CLUJ-NAPOCA
2006

SIMONA MOTOGNA

METODE DE PROIECTARE A COMPILATOARELOR

Descrierea CIP a Bibliotecii Naționale a României

MOTOGNA, S.
Metode de proiectare a compilatoarelor / Simona Motogna. - Cluj-Napoca : Editura Albastră, 2006
Bibliogr.
ISBN 973-650-172-8

62.00163:004.4'422(075.8)

Referent:

Prof. univ. Dr. Florian Mircea Boian
Facultatea de Matematică și Informatică
Universitatea "Babeș-Bolyai" Cluj-Napoca

Editura Albastră

Director editură
Smaranda Derveșteanu

Tehnoredactare computerizată
SD Special Design

Coperta
SD Special Design

Tipărit
EDITURA ALBASTRĂ

Editura este acreditată de CNCSIS (Consiliul Național al Cercetării Științifice din Învățământul Superior) cu numărul 1142 din 30 iunie 2003 și este recomandată Consiliului Național de Atestare a Titlurilor și Diplomelor Universitare.

Copyright © 2006

Toate drepturile asupra acestei lucrări aparțin s.c. Casa de Editură Albastră s.r.l. Reproducerea integrală sau parțială a textului sau a ilustrațiilor din această carte este posibilă numai cu acordul prealabil în scris al Casei de Editură Albastră.

CUPRINS

PREFAȚĂ 7

1. INTRODUCERE 11
   1.1. Ce este un compilator? 11
   1.2. Istoric – limbaje de programare și compilatoare 16
   1.3. Specificarea unui limbaj de programare 18

2. ANALIZA LEXICALĂ 22
   2.1. Detectare și clasificare 22
   2.2. Codificare 24
   2.3. Modelul analizorului lexical 27
      2.3.1. Expresii regulare 28
      2.3.2. Automate finite 29
   2.4. Tabela de simboluri 30
      2.4.1. Tabele de dispersie 32
   2.5. Generator de analiză lexicală – lex 37
      2.5.1. Formatul fișierului de specificație 38
      2.5.2. Mod de funcționare 40
   2.6. Relația dintre analizor lexical și analizor sintactic 42

3. ANALIZA SINTACTICĂ 44
   3.1. Analizor sintactic descendent cu reveniri 46
      3.1.1. Modelul analizorului sintactic descendent cu reve-niri 49
   3.2. Analizor sintactic LL(k) 53
      3.2.1. Gramatici de tip LL(k) 54
      3.2.2. Analizorul sintactic LL(1) 55
      3.2.3. Construirea tabelului de analiză LL(1) 55
      3.2.4. Funcționarea analizorului sintactic LL(1) 58
      3.2.5. Calculul funcțiilor FIRST și FOLLOW prin matrice booleene 63
   3.3. Analizoare sintactice LR(k) 65
      3.3.1. Gramatici de tip LR(k) 66
      3.3.2. Analizor sintactic LR(0) 68
      3.3.3. Analizor sintactic SLR 76
      3.3.4. Analizor sintactic LR(1) 79
      3.3.5. Analizor sintactic LALR 86

3.4. Studiul comparativ al analizoarelor sintactice 90
3.5. Generatoare de analiză sintactică 92
3.5.1. Generator de analiză sintactică descendentă - *LLgen* 92
3.5.2. Generator de analiză sintactică ascendentă -*yacc* . . 95

4. ANALIZA SEMANTICĂ 98
4.1. Scopul şi instrumentele analizei semantice 98
4.2. Arborele de sintaxă abstractă 98
4.3. Identificarea 99
4.4. Verificarea tipurilor 102
4.5. Gramatici de atribute 105
4.5.1. Evaluator de atribute 106
4.5.2. Clase speciale de gramatici de atribute 109
4.6. Metode manuale 114
4.6.1. Graful fluxului de control 114
4.6.2. Interpretare simbolică 116
4.6.3. Ecuaţii de flux de date 118

5. GENERAREA CODULUI INTERMEDIAR 121
5.1. Cod intermediar 121
5.2. Cod cu trei adrese 123
5.2.1. Reprezentarea codului cu trei adrese 126
5.2.2. Trecerea la cod cu trei adrese a arborelui sintactic . 129

6. OPTIMIZAREA CODULUI INTERMEDIAR 134
6.1. Graful de flux al unui program 134
6.2. Optimizări locale 137
6.3. Optimizarea ciclurilor 139

7. GENERAREA CODULUI OBIECT 141
7.1. Generare codului pentru calculatoare cu acumulator 142
7.2. Generare codului pentru calculatoare cu regiştri 143
7.3. Alocarea regiştrilor 144
7.3.1. Alocarea regiştrilor bazată pe colorarea grafurilor . . 146
7.4. Selectarea instrucţiunilor 150
7.4.1. Algoritmul lui Sethi – Ullman 152
7.5. Forma codului obiect 155

8. TRATAREA ERORILOR 157
8.1. Tratarea erorilor în analiza lexicală 158
8.2. Tratarea erorilor în analiza sintactică 159
8.2.1. Tratarea erorilor în analiza sintactică LL(1) 160
8.2.2. Tratarea erorilor în analiza sintactică LR(k) 160

BIBLIOGRAFIE 164

PREFAŢĂ

Unul dintre cele mai dinamice domenii, informatica, a ajuns să fie considerată astăzi o ştiinţă în sine, cu subdomenii variate, atât teoretice, cât şi practice. Timpul a demonstrat că în momentul în care putem elabora un model matematic sau algoritmic pentru o anumită problemă, şansa de a o rezolva, şi în plus de a găsi o soluţie eficientă, creşte considerabil.

Metodele de proiectare şi algoritmii din construcţia compilatoarelor oglindesc fidel evoluţia informaticii. Este de remarcat faptul că unele idei au reprezentat o asemenea inovaţie la timpul lor încât au rămas standarde timp îndelungat (douăzeci, uneori chiar treizeci de ani). Pe de altă parte, apariţia de noi limbaje de programare, noi arhitecturi de calculatoare, noi sisteme de operare au avut fiecare în parte efecte asupra compilatoarelor.

Cartea de faţă se doreşte a fi un ghid în studierea acestui domeniu, fiind utilă celor care doresc să înţeleagă proiectarea şi funcţionarea unui compilator, considerate cunoştiinţe de bază necesare oricărui bun programator. În plus, ea poate fi utilă studenţilor de la secţiile de Informatică, Matematică – Informatică şi Calculatoare, care urmează un curs pe această temă.

De ce?

Există suficiente motive pentru a ne pune această întrebare. Sper ca cititorul să-şi regăsească scopurile în cel puţin unul dintre aceste argumente.

1) Construcţia compilatoarelor reprezintă o ramură fundamentală a informaticii (şi în plus, unul dintre primele domenii care au primit acest titlu), datorită unei structurări judicioase a problemei, descrierii unui mecanism solid pentru algoritmi şi folosirii de utilitare (instrumente, generatoare) ori de câte ori e posibil.

Structurarea se referă, în primul rând, la separarea dintre analiză şi sinteză: compilatorul analizează datele de intrare, construieşte o reprezentare semantică a lor şi sintetizează apoi datele de ieşire.

Structurarea poate fi considerată şi pe un nivel următor, analiza şi sinteza fiind divizate în fazele compilării. Astfel, un analizor lexical sau sintactic pot fi folosite cu succes pentru mai multe limbaje sau maşini, modificând doar datele de intrare.

Algoritmii cei mai performanţi pentru fazele de analiză au fost construiţi

pe baza rezultatelor din teoria limbajelor formale, asigurându-se mecanismul formal. Folosirea expresiilor regulare în analiza lexicală, respectiv a gramaticilor independente de context în analiza sintactică sunt cele mai ilustrative exemple.

2) Construcţia compilatoarelor are o aplicabilitate foarte largă: Reacţia normală la prima vedere a acestei tematici ar fi: "Eu nu voi scrie niciodată un compilator. De ce să mă intereseze acest domeniu?". Tehnicile şi algoritmii proprii construcţiei compilatoarelor sunt folosite cu succes în alte domenii: prelucrarea fişierelor HTML, XML, conversia fişierelor, verificarea corectitudinii expresiilor aritmetice, interpretarea fişierelor de comenzi şi multe altele. Orice informaţie care are o structură ce poate fi descrisă printr-o gramatică poate profita de algoritmii din construcţia compilatoarelor.

3) Construcţia compilatoarelor îmbunătăţeşte experienţa de programare: implementarea diferiţilor algoritmi din fazele compilării nu este chiar o problemă trivială: alegerea structurilor de date potrivite şi implementarea eficientă a algoritmilor ajută la acumularea unei experienţe însemnate de programare.

În plus, un programator este, de fapt, doar un utilizator al unui compilator. Studiul acestui domeniu, înţelegerea mecanismelor "ascunse" de obicei de apăsarea unei combinaţii de taste sau a mouse-ului va permite unui bun programator să evite unele situaţii, să găsească uşor rezolvarea unei erori de compilare sau chiar de execuţie şi să folosească la maximum posibilităţile limbajului.

Structura cărţii cuprinde opt capitole, care urmează structurarea pe faze a unui compilator, motiv pentru care această ordine ar trebui urmată în parcurgerea cărţii. Fiecare capitol este astfel structurat încât porneşte de la noţiuni de bază şi algoritmi fundamentali corespunzători fazei respective de compilare şi continuă cu exemple şi studii de caz, pentru o mai bună înţelegere.

Capitolul 1 prezintă structura generală a unui compilator, împărţirea pe etape a operaţiilor executate şi o scurtă prezentare a relaţiei dintre compilatoare şi limbaje de programare, atât din punct de vedere cronologic cât şi al modului în care un limbaj trebuie descris pentru a putea implementa un compilator destinat lui.

