    tranzactie:=adrese_blocuri[proces_activ];
    tranzactie^.stare:=term_corect;
    proces_activ:=nedefinit;
    contor_procese:=contor_procese-1;
    if contor_procese=0 do "term" end
  end "term_proces"
*proc intrerupe_proc_princ
  begin ... end
proc introduce(proces:int)
  begin
    sir_procese[ultimul]:=proces;
    ultimul:=ultimul mod proces_max+1
  end "introduce"
proc extrage(var proces:int)
  begin
    proces:=sir_procese[primul];
    primul:=primul mod proces_max+1
  end "extrage"
proc salv_reg(registre:stare_registre)
  begin ... end
proc ref_reg(var registre:sta e_registre)
  begin ... end
proc rezerva(pointer:acces_bloc)
  begin ... end
*proc elibereaza(pointer:acces_bloc)
  begin ... end
begin
  init_program;
  primul:=1;
  ultimul:=1
end
```

Fig. 9.10. (continuare)

Principalele operații care au fost implementate sînt următoarele:

a) Crearea celor "n" procese descrise de aceeași instrucțiune **cobegin**; se efectuează la debutul unei instrucțiuni **cobegin** apelînd de "n" ori procedura "init-proces"; crearea unui proces constă în rezolvarea și completarea tranzacției procesului, alocarea necesarului de memorie și introducerea procesului în coada șirului de așteptare la UC; în final, primul proces în ordinea descrierii devine activ ("lansează- proces").

b) Trecerea în așteptare a procesului activ, se efectuează prin apelul procedurii "trece—așteptare", în momentul în care procesul activ execută o instrucțiune **when** cu toate condițiile false; el va fi reluat ulterior cu începutul aceluiași **when**; UC se alocă primului proces din coada de așteptare ("lansează-proces").

c) Terminarea unui proces; se efectuează conform procedurii "term-proces", atunci cînd acțiunile unui proces s-au epuizat; procesul părăsește definitiv UC care urmează să se aloce primului proces din șirul de așteptare; în momentul în care s-au terminat toate procesele se relansează procedura principală a programului (procesul 1); se eliberează memoria utilizată de procese, inclusiv tranzacțiile (m:=vi și respectiv

apel "eliberează") și se continuă cu operațiile care urmează instrucțiunii **cobegin**.

9.3.4. Alte activități care pot intra în evidența nucleului

a) *Programarea perifericelor* se realizează de către programator prin intermediul unor proceduri standard care asigură accesul la registrele unităților [BH82]. Pentru a ușura activitatea de programare, nucleul asigură, de asemenea, funcțiile de citire și scriere caracter (pentru consolă, cititor de cartele și imprimantă), citire și scriere sector (disc flexibil) și citire și scriere bloc (bandă magnetică). Aceste funcții sînt apelabile din program ca proceduri de prefix [RA84].

b) *Tratarea întreruperilor.* Definirea limbajului nu face necesară tratarea întreruperilor de către nucleu. În măsura în care, în funcție de calculator, evenimentele externe pot fi sesizate prin testarea unor registre hardware (de ex. încheierea unor operații de I/E), tratarea acestora se poate realiza de către programator prin intermediul procedurii **sense** [EC84a].

c) *Gestionarea timpului* de către nucleu, cu ajutorul unor întreruperi de timp, ar putea fi motivată prin următoarele două aspecte:

- Introducerea în limbaj a unei proceduri standard pentru tratarea timpului real pe bază de întreruperi de timp. Aceasta nu s-a realizat pentru că tratarea timpului prin program se poate efectua ușor, prin intermediul procedurilor de I/E, asimilînd ceasul de timp real cu un periferic.

- Realizarea în nucleu a unei divizări de timp între procese, similară cu cea din PASCAL CONCURENT/FELIX C (cap. 8). Aceasta ar asigura o anumită echitabilitate în accesul la UC dar, pe lîngă încărcarea nucleului, are dezavantajul în traducerea unui nivel suplimentar de gestiune a proceselor, inaccesibil programatorului.

9.4. Sistemul de operare

În componența sistemului de programare (fig.9.1.) intră un minisistem de operare, scris în limbajul EDISON, ale cărui principale funcții sînt: dialogul cu utilizatorul, gestionarea memoriei externe și încărcarea programelor. Structura sistemului, prezentată în fig. 9.11., reflectă fidel funcțiile sale.

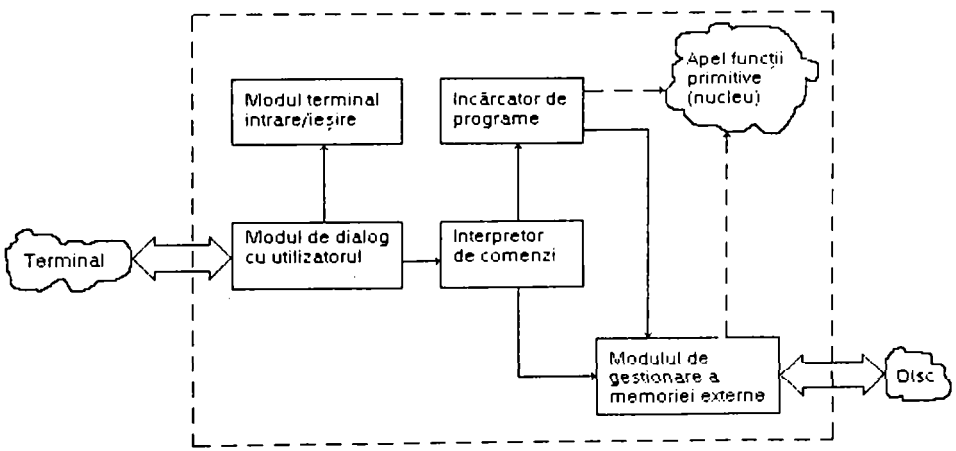


Fig. 9.11. Structura sistemului de operare

de operare. S-a avut, de asemenea, în vedere o împărțire optimă a sarcinilor între SO și executiv, astfel încît în sistem să se rețină doar funcții cu caracter mai general, nemodificabile de la un calculator la altul. În concluzie, portabilitatea sistemului de operare rezultă din mai multe aspecte, și anume:

- portabilitatea întregului sistem de programare;
- scrierea lui în limbaj de nivel înalt (EDISON);
- generalitatea funcțiilor implementate și apelul la executiv pentru acele funcții dependente de sistemul de calcul.

În continuare se parcurg principalele activități implementate în sistemul de operare [CI85]:

a) *Dialogul cu utilizatorul* este realizat prin intermediul unui limbaj de comandă. Comenzile sînt interpretate și au drept efect selectarea și apelul unor proceduri din sistem care sînt capabile să continue dialogul și, în final, să efectueze funcția solicitată. În cadrul acestor proceduri s-au prevăzut secvențe de dialog pentru toate eventualitățile imaginate de apariție a unor erori, pentru cazurile în care utilizatorul poate să facă unele opțiuni etc. Datele furnizate de utilizator se transmit ca parametri procedurilor de sistem.

b) *Gestionarea memoriei externe.* Memoria externă este reprezentată printr-o unitate dublă de discuri flexibile de 8 inch, cu o capacitate de aproximativ 250Koct/disc. Întreaga memorie externă este organizată sub forma unei grupări de fișiere identice, descrisă printr-un fișier catalog în care se reține numele fișierelor și o serie de atribute referitoare la acestea (lungime, protecție, etc.). În ceea ce privește gestiunea propriu-zisă, s-au prevăzut următoarele activități:

- gestionarea discurilor active la un moment dat și autorizarea înlocuirii unui disc;
- gestionarea și alocarea discului organizat în pagini și sectoare;
- crearea, completarea și actualizarea catalogului;
- crearea, gestionarea și protejarea fișierelor;
- furnizarea unor proceduri pentru realizarea operațiilor pe disc solicitate de utilizator; utilizatorul are acces la disc exclusiv prin intermediul acestor proceduri, realizîndu-se astfel și protecția fișierelor de pe disc.

c) *Încărcarea programelor* solicitate de utilizator și asistarea lor pe parcursul rulării, reprezintă una din funcțiile importante ale SO. Pentru realizarea acestei funcții se inițiază un dialog la terminal care permite sistemului să identifice biblioteca și să localizeze programul în cadrul bibliotecii. În cazul în care aceste operații reușesc, se declanșează încărcarea în memorie a codului programului cerut.

9.5. Posibilități de extindere a executivului EDISON pe sisteme multiprocesor

Progresele tehnologice deosebite realizate în ultimii ani în domeniul integrării pe scară largă și al structurilor de interconectare au permis realizarea unor sisteme distribuite de cele mai diverse tipuri. Ele presupun rularea unor programe ale căror module sînt localizate în unități de calcul distincte, cu posibilități de intercomunicare.

Obiectivele avute în vedere la realizarea unor sisteme distribuite sînt, în general, următoarele [LA81]:

- creșterea performanțelor (creșterea numărului elementelor procesoare);
- extensibilitatea (adaptarea configurației în scopul modificării performanțelor sau a funcționalității);
- îmbunătățirea disponibilității (redundanță fizică și logică);
- exploatarea maximă a resurselor.

În funcție de arhitectura lor, sistemele bazate pe cooperarea mai multor procesoare prezintă o gamă largă de structuri, cu deosebiri esențiale în ceea ce privește funcționarea și programarea [GE82], categoriile extreme fiind:

a) sisteme cu mai multe unități aritmetice - logice și memorie comună;

b) rețele de calculatoare în care fiecare procesor are acces doar la memoria proprie.

Un limbaj implementat pe un asemenea sistem va permite exploatarea optimă a avantajelor sale potențiale, în măsura în care realizează o distribuire corespunzătoare și la nivel logic. Din acest punct de vedere trebuie luate în considerare următoarele caracteristici ale unui limbaj:

- descrierea unor activități (procese) paralele;
- posibilități de adaptare la diferite structuri fizice ale sistemului;
- realizarea unui control cît mai descentralizat al proceselor.

Datorită, în special, acestui din urmă aspect, utilizarea unei memorii comune ca mijloc de sincronizare sau/și comunicarea între procese, poate fi nerecomandabilă. Prezența în memoria comună a unor entități (blocuri de control, fanioane) care centralizează starea proceselor la un moment dat sau influențează funcționarea lor, afectează eficiența sistemului.

Sistemele multiprocesoare cu memorie comună sînt totuși relativ răspîndite, ele stînd la baza unui număr mare de experimente și aplicații [JS80]. Din această cauză, adăugînd și unele particularități ale limbajului, se consideră că EDISON este potrivit pentru programarea sistemelor multiprocesor [BH81a, EC85c].

In EDISON, comunicarea între procese se realizează prin variabile comune (cap. 3, §3.4.1). Sincronizarea proceselor (prin zone critice condiționale) presupune existența unui fanion unic, accesibil tuturor proceselor (§9.5.2). Implementarea pe un sistem multiprocesor cu memorie comună este recomandabilă avînd în vedere cîteva caracteristici esențiale ale limbajului, care permit realizarea unui nucleu restrins, simplu și în mare parte transparent ca interfață între programator și sistem [BH81b]:

- sincronizarea prin instrucțiunea **when**;
- crearea și desființarea controlată a proceselor la începutul și respectiv la sfîrșitul instrucțiunilor **cobegin**;
- realizarea operațiilor de I/E integral în limbajul de nivel înalt.

Arhitectura sistemului multiprocesor luat în considerare pentru implementare este prezentată în fig. 9.12. Sistemul este compus dintr-un număr de procesoare identice, legate la memoria comună (MC) printr-o magistrală (MgC). Fiecare procesor are, de asemenea, o memorie proprie (MP). Aceasta este accesibilă și din celelalte procesoare prin intermediul MgC. Accesele procesoarelor la magistrala comună se exclud mutual.

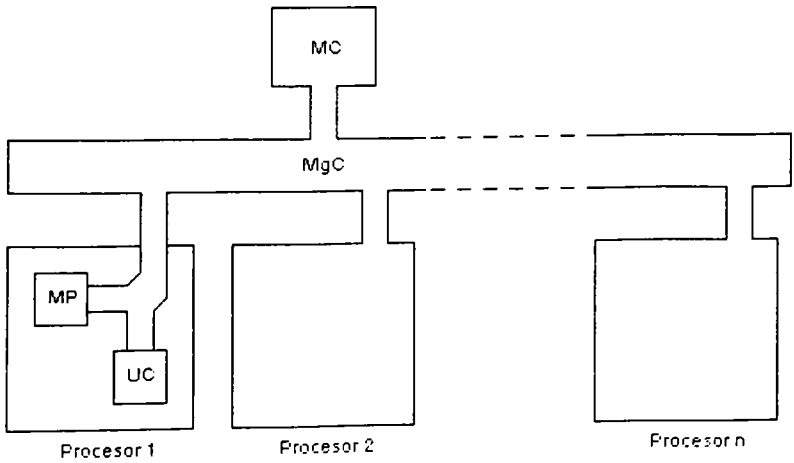


Fig. 9.12. Structura sistemului multiprocesor

In continuare se prezintă cele două aspecte esențiale ale implementării limbajului pe un astfel de sistem: gestionarea memoriei și sincronizarea proceselor.

9.5.1. Gestionarea memoriei

Repartizarea datelor și a codului în memorie se face pornind de la următoarele considerente:

- numărul acceselor la memoria comună, prin intermediul MgC, trebuie redus la minimum, datorită faptului că ele frinează sistemul;
- pentru limbaje de nivel înalt s-a constatat următoarea distribuție statistică a referirilor la memorie [JO80]:
  a) referiri la cod: 50%
  b) referiri la date locale: 40%
  c) referiri la date comune: 10%;
- variabilele unui program EDISON se împart în comune (accesibile tuturor proceselor ce rulează în paralel, la un moment dat) și proprii (accesibile exclusiv unui proces).

Prin urmare, doar variabilele comune vor fi alocate în MC. Pactic, aceasta înseamnă dispunerea în MC a stivei corespunzătoare procedurii principale. Stivele proceselor paralele se alocă în întregime în memoriile proprii, asociate lor.

Codul executat de către un procesor va fi încărcat în întregime în MP corespunzătoare. Aceasta presupune prezența în fiecare MP a interpretatorului și a rutinelor de sincronizare (nucleu). Codul obiect al procedurilor EDISON apelate din mai multe procese paralele, va fi multiplicat pentru fiecare proces. In acest sens, trebuie determinată la compilare mulțimea tuturor procedurilor apelate de către un anumit proces. Incărcătorul executivului (§9.3.1.) va asigura încărcarea în MP corespunzătoare unui procesor a codului executat de toate procesele care vor fi asociate, pe parcurs, procesorului.

9.5.2. Sincronizarea proceselor

Avînd în vedere structura sistemului multiprocesor din figura 9.12., la începutul fiecărei instrucțiuni **cobegin** se va realiza asocierea: un proces - un procesor. Această operație se poate face automat sau pe baza unor indicații date de programator (constanta procesului). Astfel unitățile centrale devin resurse proprii fiecărui proces, eliminîndu-se faza de așteptare la acest tip de resursă precum și gestionarea UC, tipică și esențială pe sistemele monoprocesor. In concluzie, un proces poate fi la un moment dat sau activ sau în faza de sincronizare a unei instrucțiuni **when**.

Spre deosebire de cazul tratat în §9.3.3., la implementarea pe un sistem multiprocesor se impune existența unui semafor unic pe program care să asigure excluderea mutuală a fazelor critice, conform definiției instrucțiunii **when** (cap. 3. §3.4.1.). O primă variantă de implementare prezentată în modulul "sincro_1" (fig. 9.13.), consideră semaforul plasat în MC fiind astfel accesibil în egală măsură tuturor proceselor.

In momentul în care un proces îndeplinește condițiile proprii pentru a executa o zonă critică, urmează să solicite acest lucru apelînd procedura "ocupă". In această procedură, procesul ciclează pînă cînd semaforul general devine liber. Testarea semaforului și ocuparea lui sînt realizate într-o operație indivizibilă (interschimb), ceea ce asigură și protecția semaforului împotriva accesului simultan din mai multe procese (alături de procesele hardware ale MgC). Această variantă are dezavantajul că fiecare testare a semaforului necesită acces la memoria comună, prin intermediul magistralei.