Capitolul 2 descrie prima fază a procesului de compilare şi anume, analiza lexicală. Sunt prezentate acţiunile urmărite de analiza lexicală, modelele matematice ce se pot folosi şi, în încheiere, generatorul de analiză lexicală *lex*.

În capitolul 3, analiza sintactică, sunt prezentate câteva metode de analiză sintactică, începând cu cea mai simplă şi continuând cu cele mai

eficiente. Fiecare metodă este descrisă prin algoritmi care pot fi relativ uşor implementaţi şi prin exemple. În încheiere sunt descrise două generatoare de analiză sintactică.

Capitolul 4 este dedicat analizei semantice, specificându-se acţiunile urmărite şi prezentându-se principalele modalităţi de efectuare a acestei faze: gramatici de atribute, interpretare simbolică, respectiv ecuaţii de flux de date.

Generarea codului intermediar face obiectul capitolului 5. Sunt descrise formele codului intermediar, punându-se accent pe codul cu trei adrese şi a reprezentărilor acestuia.

Acest capitol este urmat de capitolul 6 destinat operaţiilor de optimizare a codului intermediar, în care posibilităţile de optimizare sunt studiate din punct de vedere al costului lor.

Capitolul 7 prezintă ultima etapa din procesul de compilare, generarea codului obiect, discutându-se despre cerinţele de care depinde forma programului obiect, modul în care acesta este generat în funcţie de tipul calculatorului şi despre optimizările ce se pot aplica pentru o compilare mai eficientă.

Capitolul 8 descrie erorile care pot apărea în timpul compilării, acestea fiind clasificate în funcţie de faza în care apar, putându-se astfel alege diferite strategii specifice pentru tratarea erorilor.

Experienţa acumulată în predarea cursurilor de limbaje formale şi compilatoare, la secţia de Informatică a constituit un punct de plecare solid în elaborarea acestei cărţi. Generaţiile de studenţi cu care am lucrat m-au ajutat să structurez şi să corectez materialele.

Doresc să mulţumesc în primul rând domnului Cornel Tarţia, de la care am învăţat noţiunile de teoria compilării şi care de-a lungul timpului m-a sprijinit şi îndrumat consecvent. Mulţumirile mele se îndreaptă spre domnii prof.dr. Florian Boian şi Militon Frenţiu, pentru sprijinul, încurajările şi aprecierile relative la acest manuscris. Adresez recunoştiinţa mea colegilor prof.dr. Horia Pop şi asist. drd. Adriana Tarţa pentru ajutorul dat în redactarea şi corectarea manuscrisului.

Simona Motogna

1. INTRODUCERE

1.1. Ce este un compilator?

Limbajele de programare sunt de o mare diversitate şi total diferite de limbajului calculatorului. Şi totuşi un program scris într-un limbaj oarecare de programare poate fi executat pe un calculator. Acest lucru se explică prin faptul că programul este ***tradus*** în limbaj maşină, care poate fi executat. Translatorul este un program care primind ca intrare un program scris într-un anumit limbaj, numit ***program sursă***, produce ca rezultat un program într-un alt limbaj, numit ***program obiect*** sau rezultat, conservând seminificaţia programului sursă. Dacă programul obiect este în cod maşină sau alt limbaj apropiat de limbajul maşină, atunci translatorul se numeşte ***compilator***. În această fază, un compilator este ca o "cutie neagră", deoarece nu avem detalii despre traducerea efectivă a programului sursă în program obiect (figura 1.1).

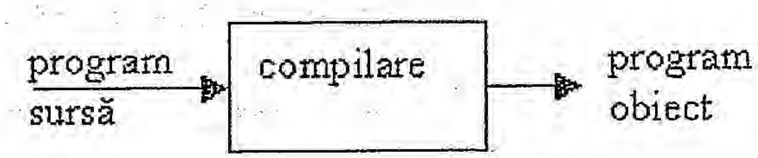

Figura 1.1: Procesul de compilare privit ca o "cutie neagră"

Procesul compilării este complex, astfel încât nu e normal atât din punct de vedere logic cât şi din punct de vedere al implementării să fie considerat un proces într-un singur pas. Programul sursă necesită o secvenţă de transformări în forme intermediare până se ajunge la forma codului maşină sau a unui alt limbaj. Aceste transformări pot fi clasificate în două mari categorii, ca în figura 1.2:

- *partea de analiză* constă în procesarea programului sursă în vederea determinării structurilor lexicale şi sintactice din program, precum

şi a semnificaţiei acestora, obţinându-se o reprezentare abstractă, independentă de limbaj, a tuturor acestor informaţii;

- această reprezentare abstractă este apoi preluată de *partea de sinteză* care are ca scop generarea programului obiect, în funcţie de caracteristicile acestuia şi ale calculatorului pe care este destinat să ruleze codul asociat.

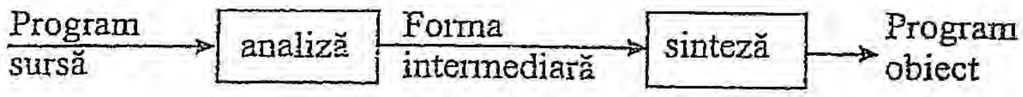

Figura 1.2: Părţile de analiză şi sinteză ale unui compilator

Cele două părţi reprezintă, la rândul lor, procese complexe care se desfăşoară în mai mulţi paşi, cu parcurgeri repetate a reprezentării programului sursă. Paşii fundamentali, numiţi *fazele compilării*, corespunzători procesului de compilare sunt descrişi în figura 1.3 şi au următoarea semnificaţie:

1. **Analiza lexicală** separă caracterele din programul sursă în subşiruri grupate logic. Programul sursă este privit ca un şir de caractere în care se delimitează subşiruri numite atomi lexicali, reprezentând: cuvinte cheie, operatori, constante, identificatori şi separatori.

2. Şirul de atomi lexicali este apoi preluat de **analiza sintactică**, al cărui rol este de a determina entităţile sintactice (declaraţii, expresii, instrucţiuni), de a verifica corectitudinea sintactică a programului şi, în caz afirmativ, de a construi arborele de analiză sintactică, ce constituie o reprezentare echivalentă a programului sursă, care specifică entităţile sintactice şi relaţiile dintre ele.

3. **Analiza semantică** porneşte de la arborele de analiză sintactică pentru a extrage informaţii relative la variabile, tipuri, expresii şi pentru a verifica consistenţa folosirii lor. Rezultatul acestei faze este arborele atribuit de sintaxă abstractă, de fapt, arborele de analiză îmbogăţit cu anumite atribute.

4. **Generarea codului intermediar** va prelucra acest arbore în scopul obţinerii unei forme simple de reprezentare a programului, care conţine doar instrucţiuni cu format stabilit, care lucrează cu operanzi şi variabile din program. Criteriul principal pentru alegerea codului

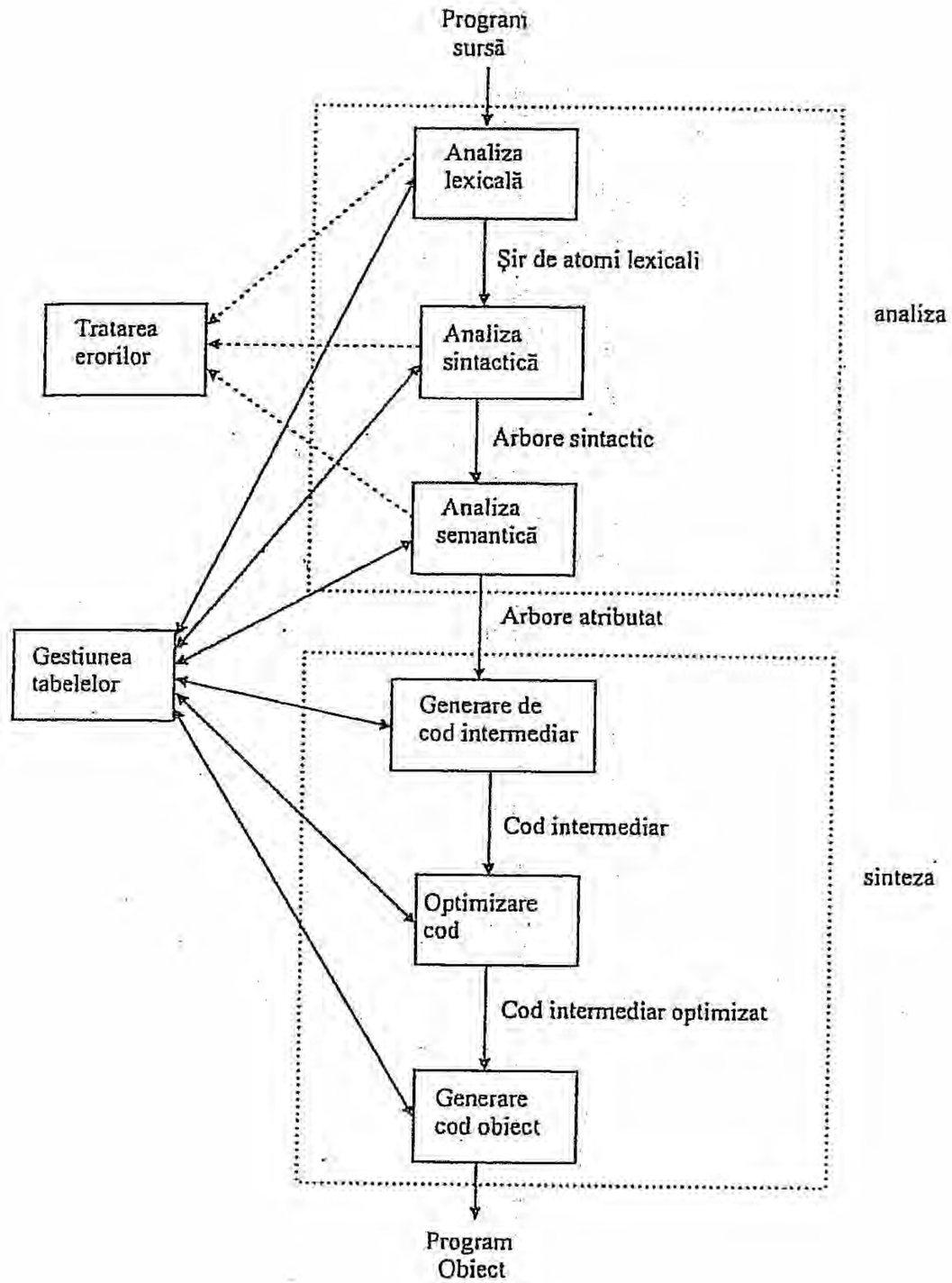

Figura 1.3: Părţile componente ale unui compilator

intermediar este de a fi cât mai uşor de generat cod maşină pentru calculatoare cât mai diverse şi constă aproape exclusiv din expresii şi instrucţiuni de control. De exemplu, o instrucţiune de ciclare este înlocuită cu teste, etichete şi salturi.

5. **Optimizarea** în sine nu este o fază obligatorie în procesul de compilare, ea fiind argumentată de eficienţă: programul obiect va fi mai mic şi execuţia lui se va face într-un timp mai scurt.

   Operaţiile de optimizare se pot aplica atât asupra codului intermediar cât şi asupra codului obiect, deci această fază se poate efectua în două locuri în procesul de compilare. Se poate amâna optimizarea pentru codul obiect, inserând în acea fază şi operaţii de optimizare specifice (alocarea regiştrilor, încărcarea şi salvarea informaţiei din regiştri). Considerentul avut în vedere la abordarea aleasă a fost de a aplica optimizarea cât mai repede cu putinţă. Ideal ar fi să existe două etape de optimizare: a codului intermediar şi a codului obiect, dar în acest caz procesul de compilare va dura mai mult. Optimizarea este, în esenţă, un compromis între eficienţa programului obiect şi durata compilării.

6. Ultima fază a compilării, **generarea de cod obiect** constă în alocarea şi gestiunea regiştrilor, alocarea variabilelor în regiştri şi translatarea instrucţiunilor din codul intermediar în cod obiect.

7. Datorită complexităţii entităţilor cu care lucrează un compilator, există două componente cu rol important, dar care nu intervin în fluxul de date transmis pe parcursul fazelor compilării, şi anume:

   - **Gestiunea tabelei de simboluri** - care conţine mai mult rutine pentru crearea, accesarea şi actualizarea informaţiilor legate de numele simbolice din program. Tabla de simboluri este creată în faza de analiză lexicală şi folosită în toate celelalte faze ale compilării: fie pentru aflarea unor informaţii, fie unele câmpuri sunt completate doar în faze mai avansate ale procesului de compilare.
   - **Tratarea erorilor** - constă într-o bibliotecă de rutine care permite tratarea specifică a erorilor de compilare corespunzător fazei în care apar.

Aceasta este o prezentare generică a compilatoarelor, însă există şi unele cazuri speciale care se abat de la această împărţire pe faze sau particularizează limbajul obiect. Multe programe sunt încadrate în familia compilatoarelor datorită funcţionalităţii lor.

Interpretorul este un program care execută acţiunile specificate de reprezentarea semantică în loc să le re-exprime într-o formă intermediară. De aceea, se spune că un interpretor nu creează cod obiect, ci "simulează" execuţia programului instrucţiune cu instrucţiune. Există unele avantaje ale folosirii unui interpretor, ca: portabilitate mărită, o mai eficientă modalitate de tratare a erorilor şi securitate mai bună (unul din principalele motive ale succesului limbajului Java). Principalul dezavantaj îl reprezintă timpul mare de execuţie.

Asamblorul este un compilator pentru limbaj de asamblare. **Preprocesoarele** sunt, de fapt, translatoare care traduc programe dintr-un anumit limbaj în alt limbaj, de obicei, limbajul rezultat având un compilator eficient. Multe limbaje "experimentale" foloseau preprocesoare care transformau programul sursă în program C. Un "**cross-compilator**"[1] este un compilator rezident pe un calculator "gazdă" scris pentru a genera cod obiect destinat unui alt calculator.

Tehnica de a crea compilatoare prin instrumente de generare a programelor a fost iniţiată de Brooker ş.a. [BMMR63] şi a evoluat consistent de-a lungul timpului, alcătuind o clasă aparte numită **generatoare de compilatoare**. Cele mai semnificative exemple sunt: generarea unui analizor lexical din specificaţii bazate pe expresii regulare, generarea unui analizor sintactic pe baza unei gramatici independente de context care să descrie sintaxa şi generatoare de cod din descrieri de atribute.

Avantajele folosirii acestor instrumente pot fi rezumate în următoarele caracteristici:

1. Intrarea pentru un generator de programe se face la un nivel mai înalt de abstractizare. Programatorul va specifica mai puţin, responsabilităţile fiind preluate de instrument (un beneficiu important pentru unele aspecte ale compilării, cum ar fi tratarea erorilor).

2. Posedă o flexibilitate şi adaptabilitate mărită. Modificările de limbaj necesită schimbări minimale.

3. Programul generat poate fi adăugit cu cod propriu sau inclus (din biblioteci), oferind posibilitatea de a construi un analizor specializat.

4. Descrierea formală permite generarea mai multor tipuri de programe. De exemplu, o gramatică asociată unui limbaj în scopul de a genera un analizor sintactic, poate fi folosită şi pentru a genera un editor dependent de sintaxă.

[1] L.D.Şerbănaţi [Şer87] introduce această denumire, o traducere aproximativă putând fi compilator de trecere

1.2. Istoric – limbaje de programare şi compilatoare

Evoluţia compilatoarelor este strâns legată de limbajele de programare şi de calculatoare. De aceea, nu putem trata compilatoarele în afara acestui cadru. Pe măsură ce limbajele de programare au devenit mai puternice, mai sofisticate, puterea lor de calcul s-a mărit, a apărut imperios necesitatea "traducerii" acestora într-un limbaj apropiat de limbajul maşină sau într-un alt limbaj de programare cunoscut. Pe de altă parte, puterea de calcul a calculatorului a permis dezvoltarea unor limbaje şi compilatoare mai eficiente. Vom încerca să amintim momentele importante ale acestui domeniu, fără a avea pretenţia că am inclus toate rezultatele importante.

Primul compilator apărut este cel al limbajului *Fortran*. Primul compilator Fortran a reprezentat un punct de referinţă în istoria compilatoarelor, deoarece la acea dată calculatoarele aveau memorie foarte mică, erau încete, iar sistemele de operare erau foarte primitive, condiţii în care programarea în limbaje de asamblare părea singura opţiune reală. Fortran a fost considerat primul limbaj de nivel înalt, iar compilatorul producea cod eficient (metoda de optimizare a codului propusă de acest compilator a rezistat timp de 20 ani). Compilatorul a fost proiectat şi implementat între 1954 şi 1957 de o echipă de la IBM, condusă de John W. Backus [Bac78].

Primul interpretor este creditat limbajului *Lisp*, în 1962 [MAEHL66].

Dacă adăugăm la aceste două realizări importante apariţia limbajului Algol, în 1960 [JN61], considerat primul limbaj de nivel înalt cu o sintaxă structurată şi sistematic definită, putem spune că această perioadă de început a compilatoarelor a reuşit să impună nişte modele şi principii fundamentale.

Ideea de a construi un compilator pentru un limbaj de programare a fost adoptată cu mult succes şi majoritatea principiilor de proiectare a compilatoarelor au fost dezvoltate în perioada următoare, 1960 - 1975. Acest progres a influenţat şi dezvoltarea limbajelor de programare, permiţând definirea unor construcţii complexe şi extinderi ale diferitelor limbaje.

Folosirea limbajelor de nivel înalt în scrierea compilatoarelor a fost impulsionată de apariţia limbajelor Pascal şi C, a căror compilatoare au fost implementate în aceste limbaje. Tot în această perioadă a scăzut importanţa sintezei codului în favoarea analizei codului. Rezultatele obţinute în teoria limbajelor formale au dus spre elaborarea unor tehnici performante care puteau fi aplicate cu succes în partea de analiză a codului, în special la analiza sintactică.

Limbajul *Pascal*, definit în 1970 de prof. Nicholaus Wirth (ETH Zürich)

[Wir71a, Wir71b], a apărut ca cel mai de succes succesor al limbajului Algol. Popularitatea şi eficienţa limbajului se datorează următoarelor trei performanţe:

- compilatorul care produce un cod intermediar simplu pentru o maşină virtuală (numit P-cod), în loc să genereze cod nativ pentru o maşină particulară şi un sistem de operare particular, asigurându-se astfel portabilitatea limbajului;
- interpretorul pentru P-cod nu necesita un calculator mainframe mare şi costisitor, putând rula şi pe un calculator personal (UCSD Pascal - University of California, San Diego) [UCS];
- lansarea mediului Turbo Pascal de către Borland, care reprezenta un compilator rapid şi ieftin şi un mediu integrat de dezvoltare pentru calculatoare personale [Bor].

Limbajul Pascal a influenţat proiectarea şi evoluţia multor alte limbaje de programare: N. Wirth a fost implicat într-o serie de proiecte ulterioare care aveau la bază limbajul Pascal (Modula-2, Simula, Smalltalk, Oberon, Component Pascal), dar şi o serie de alte limbaje de la Ada la Visual Basic.