O optimizare deosebită a performanțelor implementării se realizează eliminînd din ciclu accesul la MC. In acest caz, procesele care solicită eliberarea semaforului sînt

puse în așteptarea unei întreruperi. Procedura "ocupă" este prezentată în fig. 9.14.

```
enum stare(liber,ocupat)
var semafor_general:stare "rezervat in MC"
module "sincro_1"
  *proc ocupa
    var semafor_local:stare "rezervat in MP"
    begin
      semafor_local:=ocupat;
      while semafor_local=ocupat do
        semafor_general:=:semafor_local
      end
    end "ocupa"
  *proc elibereaza
    begin
      semafor_general:=liber
    end "elibereaza"
  begin
    semafor_general:=liber
  end "sinc. _1"
```

Fig. 9.13. Sincronizarea proceselor printr-un semafor general (varianta 1)

```
proc ocupa
  var semafor_local:stare "rezervat in MP"
  begin
    semafor_local:=ocupat;
    semafor_general:=:semafor_local;
    while semafor_local=ocupat do
      while not intrerupere do skip end;
      semafor_general:=:semafor_local
    end
  end "ocupa"
```

Fig. 9.14. Sincronizarea proceselor printr-un semafor general (varianta 2)

Generarea întreruperii trebuie realizată de procesul care termină o zonă critică, după ce a poziționat semaforul general pe liber (apel "eliberează").

In concluzie funcțiile de sincronizare sînt deosebit de simple. Gestionarea proceselor este practic absentă în cazul asocierii "un proces - un procesor". Astfel, se poate obține o implementare eficientă a limbajului EDISON pe un sistem multiprocesor cu memorie comună redusă și memorii de dimensiuni mari distribuite procesoarelor.

CONCLUZII

1. Contribuţii originale

Lucrarea reprezintă sinteza cercetărilor şi realizărilor autorului în domeniul limbajelor de programare, într-o perioadă de aproximativ 12 ani. Pe parcursul lucrării se pot scoate în evidenţă următoarele contribuţii originale:

Capitolul 1:

- O sistematizare a conceptelor de abstractizare în programare; introducerea şi definirea noţiunilor de *etapă* şi *nivel de abstractizare* precum şi a celor de *abstractizare logică* şi *fizică*.
- O sinteză comparativă privind conceptele de valoare şi obiect în limbajele de programare; prezentarea elementelor definitorii şi a particularităţilor programării orientate spre obiecte.
- O amplă analiză critică a evoluţiei conceptului de dată abstractă; introducerea şi definirea noţiunilor de *abstractizare statică* şi *dinamică*.
- Extinderea metodei de proiectare-programare top-down pentru limbaje de programare incluzînd date abstracte.

Capitolul 2:

- Definirea unor criterii de evaluare a eficienţei metodelor pentru descrierea formală a limbajelor de programare.
- Sistematizarea şi extinderea clasificării metodelor de definire formală a limbajelor de programare din literatura de specialitate obţinînd, în final, atît o sintetizare cît şi o clasificare, originale.
- Definirea modelui algebric al unui limbaj de programare bazat pe o ierarhie de tipuri de date abstracte incluzînd atît reprezentarea sintaxei cît şi semanticii limbajului.
- Analiza comparativă a metodelor de definire formală a limbajelor de programare.

Capitolul 3:

- Prezentarea şi analiza facilităţilor de exprimare a concurenţei proceselor, proprii limbajelor de nivel înalt.

Capitolul 4:

- Stabilirea relaţiei între *programarea concurentă* şi cea în *timp real* pe baza elementelor definitorii ale fiecăreia.
- Definirea unor cerinţe impuse limbajelor de nivel înalt destinate domeniului programării în timp real.
- Analiza impactului întreruperilor asupra programării în timp real.
- Stabilirea unui sistem de priorităţi ale proceselor original, în limbajul Pascal Concurent/Felix C care să favorizeze programarea în timp real.
- Elaborarea unei metodologii de realizare a programelor în timp real specifică limbajelor de nivel înalt concurente.
- Modificarea modului de lucru şi al efectului rutinelor standard *wait* şi *realtime* din Concurrent Pascal, în vederea măririi preciziei de contorizare a timpului.

Capitolul 5:

- Compararea mecanismelor pentru controlul vizibilităţii identificatorilor din Pascal, Concurrent Pascal şi Edison.
- Modelarea formală a principalelor aspecte legate de controlul vizibilităţii identificatorilor şi evaluarea, pe baza ei, a mecanismelor proprii diferitelor limbaje de programare.
- Elaborarea unor algoritmi originali pentru analiza de domeniu.
- Elaborarea unui mecanism original pentru tratarea la compilare şi la execuţie a parametrilor funcţii şi proceduri implementat în compilatorul Pascal/Felix C.

Capitolul 6:

- Elaborarea unor algoritmi pentru tratarea la compilare a tipurilor şi variabilelor

specifice limbajului Concurrent Pascal: date abstracte și locații de așteptare.

- Proiectarea unor algoritmi pentru tratarea la compilare a procedurilor, funcțiilor și variabilelor externe precum și a programelor secvențiale apelate sau lansate dintr-un program Pascal Concurent.

Capitolul 7:

- Structura compilatorului Pascal/Felix C.
- Realizarea unui set de subprograme pentru implementarea în limbaj de asamblare a unui analizor sintactic recursiv de tip predictiv.
- Proiectarea și realizarea unui generator de cod obiect relocabil (BT) aplicat la compilatoarele Pascal și Pascal Concurent/Felix C.
- Proiectarea unui sistem de compilare interactivă a programelor Pascal și Pascal Concurent, bazat pe compilatoarele neinteractive existente.

Capitolul 8:

- Structura și organizarea nucleului Pascal Concurent/Felix C.
- Structura tranzacției unui proces în conformitate cu specificul limbajului de programare Pascal Concurent/Felix C.
- Adaptarea pentru Felix C a mecanismului de creare și gestionare a proceselor din implementarea Concurrent Pascal pe PDP-11.
- Implementarea conceptului de monitor.
- Organizarea memoriei la execuția unui program Pascal Concurent.
- Proiectarea și implementarea unor mecanisme originale de compilare separată pentru Pascal și Pascal Concurent: apel de subprograme scrise în alte limbaje, apel de programe Pascal compilate separat, adaptarea sistemului de segmentare specific calculatorului Felix C.
- Proiectarea și implementarea unui mecanism original de lansare din Pascal Concurent a programelor obiect secvențiale.

Capitolul 9:

- Structura unui sistem de programare portabil pentru limbajul Edison.
- Definirea unui cod virtual optim, utilizat la implementarea cu mașină virtuală a limbajului Edison.
- Organizarea executivului și nucleului pentru sistemul de programare Edison.
- Elaborarea unor algoritmi pentru gestionarea memoriei, planificarea și sincronizarea proceselor și gestionarea unității centrale, aplicații în nucleul Edison.
- Elaborarea structurii sistemului de operare propriu mediului de programare Edison prin adaptarea sistemului de operare MONO realizat pentru PDP-11.
- Propunere de extindere a executivului Edison pe sisteme multiprocesor.

2. Valorificarea cercetărilor și direcții posibile de continuare

In afara rezultatelor teoretice scoase în evidență în paragraful precedent, cercetările prezentate în această lucrare s-au materializat și în numeroase rezultate practice: compilatoare și executive pentru limbajele de programare Pascal și Pascal Concurent/Felix C, în trei versiuni, realizate în perioada 1977-1985; părți ale unui mediu de programare Edison: compilator, executiv, editor și sistem de operare propriu, etc., implementate pe diferite mini și microcalculatoare (Independent, Felix M 18, Junior, compatibil IBM PC), realizate în perioada 1984-1989.

Aceste rezultate au fost valorificate în 73 lucrări științifice și didactice publicate și comunicate de autor, singur sau în colaborare, în 11 certificate de inovator și 14 contracte de cercetare coordonate de autor, în valoare de aproape 5 milioane lei.