Limbajul *C* a fost elaborat între 1969 şi 1973, în paralel cu dezvoltarea sistemului de operare Unix, când s-au pus bazele limbajului. Schimbări majore ale limbajului s-au produs însă abia între 1977 şi 1979, când a apărut prima descriere detaliată a limbajului în "*The C Programming Language*" [KR88]. În sfârşit, la mijlocul deceniului 80, limbajul a fost oficial standardizat ANSI. Până la începutul anilor '80, deşi existau diverse compilatoare pentru o diversitate de platforme şi sisteme de operare, limbajul era exclusiv asociat cu Unix. Mi-am permis să închei această scurtă prezentare a limbajului C, cu un citat aparţinând lui Dennis. M. Ritchie [KR88], unul din oamenii implicaţi în dezvoltarea limbajului încă de la început (1969), alături de Kerrighan:

> *Între 1973 -1980... scrierea nucleului (kernel) Unix în C ne-a dat suficientă încredere în utilitatea şi eficienţa limbajului pentru a recoda utilitarele şi instrumentele sistemului şi pentru a le transpune, apoi, pe cele mai interesante pe alte platforme...C a devenit un succes la un nivel care a întrecut orice aşteptare ...*

O a treia perioadă, din 1975 până în prezent, se poate caracteriza prin faptul că cerinţele programatorilor se îndreptau spre compilatoare profesionale: eficiente (în generare de cod şi folosire), sigure şi, de preferat, cu interfeţe grafice plăcute, punându-se accent pe calitatea codului generat.

În anii '90 au apărut un număr mare de compilatoare şi instrumente de dezvoltare de compilatoare (generatoare de analiză lexicală şi sintactică) ca parte a proiectului GNU sau iniţiative surse deschise (open source), care permiteau citirea efectivă a codului unui compilator.

Trebuie menţionate aici şi direcţiile de dezvoltare a limbajelor de programare. Apariţia unor paradigme noi, specifice (logică, funcţională, distribuită, orientată obiect) a influenţat şi dezvoltarea construcţiei compilatoarelor prin optimizarea anumitor faze şi particularizarea altora în funcţie de caracteristicile paradigmei: invocarea la distanţă a funcţiilor, alocarea şi dealocarea automată a datelor, manipularea listelor, unificarea clauzelor (pentru a menţiona doar câteva).

1.3. Specificarea unui limbaj de programare

Limbajele de programare reprezintă, de fapt, notaţii pentru descrierea unor calcule. Un limbaj de programare ideal ar trebui să posede proprietăţi de *descriere* şi *implementare* a problemei, dar practic este dificil de îndeplinit ambele condiţii: o implementare eficientă ne conduce spre un limbaj de nivel scăzut, în timp ce descrierile se realizează mai uşor în limbaje de nivel înalt.

Specificarea unui limbaj de programare constă în trei etape importante:

- sintaxă;
- semantică;
- pragmatică.

Ca primă componentă a unui limbaj, sintaxa reprezintă un set de reguli care permit răspunsuri la întrebări de genul "Se poate face aşa?" sau "Cum se face?". Din punctul de vedere al scopului nostru crearea unui compilator pentru un limbaj de programare, sintaxa are rolul primordial în primele faze ale procesului de compilare, fapt ce justifică importanţa pe care i-o acordăm.

Setul de reguli corespunzătoare sintaxei unui limbaj trebuie descris folosind un alt limbaj de descriere, numit *metalimbaj*. Folosirea unui limbaj natural, de exemplu, limba română sau limba engleză, ca metalimbaj are dezavantajul de a fi ambiguu, fiind posibile diferite interpretări. Studiul formal al sintaxei aplicat unui limbaj de programare a înregistrat un progres notoriu în anii '60, când a apărut specificarea limbajului Algol60 (Naur, 1960, 1963), care folosea o notaţie simplă şi elegantă cunoscută sub numele de ***Backus-Naur-Form (BNF)*** [Nau60].

Backus-Naur-Form foloseşte următoarele obiecte:

- simboluri de bază (primitive) sau simboluri terminale, în general cuvinte cheie, operatori, separatori ai limbajului;
- variabile metalingvistice - simboluri neterminale, a căror semnificaţii corespund construcţiilor limbajului şi se notează între paranteze unghiulare <>;
- conective metalingvistice ::= (prin definiţie egal) şi | (alternative).

Exemplul 1.1. Numerele întregi pot fi descrise în BNF astfel:

$$
\begin{aligned}
< intreg > &::= < intreg_fara_semn > \mid \\
&\quad + < intreg_fara_semn > \mid \\
&\quad - < intreg_fara_semn > \\
< intreg_fara_semn > &::= < cifra > \mid \\
&\quad < cifra >< intreg_fara_semn > \\
< cifra > &::= 0|1|...|9
\end{aligned}
$$

BNF nu permite definerea unor condiţii de context şi e uneori greoaie pentru anumite descrieri. Este şi cazul exemplului, unde din motive de simplitate am omis condiţia ca prima cifră să fie nenulă. Pentru a spori uşurinţa de a citi aceste descrieri, pentru a elimina recursivitatea unor definiţii şi pentru a nota mai uşor alternative similare, Backus-Naur-Form a fost supus mai multor extinderi. Cele mai importante extensii sunt: cea datorată lui Wirth (1977) (extensie folosită în specificarea limbajelor Pascal şi Modula-2) şi standardul britanic EBNF (1981) [EBN].

Notaţiile folosite în EBNF[2] de către Wirth sunt:

- neterminalele sunt scrise ca şi cuvinte (fără paranteze unghiulare: Intregfarasemn în loc de <intreg_fara_semn>)
- terminale sunt notate între ghilimele, de exemplu "BEGIN" (şi de obicei cu majuscule)
- | este folosit, ca în BNF, pentru a nota alternanţa
- parantezele rotunde (şi) sunt folosite pentru a nota gruparea
- parantezele drepte [şi] notează apariţia opţională (zero ori sau o dată) a unui simbol sau grup de simboluri

[2]Extended-Backus-Naur-Form

- acoladele { şi } notează repetarea opţională (o dată sau mai multe ori) a unui simbol sau grup de simboluri
- = este folosit în locul lui ::=
- punctul . este folosit ca sfârşit de regulă
- (* *) sunt folosite pentru comentarii
- ϵ, secvenţa vidă, poate fi descrisă prin notaţia [ ]
- spaţiile sunt, în general, nesemnificative.

Exemplul 1.2. Numerele întregi descrise în notaţie EBNF:

Intreg = Semn Intregfarasemn .
Intregfarasemn = cifra { cifra } .
Semn = ["+" | "-"] .
cifra = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .

Standardul britanic EBNF are următoarele caracteristici:

- elementele unei descrieri sunt separate prin virgulă
- descrierile se termină cu ';'
- spaţiile sunt nesemnificative.

Exemplul 1.3. Descrierea unei declaraţii de variabile în limbajul Pascal prin următoarea reprezentare grafică.

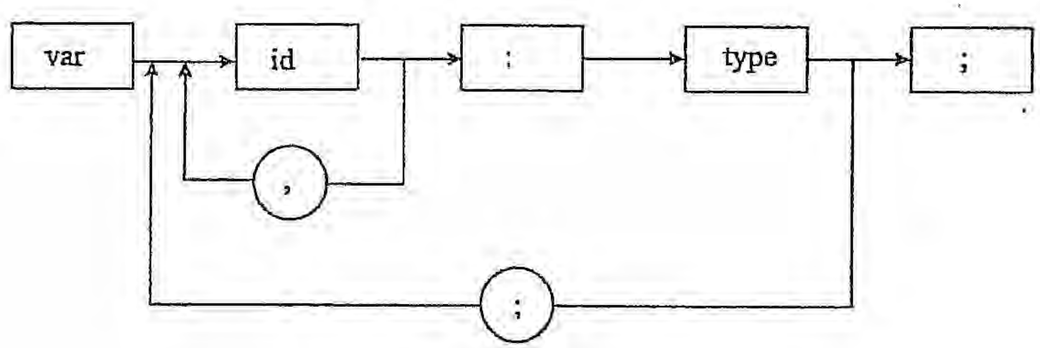

Figura 1.4: Declaraţia de variabile în limbajul Pascal

O altă modalitate de a descrie limbajele de programare se bazează pe *reprezentare grafică*, putând fi întâlnită şi sub denumirea de diagramă

de sintaxă. Au fost folosite la început pentru a defini sintaxa limbajelor Pascal, Modula-2 şi Fortran 77 şi au avantajul de a fi uşor de înţeles, mai ales pentru începători şi nespecialişti.

Regulile sintactice ale limbajului au forma unor diagrame de flux, căile posibile reprezentând secvenţe posibile de simboluri. O astfel de diagramă se parcurge începând de la stânga şi urmăreşte o cale care poate incorpora terminale sau neterminale, care au scopul de a realiza transferul spre alte diagrame. Unele reprezentări grafice folosesc terminalele scrise în cercuri şi neterminalele în dreptunghiuri pentru a face o mai clară distincţie între elementele limbajului şi construcţiile care trebuie descrise.

2. ANALIZA LEXICALĂ

Analiza lexicală[1], ca primă fază a compilării, trebuie să "transforme" programul sursă privit ca un text pentru prelucrări ulterioare. Programul sursă este privit ca un text în care dorim să identificăm anumite entităţi (*detectare*). Apoi, am vrea să împărţim aceste entităţi sau atomi pe categorii (*clasificare*) şi în final să le notăm într-o formă mai simplă (*codificare*).

Aşadar putem privi analiza lexicală ca pe o procedură:

Algoritmul 2.1 Analiza lexicală - versiunea 1

```
while (mai există caractere necitite în sursă) do
  detectare(atom);
  clasificare(atom);
  codificare(atom)
end while
```

2.1. Detectare şi clasificare

Să plecăm de la cazul unei propoziţii (secvenţe) din limbajul natural, de exemplu limba română: "Ana are mere." Analizorul lexical ar trebuie să *detecteze* următoarele:

- există trei entităţi (cuvinte): Ana, are, mere; în plus, separatorul dintre cuvinte este " " (spaţiu);
- începutul şi sfârşitul secvenţei - litera mare, respectiv ".".

Apoi, analizorul lexical execută clasificarea cuvintelor din secvenţă, având ca rezultat următoarele asocieri: Ana - substantiv, are - verb, mere - substantiv. Dacă propoziţia ar fi fost "Ioana are mere.", din punctul de vedere al analizorului lexical, rezultatul ar fi fost aproape identic: nu există diferenţă între Ana şi Ioana, ambele fiind substantive (cu rol de subiect în propoziţie).

Dacă considerăm acum o secvenţă de program (de exemplu, din limbajul C)

if (x==z) {x = y;}

[1]în engleză *scanning*

entităţile sunt: if, (, x, ==, z,), {, x, =, ;, }, iar clasificarea ţine de definiţia limbajului în care este scrisă secvenţa. În exemplul considerat se observă cele două dificultăţi importante care pot apărea în proiectarea unui analizor lexical:

1. La o primă vedere spaţiile ("albe") par neimportante în analiza lexicală: "Ana are mere." are acelaşi rezultat, dar nu acelaşi lucru se poate spune despre "An aareme re.".
2. Analizorul lexical trebuie să ia întotdeauna decizia corectă: "==" este un atom lexical de sine stătător şi nu doi atomi lexicali "=" consecutivi. Evitarea detectării greşite a acestor situaţii presupune folosirea unei tehnici de "cel mai lung şir posibil" sau de "look ahead" (o privire înainte). Astfel de situaţii sunt destul de rare în toate limbajele de programare şi pot fi tratate separat. Ele se întâlnesc în general la operatori şi la distincţia dintre atribuire şi test de egalitate.

Etapa de *clasificare* necesită ca fiecărui atom lexical să i se asocieze o singură clasă de apartenenţă. În cazul în care un atom nu poate fi asociat nici unei clase, adică nu respectă condiţiile de apartenenţă la clasa respectivă, atunci analiza lexicală se termină cu insucces: a fost detectată o eroare lexicală.

Clasele de atomi corespunzătoare oricărui limbaj de programare au menirea de a descrie toate entităţile importante dintr-un limbaj:

1. identificatori;
2. constante;
3. cuvinte rezervate (cuvinte cheie);
4. operatori;
5. separatori (delimitatori).

Observaţie: Diferenţa dintre cuvinte rezervate şi cuvinte cheie este că cele din urmă pot fi folosite şi pe post de identificatori, pe când cuvintelor rezervate nu li se poate atribui altă semnificaţie. Pentru a nu îngreuna analiza lexicală, vom considera că toate elementele de sintaxă ale limbajului sunt cuvinte rezervate.

Va trebuie să stabilim acum criterii pentru fiecare clasă de atomi lexicali, pentru a putea realiza clasificarea din cadrul analizei lexicale:

1. *Identificator*: şir de caractere care începe obligatoriu cu o literă, conţine doar caractere alfanumerice (litere sau cifre) şi eventual unele caractere speciale ("_", "#"), dar în mod categoric nu spaţiu.

2. *Constantă*: şir de caractere care conţine doar cifre, cu următoarele excepţii:

   - primul caracter poate fi "+" sau "-", în cazul constantelor cu semn;
   - poate conţine caracterul "." cel mult o dată, în cazul constantelor reale.

   Prima cifră este nenulă, exceptând constanta 0 sau numerele subunitare, de exemplu 0.56.

3. *Cuvânt rezervat*: "if", "else", "for", etc.

4. *Operator*: operatori aritmetici (+,-,*,/,...), operatori relaţionali (< ,>, ==, <=, ...), operatori logici (!, &&).

5. *Separator*:(,), ;, {, } etc.

Analizorul lexical este capabil să ia decizia corectă în privinţa detectării şi clasificării în unul din următoarele moduri:

- Cea mai simplă abordare ar fi considerarea unei strategii de tip selecţie multiplă, care să încerce pe rând potrivirea cu fiecare clasă. Ne-am putea gândi chiar la o îmbunătăţire astfel: dacă primul caracter este literă atunci atomul poate fi doar identificator sau cuvânt rezervat, dacă primul caracter este cifră atunci poate fi doar constantă.

  Pentru clasele cuvintelor rezervate, separatorilor şi operatorilor se testează apartenenţa la mulţimea respectivă. Pentru clasele identificatorilor şi constantelor este necesar să se verifice respectarea criteriilor corespunzătoare, care sunt descrise prin:

- expresii regulare;
- automate finite - pe care le vom prezenta în paragraful 2.3.2.

2.2. Codificare

În practică, o modalitate de a îmbunătăţi eficienţa compilatorului este de a micşora spaţiul ocupat de atomii lexicali. Fiecărui atom lexical i se ataşează un **cod**, care reprezintă un număr întreg mic şi care astfel ocupă mai puţin spaţiu decât atomii lexicali formaţi din unul sau mai multe

caractere. Tabelul următor prezintă o astfel de codificare parţială. În realitate, fiecare atom lexical are un cod ataşat.

Atom	Cod
Identificator	1
Constantă	2
+	3
-	4
...	...
==	10
(	11
)	12
{	13
}	14
=	15
;	16
if	17
else	18
for	19
while	20
do	21
...	...

Proiectarea unui compilator impune, după cum s-a mai precizat şi în capitolul precedent, o echivalenţă între formele intermediare construite între fazele compilării. Astfel, programul sursă trebuie să se poată reconstitui pe baza rezultatului analizei lexicale. Pe de altă parte, faza următoare - analiza sintactică - are nevoie doar de informaţii sintactice. De exemplu, pentru un atom lexical "x", analizorul sintactic trebuie să ştie că este un identificator.

În acest scop, analiza lexicală va furniza doar aceste informaţii, celelalte detalii fiind depuse într-o listă "specială", numită ***tabela de simboluri (TS)***. Aceste tipuri de atomi lexicali sunt particulare pentru fiecare program în parte şi doar ele vor fi depuse în tabela de simboluri. Organizarea tabelei de simboluri precum şi operaţiile care se execută asupra ei sunt descrise în paragraful 2.4. Este totuşi important să ştim că un nume simbolic (identificator, constantă) apare o singură dată în tabela de simboluri.

Astfel rezultatul analizei lexicale, numit ***forma internă a programului (FIP)***, reprezintă o secvenţă de perechi, în care primul element este codul corespunzător, iar al doilea element al perechii îl reprezintă poziţia

din tabela de simboluri (în practică, un pointer spre poziţia respectivă). Pentru celelalte tipuri de atomi (cuvinte rezervate, operatori, separatori) al doilea câmp se completează cu o valoare inexistentă în tabelă (de obicei, 0 sau -1).

Exemplul 2.1. Să considerăm următoarea secvenţă de program supusă analizei lexicale:

if (x==z) {x = y;}

Forma internă corespunzătoare este:

Cod Atom	Poziţie în TS
17	0
11	0
1	1
10	0
1	3
12	0
13	0
1	1
15	0
1	2
16	0
14	0

iar tabela de simboluri va conţine trei câmpuri: pe prima poziţie x, apoi y, iar pe ultima poziţie z.

Putem să rafinăm acum algoritmul corespunzător analizei lexicale astfel:

Algoritmul funcţionează în următorul mod: după ce un atom a fost detectat (*detectare(atom)*), se încearcă clasificarea lui, prin cele trei instrucţiuni if. Se observă că se testează întâi apartenenţa atomului la elementele limbajului (cuvânt revervat, operator, separator) pentru a nu face o clasificare eronată: orice cuvânt rezervat respectă definiţia unui identificator (conţine doar litere). O altă observaţie importantă este că se folosesc instrucţiuni condiţionale cu două alternative, continuarea clasificării realizându-se pe o ramură else. Această abordare are două semnificaţii: se implementează condiţia ca un atom să aparţină cel mult unei clase şi permite detectarea erorilor lexicale, cazurile în care un atom nu poate fi clasificat.

După clasificarea atomului, procedura *genFIP(cod,loc)* va completa forma internă a programului cu codul atomului *cod* şi locaţia *loc* din

Algoritmul 2.2 Analiza lexicală - versiunea 2

```
INPUT: program sursă, codificare
OUTPUT: FIP, TS, erori
while (mai există caractere necitite în program sursă) do
  detectare(atom);
  if este cuvânt rezervat sau operator sau separator then
    genFIP(cod,0)
  else
    if este identificator then
      indice := poz(atom,TS);
      genFIP(cod_identificator, indice);
    else
      if este constantă then
        indice := poz(atom,TS);
        genFIP(cod_constanta, indice);
      else
        mesaj:"Eroare lexicala la atomul:", atom
      end if
    end if
  end if
end while
```

tabela de simboluri (0 pentru atomii care nu apar în tabelă). Funcţia *poz(atom,TS)* va returna poziţia atomului *atom* din tabela de simboluri *TS* astfel: dacă *atom* există în *TS* se va returna poziţia pe care se află (căutare), dacă nu este găsit se va insera *atom* în *TS* (adăugare). Astfel ambele operaţii asupra tabelei de simboluri sunt implementate sub un unic apel.

2.3. Modelul analizorului lexical

Construirea unui model pentru analiza lexicală se bazează pe două observaţii importante din teoria limbajelor formale. În primul rând, lexicul oricărui limbaj (natural sau de programare) poate fi specificat printr-o gramatică regulară. Gramaticile regulare sunt simple, uşor de înţeles şi există algoritmi eficienţi corespunzători lor. În al doilea rând, clasa gramaticilor regulare este echivalentă cu cea a expresiilor regulare şi cu cea a automatelor finite: pentru orice gramatică regulară există o expresie regulară care să descrie limbajul generat de gramatică şi există un automat finit echivalent.