Desigur, cercetările efectuate nu sînt închise. Ele pot fi continuate în aproape toate direcțiile semnalate în paragraful precedent. Astfel, modelarea formală completă a limbajelor de programare a avut ca scop inițial elaborarea unei metode practice de realizare a compilatorului după acest model formal, obiectiv nefinalizat de autor. De asemenea, formalizarea aspectelor legate de vizibilitatea identificatorilor nu cuprind, în întregime, problematica specifică datelor abstracte și programării

orientate spre obiecte.

In ceea ce privește programarea concurentă, comunicarea prin mesaje a proceselor este doar semnalată și caracterizată, fără a se prezenta soluții concrete de implementare. De asemenea, nu s-au abordat decît tangențial limbajele destinate programării sistemelor distribuite și problemele specifice rețelelor de calculatoare

BIBLIOGRAFIE

[AG86] - AGHA G. - "An overview of Actor languages", SIGPLAN Notices, V21, No. 10, october, 1986

[AH79] - AHO A. V., HOPCROFT J. E., ULLMAN J. D. - "Data structures and algorithms", Addison-Wesley, Reading Mass, 1979.

[AH86] - AHO A. V., ș.a. - "Compilers. Principles, techniques, and tools", Addison-Wessley Publishing Company, 1986

[AL79] - ALLEN J. - "The anatomy of LISP", Mac Graw Hill, New York, 1979

[AM77] - AMMANN U. - "On code generation in a Pascal Compiler", Softw. Pract. and Exper., vol. 7, 1977

[AN78] - ANDLER S. - "Synchronization primitives and the verification of cocurrent programs", IRIA and CMU, Symp. of operating syst., Rocquencourt, 1978

[AN79] - ANSI - "Specificarion for the Computer programming language Pascal", I.S.O., 1979

[AN83] - ANSI - "The programming language Ada", Reference Manual, ANSI/MIL-STD-1815A, Springer-Verlag, 1983

[AR84] - APPELBE W.F., ROWN A.P. - "Encapsulation constructs in systems programming languages", ACM Trans. on Prog. Lang. and Syst., V6, N2, 1984

[AS83] - ANDREWS G.R., SCHNEIDER F.B. - "Concepts and notations for concurrent programming", ACM Computing Surveys, V15, N1, 1983

[AU78] - AHO A.V., ULLMAN J.D. - "Principles of Compiler design", Addison-Wesley, 1978

[BA71] - BALZER R.M. - "PORTS-a method for dynamic interprogram communication and job control", AFIPS Press, V38, Arlington, 1971

[BA74] - BALTAC V., ș.a. - "Felix C-256. Structura și programarea calculatorului", Ed. Tehnică București, 1974

[BA80] - BARNES J.G.P. - "The standardization of RTL/2", Softw. Pract. and Exper., V10, 1980

[BA82] - BARNES J.G.P. - "Programming in Ada", Addison Wesley, 1982

[BD85] - BABUTIA I, DRAGOMIR T.L., MURESAN I., PROSTEAN O. - "Conducerea automată a proceselor", Ed. Facla, 1985

[BE80] - BERRY D.M., ș.a. - "Toward modular verifiable exception handling", Comp. Lang., V5, 1980

[BH70] - BRINCH HANSEN P. - "The nucleus of multiprogramming system", Comm. of ACM, 13(4), 1970

[BH72] - BRINCH HANSEN P. - "Structured multiprogramming", Comm. of ACM, V15, No. 7, 1972

[BH73a] - BRINCH HANSEN P. - "Concurrent programming concepts", ACM Computing Surveys, V5, N4, 1973

[BH73b] - BRINCH HANSEN P. - "Operating system principles", Prentice Hall, Englewood Cliffs, 1973

[BH75a] - BRINCH HANSEN P. - "Concurrent Pascal Report", Inform. Science Calif. Inst. of Techn., Pasadena, 1975

[BH75b] - BRINCH HANSEN P. - "Concurrent Pascal Introduction", Inform. Science Calif. Inst. of Techn., Pasadena, 1975

[BH75c] - BRINCH HANSEN P. - "The Programming Language Concurrent Pascal", IEEE Trans. on Softw. Eng., V1, N2, 1975

[BH76a] - BRINCH HANSEN P. - "The Solo operating system: processes, monitors and classes", Softw. Pract. and Experience, V6, N2, 1976

[BH76b] - BRINCH HANSEN P. - "The Solo operating system: a Concurrent Pascal program", Softw. Pract. and Experience, V6, N2, 1976

[BH76c]- BRINCH HANSEN P. - "The Solo operating system: job interface", Softw. Pract. and Experience, V6, N2, 1976

[BH77a] - BRINCH HANSEN P. - "The architecture of concurrent programs",

Prentice-Hall, Englewood Cliffs, 1977
[BH77b] - BRINCH HANSEN P. - "Network: a multiprocessor program", IEEE Comp. Softw. and Appl. Conf., Chicago, 1977
[BH77c] - BRINCH HANSEN P. - "Experience with modular concurrent programming", IEEE Trans. on Softw. Eng., V3, N2, 1977
[BH78] - BRINCH HANSEN P. - "Distributed processes: a concurrent programming concept", Comm. of ACM, V21, N11, 1978
[BH81a] - BRINCH HANSEN P. - "Edison-a multiprocessor language", Softw. Pract. and Exper., V11, N4, 1981
[BH81b] - BRINCH HANSEN P. - "The design of Edison", Softw. Pract. and Exper., V11, N4, 1981
[BH81c] - BRINCH HANSEN P. - "Edison programs", Softw. Pract. and Exper., V11, N4, 1981
[BH82] - BRINCH HANSEN P. - "programming a personal computer", Prentice-Hall, Englewood Cliffs, 1982
[BL70] - BIRKHOFF G., LIPSON J. D. - "Heterogeneous algebras ", J. Comb. Theor., No. 8, 1970
[BN84] - BIRELL A. D., NELSON B. J. - "Implementing remote procedure calls", ACMTrans. on Comp. Syst., V2, N1, 1984
[BO85] - BOBROW D. K., ş.a. - "Commonloops: merging Common Lisp and object oriented programming", OOPSLA.1986
[BO86] - BOOCH G. - "Object-oriented development", IEEE Trans. on Softw. Eng., V12, N2, 1986
[BR83] - BROOKES S. D., HOARE C. A. R., ROSCOE A. W. - "A theory of communicating sequential processes", Carnegie-Mellon Univ., CMU-CS-83-153, Pittsburgh, 1983
[BR87] - BROY M., ş.a. - "On the algebraic definition of programming languages", ACM, Trans. on Progr. lang. and Syst., V9, No. 1, 1987
[CC80] - CIOCARLIE H., CIOCIRLIE R., ş.a. - "Limbajul Pascal Concurent pentru calculatorul FELIX", Versiunea 1, Manual de utilizare, CCE IPT, 1980
[CC81] - CIOCÂRLIE H., CIOCÂRLIE R. - "Implementarea conceptului de concurenţa la compilatorul Pascal Concurent realizat pentru calculatorul FELIX C-256", vol.-Programarea calculatoarelor. Studii Si aplicaţii-, Editura Facla, Timişoara, 1981
[CE79] - CIOCÂRLIE H., ELES P., - "Aspecte specifice ale compilării unui program Pascal (secvenţial)", al V-lea Simp. "Informatică şi conducere", Cluj-Napoca, 1979
[CE80] - CIOCÂRLIE H., ELES P. - "LimbajulPascal (secvenţial) pentru calculatorul FELIX", Consf. cadrelor de la centrele de calcul din învăţământul superior. Cluj-Napoca, 1980
[CE81a] - CIOCÂRLIE H., ELES P., ş.a. - "Limbajul Pascal pentru calculatorul FELIX", Manual de utilizare, CCE IPT, 1981
[CE81b] - CIOCÂRLIE H., ELES P., ş.a. - "Limbajele Pascal pentru calculatoarele din gama FELIX", Buletinul de informatică al ASE, Bucureşti, 1981
[CE81c] - CIOCÂRLIE H., ELES P. - "Limbajul Pascal concurent-instrument de programare concurentă, conversaţională şi în timp real pentru calculatoarele din gama FELIX", The 4th I. C. C. S. C. S., V4, Bucureşti, 1981