2.3.1. Expresii regulare

Dacă considerăm clasa identificatorilor, o gramatică corespunzătoare este:

$$
\begin{aligned}
< id > &\rightarrow < litera > C \\
< litera > &\rightarrow A|...|Z|a|...|z \\
C &\rightarrow < alfanum > | < alfanum > C \\
< alfanum > &\rightarrow < litera > | < cifra > | < special > \\
< cifra > &\rightarrow 0|1|...|9 \\
< special > &\rightarrow _|\#
\end{aligned}
$$

Folosind expresiile regulare vom putea descrie fiecare clasă de atomi lexicali. Pentru aceasta, vom reaminti ce este o expresie regulară.

Definiţia 2.1. *[AU72] Fiind dat un alfabet Σ, expresiile regulare peste acest alfabet se definesc inductiv astfel:*

- *ϕ este expresia regulară corespunzătoare mulţimii vide $\emptyset$*
- *ϵ este expresia regulară corespunzătoare mulţimii ce conţine secvenţa vidă ϵ*
- *a este expresia regulară corespunzătoare mulţimii $\{a\}$*
- *dacă a şi b sunt expresii regulare, corespunzând respectiv mulţimilor A şi B, atunci:*
  - *$a + b$ este expresia regulară corespunzătoare mulţimii $A \cup B$ (reuniune);*
  - *ab este expresia regulară corespunzătoare mulţimii AB (concatenare);*
  - *a^* este expresia regulară corespunzătoare mulţimii A^* (închidere reflexiv-tranzitivă).*

Claselor de atomi lexicali le corespund următoarele expresii regulare:

1. Constante întregi, fără semn:

$$
\begin{aligned}
cifra_nenula &= '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9' \\
cifra &= cifra_nenula + '0' \\
intreg &= cifra_nenula\ cifra^* + cifra
\end{aligned}
$$

(o constantă întreagă este fie o secvenţă de cifre din care prima este nenulă, fie o cifră).

2. Identificatori:

$$
\begin{aligned}
litera &= 'A' + 'B' + \cdots + 'Z' + 'a' + \cdots + 'z' \\
special &= '_' + '\#' \\
identificator &= litera(litera + cifra + special)^*
\end{aligned}
$$

(un identificator este o secvenţă de caractere alfanumerice sau speciale, din care prima este obligatoriu literă)

3. Cuvinte rezervate:

$rezervat = 'if' + 'else' + 'for' + 'do' + \cdots$

4. Operatori:

$operator = '+' + '-' + '*' + \cdots + '<' + '>' + '==' + \cdots + '\&\&' + '!' + \cdots$

5. Separatori:

$separator = '(' + ')' + '\{' + '\}' + ';' + \cdots$

Analiza lexicală poate fi privită astfel: pentru fiecare atom lexical detectat se încearcă potrivirea cu o expresie regulară. Dacă atomul respectă definiţia uneia dintre expresiile regulare specificate, atunci atomul aparţine clasei respective, altfel a apărut o eroare lexicală.

Dată fiind echivalenţa expresiilor regulare cu automate finite, putem să descriem un model pentru analiza lexicală bazat pe automate finite. Ideea de bază ar fi să construim câte un automat finit pentru fiecare clasă de atomi lexicali şi apoi un automat finit care să reprezinte reuniunea acestora şi pe baza căruia se va putea face clasificarea atomilor lexicali întâlniţi în text.

2.3.2. Automate finite

Definiţia 2.2. *[AU72] Un automat finit M este un 5-tuplu $M = (Q, \Sigma, \delta, q_0, F)$ unde:*

- *Q este o mulţime finită de stări;*
- *Σ este un alfabet finit;*
- *δ este funcţia de tranziţie: $\delta : Q \times \Sigma \rightarrow P(Q)$;*
- *q_0 reprezintă starea iniţială a automatului finit;*
- *$F \subseteq Q$ reprezintă mulţimea stărilor finale.*

Echivalenţa dintre mulţimea expresiilor regulare şi mulţimea limbajelor generate de automate finite ne permite să exprimăm entităţile lexicale prin automate finite. Funcţia de tranziţie pentru starea curentă şi un anumit simbol de intrare vor modela mecanismul prin care se detectează un atom lexical. Figura 2.1 reprezintă un automat finit corespunzător unui atom lexical identificator, unde l notează orice literă, iar c notează orice cifră.

Figura 2.1: Automat finit pentru identificatori

În mod asemănător putem construi un automat finit pentru constante întregi (fără semn), ca în figura 2.2:

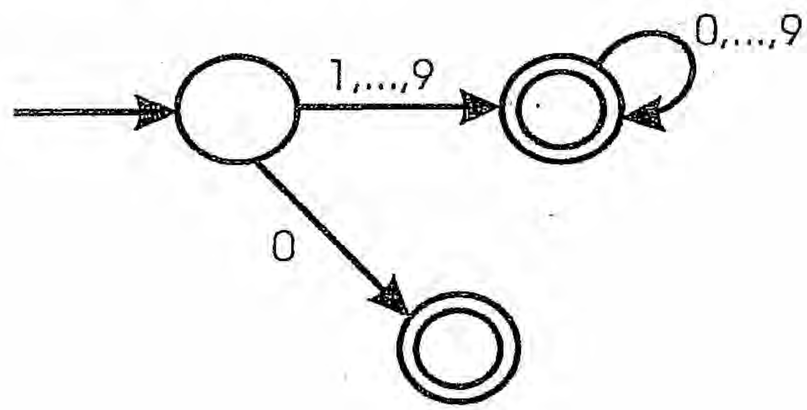

Figura 2.2: Automat finit pentru constante întregi fără semn

Analizorul lexical va fi construit ca automatul finit corespunzător reuniunii automatelor finite corespunzătoare tuturor atomilor lexicali, obţinându-se automatul din figura 2.3.

2.4. Tabela de simboluri

Toate informaţiile colectate de compilator în legătură cu numele simbolice care apar în programul sursă se află în *tabela de simboluri*.

Într-un compilator poate exista una din următoarele variante:

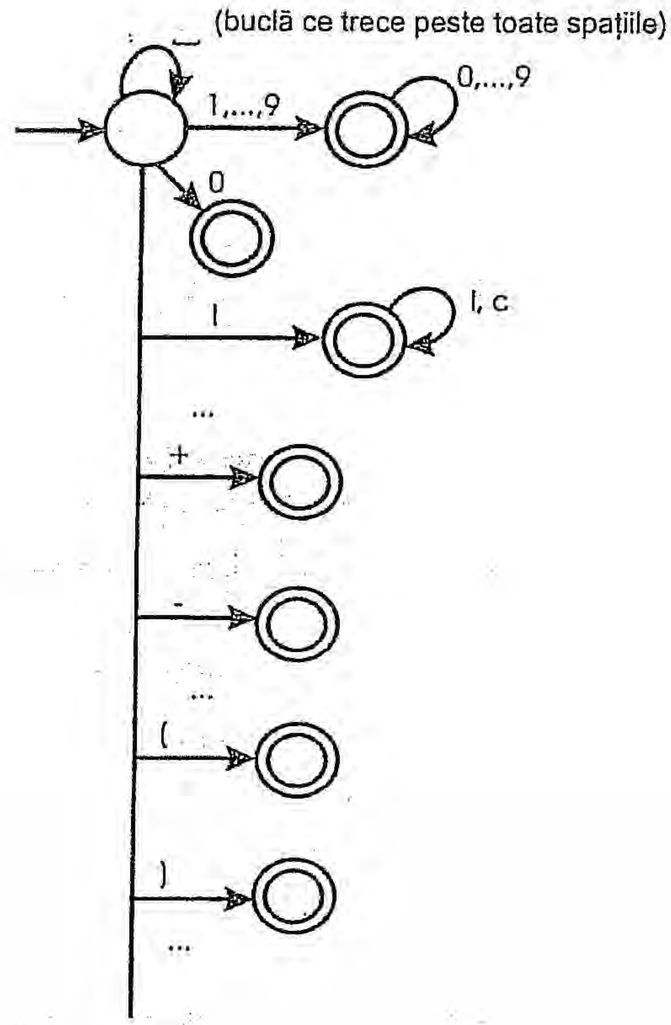

Figura 2.3: Analizorul lexical reprezentat ca automat finit

- tabelă de simboluri *unică*, care conţine toate numele simbolice, indiferent de tipul lor;

  sau

- tabele de simboluri *distincte* pentru fiecare tip de nume simbolic; pentru un limbaj simplu vom avea o tabelă a identificatorilor şi o tabelă a constantelor.

Informaţiile reţinute în tabela de simboluri diferă de la un compilator la altul şi, cu excepţia numelui, sunt în general determinate în fazele ulterioare ale compilării. Din aceste informaţii, compilatorul trebuie să obţină: tipul, adresa de memorie, valoarea.

Pentru fiecare nume simbolic întâlnit în program va exista o intrare

unică în tabela de simboluri care permite accesul la aceste informaţii. De aceea, inserarea se face doar dacă operaţia de căutare se termină cu insucces.