[CE81d] - CIOCÂRLIE H., ELES P., ş.a. - "Compilator pentru limbajul Pascal (secvenţial)", al VII-lea Simpozion "Informatică şi conducere", Cluj-Napoca, 1981
[CE82] - CIOCÂRLIE H., ELES P. - "Probleme de perspectivă privind limbajele Pascal şi pascal Concurent", Consfătuirea cadrelor de la centrele de calcul din învăţământul superior , Timişoara, 1982
[CE83a] - CIOCÂRLIE H., ELES P., ş.a. - "Limbajul Pascal şi Pasal Concurent pentru calculatorul FELIX", Versiunea 2, Manual de utilizare şi operare, Raport CCE IPT, 1983
[CE83b] - CIOCÂRLIE H., ELES P., ş.a. - "Limbajul Pascal (secvenţial) şi Pascal Concurent pentru calculatoarele FELIX C-256, 512", sesiunea "Utilizarea calculatoarelor

în industrie", Timişoara, 1983
[CE84a] - CIOCÂRLIE H., ELES P., ş.a. - "Run time support for the programming language Edison", Buletinul ştiinţific şi tehnic al I.P.Timişoara, Tom 29, Fascicola 1-2, 1984
[CE84b] - CIOCÂRLIE H., ELES P., ş.a. - "Limbajele Pascal şi Pascal Concurent pentru calculatorul FELIX", Versiunea 3, Manual de utilizare şi operare, Raport CCE IPT, 1984
[CE84c] - CIOCÂRLIE H., ELES P. - "Aspecte privind perfecţionarea limbajelor Pascal şi Pascal Concurent pentru calculatoarele din gama FELIX C", Sesiunea "Tehnic 2000", Timişoara, 1982
[CE84d] - CIOCÂRLIE H., ELES P. - "Definirea codului virtual pentru implmentarea unor limbaje de programare de nivel înalt", CNETAC, Bucureşti, 1984
[CE84e] - CIOCÂRLIE H., ELES P. - "Studiu comparativ privind implementarea unor algoritmi pentru analiza de domeniu", Simpozionul naţional de Teoria Sistemelor, Craiova, 1984
[CE85a] - CIOCÂRLIE H., ELES P., BALLA I. - "Limbajele de programare Pascal şi Pascal Concurent",Editura Facla, Timişoara, 1985
[CE85b] - CIOCÂRLIE H., ELES P. - "Programarea structurată şi concurentă în limbajul Pascal", AMC, vol. 51, Editura Tehnică, Bucureşti, 1985
[CE85c] - CIOCÂRLIE H., ELES P. - "Gestionarea timpului real intr-un program Pascal Concurent", Sessiunea "Tehnic 2000", Timişoara, 1985
[CE85d] - CIOCÂRLIE H., ELES P. - "Interacţiunea proceselor într-un program concurent", Sesiunea "Tehnic 2000", Timişoara, 1985
[CE86a] - CIOCÂRLIE H., ELES P. - "Realizarea sistemelor în timp real pentru supravegherea şi conducerea proceselor industriale, în limbaje concurente de nivel înalt", Sesiunea "Tehnic 2000", Timişoara, 1986
[CE86b] - CIOCÂRLIE H., ELES P. - "Maşină virtuală pentru implementarea unui limbaj concurent de nivel înalt", Buletinul Simpozionului Naţional de Teoria Sistemelor, Craiova, 1986
[CE87a] - CIOCÂRLIE H., ELES P. - "Conceptul de dată abstractă. Aspecte privind definirea şi implementarea sa în limbaje de programare de nivel înalt", Sesiunea "Tehnic 2000", Timişoara, 1987
[CE87b] - CIOCÂRLIE H., ELES P. - "Aspecte specifice implementării conceptelor de programare în timp real In limbaje de nivel înalt", The 7-th I.C.C.S.C.S., Bucureşti, 1987
[CE88a] - CIOCÂRLIE H., ELES P. - "Etape şi nivele de abstractizare în limbajele de programare actuale", Sesiunea "Tehnic 2000", Timişoara, 1988
[CE88b] - CIOCÂRLIE H., ELES P. - "Proiectarea şi realizarea compilatoarelor pe baza metodelor de descriere formală a limbajelor de programare", Lucrările Simpozionului Naţional de calculatoare şi conducere automată a proceselor, Timişoara, dec. 1988
[CE89a] - CIOCÂRLIE H., ELES P. - "Valori şi obiecte în limbajele de programare actuale", ACM, Editura Tehnică, în curs de apariţie
[CE89b] - CIOCÂRLIE H., ELES P. - "Definirea formală a limbajelor de programare. Metode algebrice", Sesiunea "Tehnic 2000", Timişoara, 1989
[CH59] - CHOMSKY N. - "On certain formal properties of grammars", Information and Control No. 2, 1959
[CI75] - CII - "Normes de programation", CII, Manual d'utilization et d'operation, mai 1975
[CI76] - CII - "Moniteur SIRIS-3", CII, 1976
[CI78a] - CIOCÂRLIE H., - "Structuri de date în limbajul Pascal", Consfătuirea cadrelor de la centrele de calcul din învăţământul superior, Gura Humorului, 1978
[CI78b] - CIOCÂRLIE H., ş.a. - "Analizor sintactic şi de domeniu pentru un compilator Pascal Concurent", al IV-lea Simpozion "Informatică şi conducere", Cluj-Napoca, 1978

[CI79a] - CIOCÂRLIE H., ș.a. - "Pachet de programe pentru realizarea programării concurente", Raport CCE IPT, 1979
[CI79b] - CIOCÂRLIE H., ș.a. - "Compilator pentru limbajul Pascal Concurent. Documentația de realizare", Raport CCE IPT, 1979
[CI79c] - CIOCÂRLIE H., ș.a. - "Analiza declarațiilor și a instrucțiunilor într-un compilator Pascal Concurent", Sesiunea de comunicări I.C.I., 1979
[CI79d] - CIOCÂRLIE H., ș.a. - "Limbajul Pascal Concurent pentru calculatorul FELIX C", Sesiunea de comunicări a cadrelor didactice I.P. Timișoara, 1979
[CI79e] - CIOCÂRLIE H. - "Implementarea concurenței dintre procesele unui program Pascal Concurent", Sesiunea de comunicări I.C.I. 1979
[CI80a] - CIOCÂRLIE H., ș.a. - "Facilitați de programare în limbajul Pascal Concurent", Progrese în informatica romanească, Cluj-Napoca, 1980
[CI80b] - CIOCÂRLIE H. - "Sincronizarea proceselor monitoarelor și claselor dintr-un program Pascal Concurent", Progrese în informatica romanească vol. I-II, Cluj-Napoca, 1980
[CI80c] - CIOCÂRLIE H., ș.a. - "Compilator pentru limbajul Pascal Concurent", al VI-lea Simpozion "Informatică și conducere", Cluj-Napoca, 1980
[CI80d] - CIOCÂRLIE H., ș.a. - "Analiza semantică și generarea codului obiect într-un compilator pentru limbajul Pascal Concurent", Progrese în informatica romanească, vol. I-II, Cluj-Napoca, 1980
[CI81a] - CIOCÂRLIE H., ș.a. - "Programarea structurată și concurentă în limbajele Pascal pe calculatoarele FELIX", Colocviul de cibernetică, Timișoara, 1981
[CI81b] - CIOCÂRLIE H., ș.a. - "Compilator pentru limbajul Pascal (secvențial)", Documentația de realizare Raport CCE IPT, 1981
[CI81c] - CIOCÂRLIE H., ș.a. - "Sistem de lansare conversațională și în timp real, din Pascal Concurent a programelor IMT secvențiale", al VI-lea Simpozion "Informatică și conducere", Cluj-Napoca, 191
[CI82] - CIOCÂRLIE H., ș.a. - "Compilatoare Pascal și Pascal Concurent conversaționale", Lucrările CNETAC, București, 1982
[CI83] - CIOCÂRLIE H. - "On scope analysis in a compiler for Pascal-type languages", Buletinul științific și tehnic al IPT, Tom 27(41),1983
[CI84a] - CIOCÂRLIE H. - "Realizarea conceptelor de multiprogramare Si de timp real In limbaje de nivel înalt", referat de doctorat, Timișoara, 1984
[CI84b] - CIOCÂRLIE H. - "Proiectarea și realizarea compilatoarelor pentru limbaje structurate de tip Pascal", referat de doctorat, Timișoara, 1984
[CI85] - CIOCÂRLIE H., ș.a. - "The structure of a portable operating system for personal computers", Simpozionul "Microprocesoare, microcalculatoare și aplicații în economie", Timișoara, 1985