Din studiul algoritmului de analiză lexicală se observă că operaţiile care se efectuează asupra tabelei de simboluri sunt căutarea şi inserarea. Din acest motiv, forma de organizare a tabelei de simboluri va trebui să ţină cont de faptul că dorim ca aceste operaţii să se desfăşoare într-un timp cât mai scurt. Există mai multe posibilităţi de organizare, fiecare dintre ele îmbunătăţind complexitatea algoritmului de căutare:

1. tabel neordonat - inserarea se face în ordinea în care simbolurile sunt întâlnite în program; vom avea o căutare secvenţială, total ineficientă, cu un timp de execuţie $\Theta(n)$;

2. tabel ordonat (ordine lexicografică) - inserarea se face astfel încât tabelul să rămână ordonat; în acest caz, se poate folosi căutarea binară cu un timp de execuţie mai bun($\Theta(lgn)$);

3. arbore binar de căutare echilibrat - pentru care căutarea necesită un timp de execuţie $\Theta(lgn)$;

4. tabele de dispersie[2][CLR00] - pentru care căutarea are timpul de execuţie $\Theta(1)$, ceea ce sugerează că această organizare este cea mai eficientă pentru tabele de simboluri.

2.4.1. Tabele de dispersie

Cea mai folosită organizare a tabelei de simboluri este tabela de dispersie, care a fost menţionată prima dată de H.P.Luhn şi A.D.Lin în 1953 şi folosită de A.L.Samuel,G.M.Amdahl şi E.Boehm în asamblorul IBM 701 în 1954 [Hop69]. Această metodă foloseşte o funcţie, numită *funcţie de dispersie*, notată h, pentru a calcula locaţia la care se va memora un element: dacă prin adresare directă un nume k se memorează în locaţia k, într-o tabelă de dispersie, acesta va fi memorat în locaţia $h(k)$.

Prin folosirea funcţiei de dispersie, mulţimea identificatorilor este partiţionată într-un număr mare, dar finit, de clase. Timpul de căutare se reduce la timpul de calcul al indexului identificatorului şi la timpul de căutare într-o singură clasă de dispersie, cea corespunzătoare indexului şi nu în mulţimea tuturor indentificatorilor. Dacă funcţia de dispersie este bine aleasă, iar numărul indecşilor destul de mare, este foarte probabil ca într-un program nu foarte lung să se disperseze pentru fiecare index cel mult un identificator.

[2]hash table în engleză

Va trebui să particularizăm funcţia de dispersie pentru cazul în care cheile sunt identificatori(şiruri de caractere) sau constante. În cazul în care se lucrează cu o singură tabelă de simboluri, constantele vor fi tratate şi ele ca şi şiruri de caractere, asigurând astfel compararea tuturor numelor simbolice între ele. Dacă notăm:

K = mulţimea tuturor numelor simbolice acceptate de limbaj

A = mulţimea poziţiilor din tabel ($|A| = m$)

$f : K \rightarrow A$

atunci $h(k) = (val(k)\ mod\ m) + 1$

unde $val(k)$ se consideră suma codurilor ASCII a caracterelor ce compun numele simbolic.

O funcţie de dispersie "bună" ar trebui să mapeze identificatori la valori întregi în mod unic. Este dificil de găsit o astfel de funcţie. McKenzie, Harries şi Bell [MHB90] au analizat performanţa mai multor funcţii de dispersie şi au ajuns la concluzia că un rezultat performant este garantat de următoarea funcţie:

$h_0 = 0$

$h_i = k \times h_{i-1} = c_i, \forall i = 1, n$

$H = bit(h_n, 30) MOD\ N$

unde: n este lungimea identificatorului, c_i este codul ASCII al celui de-al i-lea caracter din identificator, iar $bit(h_n, 30)$ reprezintă cei mai semnificativi 30 de biţi ai întregului h. Se sugerează alegerea următoarelor valori: pentru k 613, iar pentru N 1008.

Dacă pentru un index există mai mult de un identificator avem o situaţie de *coliziune*. Rezolvarea coliziunilor se face prin:

- adresare deschisă;
- înlănţuire (sau zonă auxiliară).

Tabele de dispersie cu înlănţuire

Prin înlănţuire toate numele simbolice care se dispersează într-o anumită locaţie sunt memorate într-o listă înlănţuită. Pentru aceasta, tabela de simboluri va conţine pentru fiecare înregistrare câte un pointer spre această listă. Deşi se măreşte spaţiul alocat tabelei, prin această metodă nu se limitează numărul elementelor care pot fi memorate, alocarea dinamică permiţând inserarea unui număr nelimitat de nume simbolice [CLR00].

Timpul de căutare şi inserare se reduce la căutarea, respectiv inserarea în lista înlănţuită, după cum se observă din algoritmul 2.3.

Exemplul 2.2. Presupunem că într-un program se întâlnesc următoarele nume simbolice (în această ordine) iar valoarea funcţiei de dispersie asociate este cea din paranteză:

Algoritmul 2.3 Poz_dispersie_înlănţuire

```
INPUT: nume, T - tabela de dispersie
OUTPUT: poz - poziţie în T
i = h(nume);
if T[i].info = nume then
  poz := p
else
  if T[i].info = "" then
    T[i].info := nume;
    T[i].leg := nil;
    poz := i
  else
    p := T[i].leg; poz :=0;
    while (p <> nil) sau (poz <> 0) do
      if p^.info = nume then
        poz :=p
      else
        p := p^.leg
      end if
    end while
    if poz = 0 then
      new(q); q^.info := nume; q^.leg := T[i].leg;
      T[i].leg := q;
      poz := q;
    end if
  end if
end if
```

NODE (1), AN (3), FUNCTION (9), STORE (2), B (9), ADD (3), BRAN (9), PARAMETER (9).

Tabela de dispersie cu înlănţuire corespunzătoare este ilustrată în figura 2.4 şi se completează astfel: poziţia 1 este liberă se va insera NODE, poziţia 3 este liberă se va insera AN şi aşa mai departe. Când se detectează ADD, poziţia 3 este deja ocupată. Se va actualiza pointerul corespunzător poziţiei 3 pentru a referi o locaţie unde va fi memorat ADD şi se va returna această locaţie. La fel se va proceda şi pentru celelalte nume simbolice care ar trebui să se disperseze la poziţia 9.

Tabelă de dispersie cu adresare deschisă

Toate elementele sunt memorate în interiorul tabelei, astfel încât se foloseşte la maximum spaţiul tabelei. Avantajul acestei metode este că

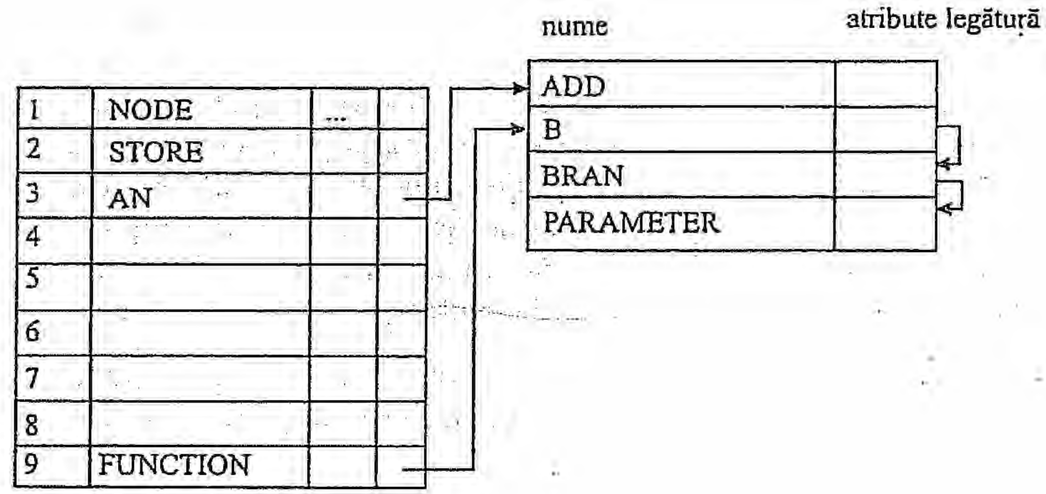

Figura 2.4: Tabelă de dispersie cu înlănţuire

nu se foloseşte spaţiu suplimentar şi se evită folosirea pointerilor [CLR00]. Dezavantajul principal constă în faptul că numărul de elemente care se pot insera în tabelă este limitat de dimensiunea acesteia. În momentul în care tabela de dispersie este plină, inserarea unui nou nume simbolic presupune "mărirea spaţiului", existând mai multe soluţii pentru acest impas:

- crearea unei tabele adiţionale după dispersie;
- modificarea funcţiei de dispersie atfel încât să se poată redimensiona tabela, fără a modifica poziţiile completate;
- mărirea dimensiunii tabelei şi redispersarea tuturor elementelor în noua tabelă.

Adresarea deschisă se bazează pe următoarea idee: dacă poziţia (obţinută ca valoare a funcţiei de dispersie) este liberă atunci se inserează elementul pe acea poziţie, dacă nu, atunci se caută următoarea poziţie liberă. Şirul de poziţii examinate se calculează în funcţie de elementul care se inserează. Dacă se ajunge la dimensiunea maximă a tabelei, căutarea se reia de la începutul tabelei.

Algoritmul 2.4 efectuează căutarea şi inserarea unui nume în tabela de dispersie T, returnând poziţia numelui în tabelă.