[CK78] - CHANG E., KADEN N., ELLIOTT W. D. - "Abstract data types in Euclid", ACM SigplanNotices, V13, N13, 1978
[CK80] - CAMPBELL R. H., KOLSTAD R. B. - "An overwiew of Path Pascal's design: Path Pascal user manual", ACM SIGPLAN Notices, V15, N9, 1980
[CM84] - CLOCKSIN W. F., MELLSIH C. S. - "Programming in Prolog", Springer Verlag, 1984
[CR84] - CRETU V. - "Sistem de operare timp real pentru sisteme de calcul cu multiprocesor", teză de doctorat, I.P.T., 1984
[CR87] - CRETU V. - "Structuri de date și tehnici de programare", vol. 1, curs litografiat, I.P.Timișoara, 1987
[DA83] - DAVIDOVICI A., ș.a. - "Minicalculatoarele și microcalculatoarele în conducerea proceselor industriale", EdituraTehnică București, 1983
[DB86] - DAVIDOVICI A., BARBAT B. - "Limbaje de programare pentru sisteme în timp real", Editura Tehnică București, 1986
[DD72] - DAHL O.J., DIJKSTRA E.W., HOARE C.A.R. - "Structured programming", Academic Press, London, 1972
[DD78] - Department of Defense - "Steelman requirements for DoD high order

computer programming languages", U.S.DoD, Washington D.C., 1979
[DD80a] - Department of Defense - "Steelman requirements for Ada programming support environments", U.S.DoD, Washington D.C., 1980
[DD80b] - Department of Defense - "The progamming language Ada", Reference manual, Springer Verlag, 1980
[DH80] - DUNCAN A.G., HUTCHISON J.S. - "Using Ada for industrial embeddead microprocessor application", ACM Symp on Ada, 1980
[DI68] - DIJKSTRA E.W. - "Cooperating sequential processes. Progamming languages", Academic Press, New-York,1968
[Di71] - DIJKSTRA E.W. - "Hierarchical ordering of sequential processes", Acta Informatica , V1, 1971
[Dm68] - DAHL O.J., MYHRHAUG B., NYGAARD H. - "The Simula 67 common base language", Publication no. s-2, NCC, Oslo, 1968
[DR82] - DRUFFEL L.E. - "The potential effect of ADA on software engineering in the 1980's", ACM Softw. Eng. Notes, V7, N3, 1982
[EC81] - ELES P., CIOCÂRLIE H. - "Alocarea dinamică a memoriei la compilatorul PASCAL secvențial pentru calculatorul FELIX", vol. Informatica pentru conducere - Realizări și aplicații, Cluj-Napoca, 1981
[EC82] - ELES P., CIOCÂRLIE H. - "Strucura și funcțiile unui sistem de programare concurentă pentru limbajul EDISON", CNETAC, București, 1982
[EC83] - ELES P., CIOCÂRLIE H. - "Considerații privind utilizarea limbajelor de nivel înalt pentru programarea sistemelor multiprocesor", The 5-th I.C.C.S.C.S., vol. II, București, 1983
[EC84a] - ELES P., CIOCÂRLIE H. - "EDISON, limbaj de programare concurentă pentru microsisteme", Sesiunea "Tehnic 2000", Timișoara, 19841
[EC84b] - ELES P., CIOCÂRLIE H. - "Conceptul de apel de procedură la distanță în limbajele de programare concurentă", CNETAC, București, 1984
[EC85a] - ELES P., CIOCÂRLIE H. - "Scope Analysis in a EDISON Compiler", Buletinul științific și tehnic al I.P. Timișoara, Tom 30 (44), 1985
[EC85b] - ELES P., CIOCÂRLIE H. - "EDISON D - limbaj de nivel înalt pentru programarea sistemelor distribuite", The 6-th Internat. Conf. on Control Syst. and Comp. Science, București, 1985
[EC85c] - ELES P., CIOCÂRLIE H. - "Aspects specifiques de la programation des microsistemes multiprocesseurs dans languages de haut niveau", Simp. "Microprocesoare, microcalculatoare și aplicații în economie", Timișoara, 1985
[EC86a] - ELES P., CIOCÂRLIE H. - "Interacțiunea proceselor prin APEL DE PROCEDURA LA DISTANTA SI RENDEZ VOUS în limbaje de programare de nivel înalt", Buletinul Simpozionului Național de Teoria sistemelor, Craiova, 1986
[EC86b] - ELES P., CIOCÂRLIE H. - "Aspecte specifice privind realizarea unui nucleu EDISON pentru un sistem multiprocesor", Buletinul St. și Tehn. al I.P. Timișoara, Tom 31, Fascicula 1-2, 1986
[EC87] -ELES P., CIOCÂRLIE H. - "Considerații privind implementarea unor variante ale conceptului de monitor", The 7-th I.C.C.S.C.S., București, 1987
[EC88a] - ELES P., CIOCÂRLIE H. - "Programarea concurentă cu monitoare. Implementare", Sesiunea "Tehnic 2000", Timișoara, 1988
[EC88b] - ELES P., CIOCÂRLIE H. - "Analiza de domeniu pentru un limbaj de programare structurat pe blocuri, cu module", Simp. nat. C.C.A.P., Timișoara, dec., 1988
[FG64] - FARBER D. J., GRISWOLD R. E., POLONSKY I. P. - "SNOBOL, a string manipulation language", Journal of ACM, 11/ian. 1964
[GC78] - GOOD D. I., COHEN R. M., ș.a. - "A report on the development of GYPSY", ICSCA-CMR13, The University of Texas at Austin, 1978
[GC79] - GOOD D. I., COHEN R. M., ș.a. - "Principles of proving concurrent programs in GYPSY", ICSCA-CMP15, The University of Texas at Austin, 1979
[GE82] - GEHRINGER E. F., ș.a. - "The Cm testbed", Computer, october 1982
[GE83] - GEHANI N. - "Ada-An advanced introduction", Prentice-Hall, Englewood

Cliffs, 1983
[GE85] - GEORGESCU I. - "Elemente de inteligență artificială", Editura Academiei. București. 1985
[GO74] - GOLDBERG R. P. - "Survey of virtual machine research", IEEE Computer. June. 1974
[GR83] - GOLDBERG A., ROBSON D. - "Smaltalk-80. The language and its implementation", Addison-Wesley. 1983
[GU77] - GUTTAG J. - "Abstract data types and the development of data structures". Comm. of ACM. V20. N6. 1977
[HA77] - HADDON B. K. - "Nested monitor calls", ACM SIG. V11, N4. 1977
[HI64] - HIGGINS P. J. - "Algebras with a scheme of operators", Math Nachr., No.27. 1964
[HI80] - HONEYWELL INC. - "Formal definition of the Ada programming language". Honeywell Inc., Minneapolis. MN. november 1980
[HL82] - HERLIHY M., LISKOV B. - "A value transmission method for abstract data types", ACM Trans. on Prog. Lang. and Syst., V4, N4. 1982
[HM84] - HULL E. M., Mc KEAG R. M. - "Communication sequential processes for centralized and distributed operating system design", ACM Trans. on Prog. Lang. and Syst., V6. N2. 1984
[HO72] - HOARE C. A. R. - Towards a theory of parallel programming. Operating systems tehniques", Academic press NY, 1972
[HO74] - HOARE C. A. R. - "Monitors: on operating system structuring concept". Comm. of ACM V17. N7. 1974
[HO78] - HOARE C. A. R. - "Communicating sequential processes", Comm. of ACM V21, N5, 1978
[HO80] - HOPPE J. - "A simple nucleus written in Modula-2. A case study", Softw. Pract. and Experience. V10. 1980
[HO83] - HOLT R. C. - "Concurrent Euclid. The Unix system and Tunis". Addison-Wesley. 1983
[HO84] - HOROWITZ E. - "Fundamentals of programming languages", Comp. Science Press. Rockville. 1984
[HO88] - HORN B. L. - "An introduction to object oriented programming, inheritance and method combination", CMU-CS-87-127. ian., 1988
[HR77] - HARTMANN A. C. - "A Concurrent Pascal compiler for minicomputers". Springer Verlag, 1977
[HW73] - HOARE C. A. R., WIRTH N. - "An axiomatic definition of the programming language Pascal", Acta Informatica. No. 2. 1973
[HW80] - HOLDEN J., WAND I. C. - "An assessment of Modula", Softw. Pract. and Experience. V10. 1980
[HW82] - HOLT R. C. - WORTMAN D. B. - "A model for implementing Euclid modules and prototypes", ACM Trans. on Prog. Lang. and Syst., V4, N4, 1982
[IC74] - IMPERIAL CHEMICAL INDUSTRIES LIMITED - "RTL/2 language specification", vesion 2, 1974
[IC79] - ICHBIAN J. D., ș.a. - "Rationale for design of the Ada programming language", ACM SIGPLAN Notice, V14, N6, 1979
[IN81a] - INTEL - "Introduction to the IAPX432 architecture", Intel Corporation, Santa Clara. 1981
[IN81b] - INGALLS D. H. H. - "Design principles behind smalltalk", Byte, V6, N8, 1981
[JI81a] - JAPAN INFORMATION PROCESSING DEVELOPMENT CENTER - "Preliminary report on study and research of fifith generation ccomputers", Tokyo, 1981
[Ji81b] - *** - "Processing of International Conference on fifth generation computer system", Japan information processing development center. Tokyo, 1981
[JM78] - JITARU M., MACARIE C., NICULESCU S. - "Indrumător de limbaje de programare", Editura Tehnică, București, 1978
[JO79] - JOSEPH M. - "Towards more general implementation languages for operating

systems", O. S. Theory and Practice, North Holland Publishing Co., 1979
[JO80] - JONES A. K. - "The Cm multiprocessor project: a research review", Comp. Science Depart., C-M. University, 1980
[JS80] - JONES A. K., SCHWARZ P. - "Experience using multiprocessor systems", ACM Computing Surveys, V12, N2, 1980
[JS81] - JULE K., SCHNEIDER H. - "Concurrent Pascal on the Intellec MDS", Euromicro. North Holland P-C, 1981
[JU84] - JURCA I. - "Sisteme de operare", Curs litografiat, I.P.T., 1984
[JW74] - JENSEN K., WIRTH N. - "Pascal-user manual and report", Springer Verlag, 1974
[KA81] - KACSUK P. - "A data driver emulator module based on a Concurrent Pascal oriented multiple microprocessor system", Euromicro, North Holland P-C, 1981
[KJ82] - KAUFMANN I., JURCA I., PETRIU D., CRETU V. - "Programarea in limbajul Ada", Editura Facla, Timişoara, 1982
[KN71] - KNUTH D. E. - "Semantics of context-free languages", Math. System Theory, No. 5, 1971
[KR82] - KRUIJER H. S. - "Processor management in a Concurrent Pascal Kernel", ACM SIG, V16, N2, 1982
[KS78] - KIEBURTZ R. B., SILBERSCHATZ A. - "Capability manegers", IEEE Trans. on Soft. Eng., V4, N6, 1978
[KS83] - KIEBURTZ R. B., SILBERSCHATZ A. - "Access right expressions", ACM Trans. on Prog. Lang. and Syst., V15, N1, 1983
[LA77] - LAMPSON B. W., ş.a. - "Report on the programming language Euclid", ACM Sigplan Notices, V12, N2, 1977
[LA81] - LAMPSON B. W., ş.a. - "Distributed systems-architecture and implementation", Springer, Verlag, 1981
[LD81] - LORIN H., DEITEL H. M. - "Operating systems", Addison-Wesley, 1981
[LE76] - LEWIS P. M., ş.a. - "Compiler design theory", Addison-Wesley, 1976
[LE77] - LEDGARD M. F. - "Production systems: a notation for defining syntax and translation of programming languages", IEEE Trans. on Soft. Eng., april, 1977
[LI77] - LISKOV B., ş.a. - "Abstraction mechanism in CLU", Comm of ACM, V20, N8, 1977
[LI78] - LIENHARD H. - "The real time programming language Portal. Introduction and survey", Sandis and Gyr Review, No 25, 1978
[LI79] - LISKOV B., ş.a. - "CLU reference manual", Labor. for Comp. Science, MIT, TR-225, Cambridge, 1979
[LM79] - LINGER R. C., MILLS H. D., WITT B. I. - "Structured programming: theory and practice", Addison-Wesley, 1979
[LP80] - LUCKHAM D. C., POLAK W. - "Ada exception handling: on axiomatic approach", ACM Trans. on Prog. Lang. and Syst., V2, N2, 1980
[LS74] - LEAVENWORTH B. M., SAMMET K. E. - "An overview of nonprocedural languages", SIGPLAN Notices, 9, nr. 4, 1974
[LS79] - LISKOV B., SNYDER A. - "Exception handling in CLU", IEEE Trans. on Soft. Eng., V5, N6, 1979
[LT77] - LISTER A. - "The problem of nested monitor calls", ACM SIG.,V11, N3, 1977
[MA76] - MARCOTTY M., ş.a. - "A sampler of formal definition", Computing Surveys, Vol. 8, No. 2, June 1976
[MA79] - "McGRAW J. R., ANDREWS G. R. - "Access Control in parallel programs", IEEE Trans. on Soft. Eng., V5, N1, 1979
[MA80] - MacLAREN L. - "Evolving toward Ada in real-time systems", ACM Symp. on Ada, 1980
[MA82] - MAYERS G. - "Advances in computer architecture", John Wiley and sons, NY, 1982
[MC72] - McCARTHY J. - "Formal semantic definition and the proof of compiler correctness", Courant Comp. Sc. Symp. 2, sept. 1970, Prentice-Hall, 1972
[ME81] - MacEVEN G. H., MARTIN P. - "Abstraction hierarchies in top-down design",

The Journal of Syst. and Softw., 2, 1981
[MK70] - Mc KEEMAN W. M. - "A compiler generator", Prentice-Hall, Englewood Cliffs, N.J., 1970
[ML80] - MacLENNAN B. J. - "Values and objects in programming languages", ACM Sigplan Notices, V15, N12, 1980
[MM86] - MUNTEANU E., MIHUT I., IVANOV A. - "Inițiere în limbajul Ada", Editura Tehnică, București, 1986
[MN81] - MOLLER-NIELSEN P., ș.a. - "A multi-processor for multi-program experience", Euromicro, North-Holland P-C, 1981
[MU85] - MURESAN I. - "Sisteme multiprocesor cu prelucrare în timp real", teză de doctorat, I.P.T., 1985
[MZ86] - McALLISTER D., ZABIH R. - "Boolean calasses", OOPSLA, october, 1986
[NA63] - NAUR P. - "Revised report on the Algorithmic language ALGOL 60", Comm. of ACM, V6, N1, 1963
[NG82] - NICULESCU S., GOLESTEANU D. P. - "Macroprocesoare și limbaje extensibile", Ediura Științifică și Enciclopedică, București, 1982
[NI75] - NICHOLLS J. E. - "The structure and design of programming languages", Addison-Wesley , 1975
[NI85] - NICOLAU E. - "Ingineria cunoașterii", Editura Albatros, București, 1985
[NS79] - NICULESCU S. - "Fortran. Inițiere în programarea structurată", Editura Tehnică, București, 1979
[PA72a] - PARNAS D. L. - "On the criteria to be used in decompasing systems into modules", Comm. of ACM, V15, N12, 1972
[PA72b] - PARNAS D. L. - "A tehnique for software modul specification with exemples", Comm. of ACM, V15, N5,1972
[PA82] - PAUNESCU F. - "Analiza și concepția sistemelor de operare", Editura Științifică și Enciclopedică, București, 1982
[PE87] - PETRESCU M. ș.a. - "Optimal querry processing in relational DBMS", The 4th ICCSCS, vol. III, Bucharest, 1987
[PS75] - PARNAS D.L.,SIEWIOREK D.P. - "Use of the concept of transparency in the design of hierarchically structured systems", Comm of ACM, V18, N7, 1975
[PS85] - PETERSON J.L., SILBERSCHATZ A. - "Operating system concepts", Addison-Wesley, 1985
[RA84] - RADOACA V., ș.a. - "Implementarea limbajului Edison pe minicalculatoarele si micrositemele românești", Lucrările C.N.E.T.A.C., I.P.București, 1984
[RB81] - ROBERTS E.S., ș.a. - "Task management in Ada a critical evaluation for real-time multiprocessors", Softw. Pract. and Exper., V11, 1981
[RE82] - RENTSCH T. - "Object oriented programming", ACM Sigplan Notices, V17, N9, 1982
[RG81] - ROBSON D., GOLDBERG A. - "The Smalltalk-80 system", Byte, V6, N8, 1981
[RO81] - ROBSON D. - "Object oriented software systems", Byte, V6, N8, 1981
[RU83] - RUS T. - "Mecanisme formale pentru specificarea limbajelor", Editura Academiei, 1983
[SB78] - SCHNEIDER F.B., BERNSTEIN A.J. - "Scheduling in Concurrent Pascal", ACM SIGPLAN, V12, N2, 1978
[SB79] - SCHNEIDER F.B., BERNSTEIN A.J. - "Mechanism for specifying ccheduling policies", Cornell Univ., New York, 1989
[SC72] - SCHWARTZ J.T. - "Semantic definition methods and the evolution of the programming languages", Courant Comp. Sc., Symp. 2, sept. 1970, Prentice-Hall Inc., 1972
[SC78] - SCHILD R., - "Parallel processes in Portal exemplifical in a group project", Landis and Gyr Review, No. 5.1978
[SC84] - SCHWARTZ P.M. - "Transactions of typed objects", Cornegie-Mellow Univ., Technical Raport, CMU-CS-84-166, Pittsburgh, 1984
[SC86] - SCHAFFERT C., ș.a. - "An introduction to Trellis/Owl", OOPSLA, 1986
[SE78] - SEIDEL H.A. - "A Kernel for a Concurrent Pascal operating system", IRIA

and CMU Symp. of Operating syst. Theory and Pract., Rocqencourt, 1978
[SH74] - SHAW A.C. -"The logical design of operating systems", Prentice-Hall, Englewood Cliffs, 1974
[SH79] - SHOCH J.F. - "An overview of the programming language Smalltalk-72", ACM Sigplan Notices, V14, N9, 1979
[SI77] - SILBERSCHATZ A., ş.a. - "Extending Concurrent Pascal to allow dynamic resource management", IEEE Trans. on Soft. Eng., V3, N3, 1977
[SL80] - SCHILD R., LIENHARD H. - "Real-time programming in Portal", ACM SIGPLAN NoticesV15, N4, 1980
[SN85] - SNYDER A. - "Object oriented programming in Common Lisp", HP-Company, ATC-85-1, february, 1985
[SS83] - SCHWARZ P.M., SPECTOR A.Z. - "Synchronising chared abstract types", Carnegie -Mellow Univ., Technical Raport, CMU-CS-84.166, Pittsburgh, 1984
[ST84] - STEELE G.L. - "Common LISP, the language", Digital Press, Maynard, Massachussetts, 1984
[ST86a] - STREIANU I. - "LISP, limbajul de programare al inteligenței artificiale", Editura Științifică și Enciclopedică, București, 1986
[ST86b] - STROUSTRUP B. - "The C++ programming language", Addison Wesley, 1986
[ST88] - SANNELLA D., TARLECKI A. - "Toward formal development of programs algebric specifications:implementations revisited", Acta Informatica, 25, 1988
[ŞA89] - ŞAFAREVICI I.R. - "Noțiuni fundamentale ale algebrei", Editura Academiei RSR, 1989
[ŞC84] - ŞERBANAȚI L.D., CRISTEA V., MOLDOVEANU F., IORGA V. - "Programarea sistematică în limbajele Pascal și Fortran", Editura Tehnică, București, 1984
[ŞE80] - ŞERBANAȚI L.D. - "Implementarea unui limbaj pentru un sistem interactiv de prelucrare a datelor", -Teza de doctorat, I.P.B., 1980
[ŞE87] - ŞERBANAȚI L.D. - "Limbaje de programare și compilatoare", Editura Academiei , București, 1987
[TA79] - TAPOLA P.K.J. - "Pascal compiler for 8-bit microprocessor", Helsinki Univ. of Tecnol., Computer Center, Technical Raport, No 7., 1979 [T85] - TAYLOR R.N. - "Steps to an advanced Ada programming environment", IEEE Trans on Softw. Eng., V11, N3, 1985
[TA85] -TAYLOR R.N. - "Steps to an advanced Ada programming environment", IEEE Trans. on Soft. Eng., V11, N3, 1985
[TE81] - TESLER L. - "The Smalltalk environment", Byte, V6, N8, 1981
[TH86] - THOMSEN K. - "Multiple inheritance a structuring mechanism for data, processes and procedures", Departament of C.S., Arhus University, april, 1989
[TS85] - TREMBLAY J.P., SORENSON P.G. - "The theory and practice of computer writing", McGraw Hill, New York, 1985
[VA77] - VADUVA I. - "Modele de simulare cu calculatorul", Editura Tehnică, București, 1977
[VB78] - VADUVA I., BALTAC V., FLORICICA I. - "Programarea structurată", Editura Tehnica, București, 1978
[VB86] - VADUVA I., BALTAC V., FLORICICA I., JITARU M. - "Ingineria programării", Editura Academiei, București, 1986
[WB79] - WELSH J., BUSTARD D. W. - "Pascal-Plus, another language for modular multiprogramming", Softw. Pract. and Experience, V9, 1979
[WE72] - WEGNER P. - "The Vienna definition language", Computing Surveys, Vol. 4, No. 1, March 1972
[WE80] - WEGNER P. - "Programming with Ada. An introduction by means of graduated examples", Prentice Hall, Englewood-Cliffs, 1980
[WH84] - WICHMANN B. A. - "Is Ada too big ? A designer answers to critics", Comm. of ACM, V27, N2, 1984
[WI71] - WIRTH N. - "The programming language Pascal", Acta Informatica, No. 1, 1971

[WI73] - WIRTH N. - "Systematic programming. An introduction", Prentice Hall, Englewood Cliffs, New Jersey, 1973
[WI75] - WIRTH N. - "Pascal-S: a subset and its implementation", Berichte das Instituts für Informatik ETH Zürich, 1975
[WI76] - WIRTH N. - "Algorithms + data structures = programs", Prentice Hall, Englewood Cliffs, 1976
[WI77a] - WIRTH N. - "Modula: a language for modular multiprogramming", Softw. Pract. Exper. V7, 1977
[WI77b] - WIRTH N. - "The use Modula", Softw. Pract. and Exper., V7, 1977
[WI77c] - WIRTH N. - "Design and implementation of Modula", Softw. Pract. and Exper., V7, 1977
[WI77d] - WIRTH N. - "Toward a discipline of real-time prtogramming", Comm. of ACM, V20, N8, 1977
[WI82] - WIRTH N. - "Programming in Modula-2", Springer Verlag, 1982
[WJ69] - WIJNGAARDEN van A. ş.a. - "Report on the algorithmic language Algol 68", MR 101, Math Centrum, Amsterdam, 1969
[WJ65] - WIJNGAARDEN van A. ş.a. - "Revised report on the algorithmic language Algol 68", Acta Informatica, V5, N 1-3, 1975
[WL81] - WELSH J., LISTER A. - "A comparative study of task communication in Ada", Softw. Pract. and Exper. V11, 1981
[WN79] - WINSTON P. - "LISP" - Addison-Wesley, 1979
[WO85] - WOLF A. L. - "Language and tool support for precise interface controll", Ph. D. disertation, tehn. rep. 85-23, Univ. of Massachussetts, september 1985
[WO87] - WOLF A. L., ş.a. - "A model of visibility control", COINS Technical Report, 87-12, February, 1987
[WS84] - WEINER R., SINCOVEC R. - "Software engineering with Modula-2 and Ada", John Wiley and Sons, New York, 1984
[WU81] - WULF W. A. - "Fundamental structures of computer science", Addison-Wesley, 1981
[YC79] - YORDON E., CONSTANTINE L. - "Structured design. Fundamentals of a discipline of computer program and system design", Prentice-Hall, Englewood Cliffs, 1979
[YO81] - YOUNG S. - "Real time languages: design and development", John Wiley and Sons, New York, 1981
[ZE81] - ZEIGLER S. - "Ada for the Intel 432 microcomputer", IEEE Computer, June 1981