Exemplul 2.3. Considerând aceleaşi nume simbolice ca în exemplul 2.2, şi anume

NODE (1), AN (3), FUNCTION (9), STORE (2), B (9), ADD (3), BRAN (9), PARAMETER (9),

Algoritmul 2.4 Poz_dispersie_adresare_deschisă

```
INPUT: nume, T - tabela de dispersie
OUTPUT: poz - poziţie în T
i = h(nume);
j := 0; {contorizează poziţiile vizitate pt. a vedea dacă tabela e plină}
poz := 0;
repeat
  if T[i] = nume then
    poz := i
  else
    if T[i] = "" then
      T[i] := nume;
      poz := i
    else
      i := (i + 1) mod dim;
      j := j+1
    end if
  end if
until (poz ≠ 0) sau (j=dim)
if j = dim then
  EROARE "depăşire dimensiune tabelă"
end if
```

(valorile din paranteză reprezentând valorile funcţiei de dispersie), tabela de dispersie cu adresa deschisă corespunzătoare este cea din figura 2.5: se inserează fără probleme NODE, AN, FUNCTION şi STORE deoarece locaţiile sunt libere. În momentul în care dorim să inserăm B în tabelă, locaţia 9 este ocupată, vom căuta următoarea poziţie liberă care este 10 şi se va returna această poziţie. Pentru numele simbolic ADD, următoarea poziţie liberă este 4. Poziţia pe care ar trebui să inserăm BRAN este 9, dar este ocupată, următoarea poziţie (10) este şi ea ocupată şi în final, poziţia 11 liberă poate fi folosită pentru acest nume simbolic. Pentru ultimul nume simbolic, PARAMETER, căutarea unei locaţii libere va întâlni marginea superioară a tabelei, caz în care căutarea este reluată de la prima poziţie, oprindu-se la poziţia liberă 5.

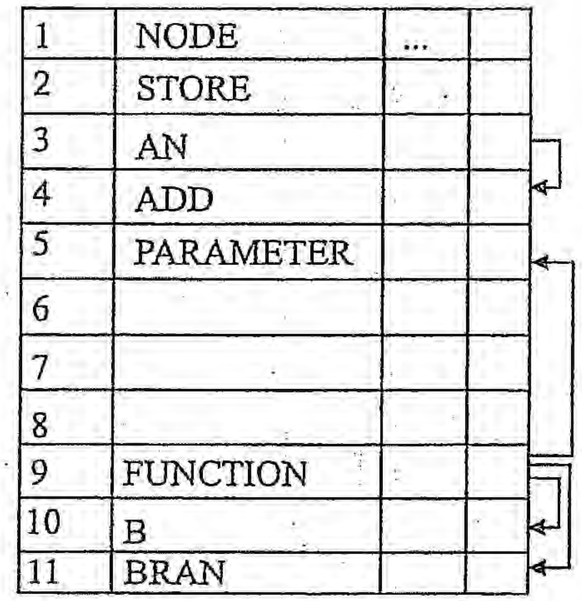

Figura 2.5: Tabelă de dispersie cu adresare deschisă

2.5. Generator de analiză lexicală – lex

Fazele din partea de analiză a unui compilator se bazează pe algoritmi generali, care nu depind de caracteristicile limbajului. De aceea construcţia unui analizor lexical se poate baza pe un program care implementează un algoritm generic, iar specificaţia limbajului, a regulilor de formare a atomilor lexicali să fie realizată separat de acest program. O astfel de abordare este benefică, putând folosi un program de analiză lexicală pentru mai multe limbaje, idee care stă la baza *generatoarelor de analiză lexicală*.

Cel mai cunoscut generator de analiză lexicală este *lex* [LMB92, LY, LS75], un utilitar existent sub sistemul de operare Unix, care foloseşte un fişier de specificaţie în care descrierea atomilor lexicali se face prin expresii regulare. Lex a fost proiectat ca un procesor lexical al caracterelor din fluxul de intrare, poate fi folosit pentru a realiza analiza lexicală ca fază a unui compilator, dar aplicabilitatea lui este mult mai mare: în prelucrări de analiză şi statistică a fişierelor text, în analiza fişierelor de comenzi şi multe altele.

Modul de funcţionare, descris în figura 2.6, se bazează pe fişierul de specificare, care este transmis generatorului, obţinând ca rezultat un program în limbajul C, numit *lex.yy.c*, care realizează analiza lexicală. Acest program este apoi compilat şi link-editat şi furnizându-i programul sursă îl analizează din punct de vedere lexical.

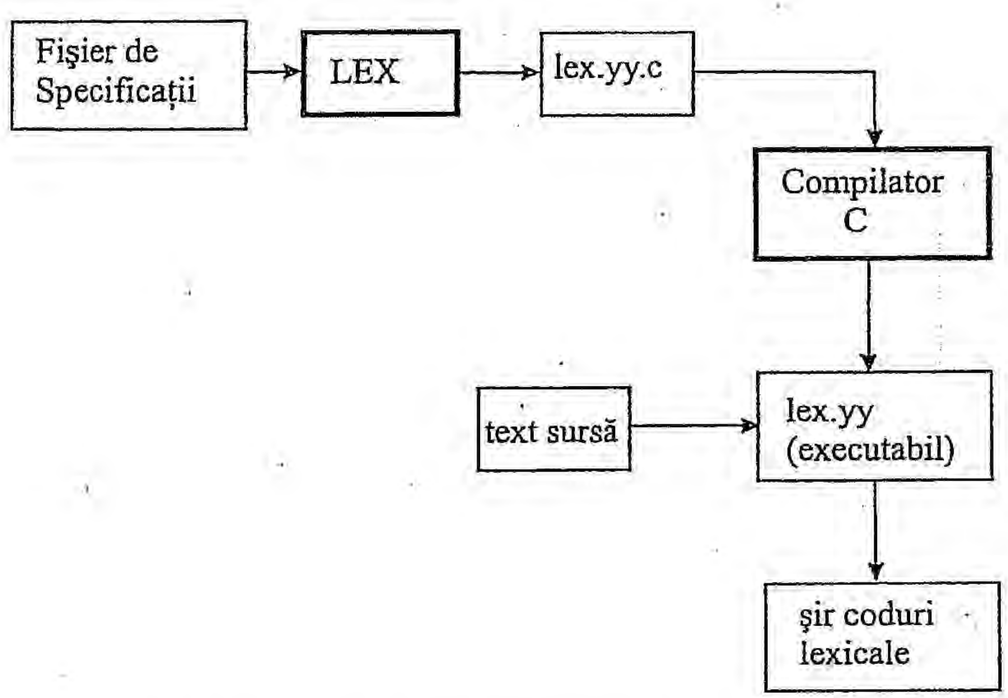

Figura 2.6: Funcţionarea generatorului de analiză lexicală *lex*

Flexibilitatea acestui generator constă în faptul că structura fişierului de specificaţii permite:

- recunoaşterea entităţilor lexicale din fişierul de intrare conform expresiilor regulare specificate şi, în plus, în momentul detectării lor se pot asocia acţiuni;
- utilizatorului să îşi scrie propriul cod, particularizat propriilor cerinţe, care va fi adăugat programului de analiză lexicală.

2.5.1. Formatul fişierului de specificaţie

Fişierul ce conţine specificaţia este un fişier text care poate avea orice nume acceptat de sistemul de operare. Din considerente istorice se recomandă ca fişierul de specificaţie să aibă extensia *.lxi*. Conţine 3 secţiuni separate de câte o linie ce conţine %%:

definiţii
%%
reguli
%%

cod utilizator

Secţiunea de definiţii:

Această secţiune cuprinde:

- includere de cod: utilizatorul îşi poate include propriile sale biblioteci sau propriile sale declaraţii respectând sintaxa limbajului C;
- condiţii de start;
- declaraţii nume (pentru a simplifica specificarea analizorului lexical), de forma

  nume definiţie

  unde:

  nume este un cuvânt compus din una sau mai multe litere, cifre, '_' sau '-', cu observaţia că primul caracter trebuie să fie literă sau '_' şi să se afle pe prima poziţie a liniei;

  definiţie este o expresie regulară şi se consideră că începe cu primul caracter non-spaţiu de după nume şi ţine până la sfârşitul liniei.

Secţiunea de reguli

Scopul principal al acestei secţiuni este acela de a asocia acţiuni semantice cu expresiile regulare. În plus, ea mai poate conţine cod C definit de utilizator. Pentru asocierea acţiunilor semantice cu expresiile regulare se utilizează o construcţie de forma:

şablon acţiune

unde:

şablon este o expresie regulară, al cărei prim caracter trebuie să se afle pe prima poziţie a liniei;

acţiune este o secvenţă formată din una sau mai multe instrucţiuni C care trebuie să înceapă pe aceeaşi linie cu şablonul. Dacă secvenţa e formată din mai multe instrucţiuni acestea se vor închide între acolade. În particular, acţiunea poate fi şi instrucţiunea vidă.

Semnificaţia unei asemenea construcţii este că analizorul generat va executa secvenţa descrisă ca acţiune atunci când va recunoaşte în textul sursă un şir care se potriveşte cu şablonul. Dacă acţiunea este redată prin instrucţiunea vidă, la recunoaşterea şirului descris de şablonul în cauză, analizorul nu va executa nimic (de fapt, va trece pur şi simplu la delimitarea următorului atom).

Secţiunea de cod utilizator

Este o secţiune opţională (daca lipseşte, atunci şi separatorul %% care urmează după secţiunea de reguli poate lipsi). Dacă este prezentă, atunci conţinutul ei, care constă în cod C utilizator, este copiat fără modificări la sfârşitul fişierului lex.yy.c.

În mod normal, în secţiunea de cod utilizator pot să apară:

- funcţia main() care să conţină apel(uri) la *yylex*(), funcţia principală care realizează analiza lexicală şi care este definită în programul *lex.yy.c*, dacă se doreşte ca analizorul să lucreze în mod autonom (de exemplu pentru a-l testa);
- alte funcţii apelate din *yylex*(); se menţionează că pentru aceste funcţii trebuie să avem în codul utilizator din secţiunea de definiţii: fie prototipurile, fie directive #include ale headerelor ce conţin prototipurile.

2.5.2. Mod de funcţionare

Lansarea în execuţie se face utilizând comenzile:

$ lex [opţiuni] [nume_fişier_specificaţii]

unde nume_fişier_specificaţii este fişierul de intrare (implicit stdin). Dacă execuţia generatorului lex s-a terminat cu succes (fără erori), atunci directorul curent trebuie să conţină un fişier *lex.yy.c*

$ gcc lex.yy.c -o lex_meu

obţinând fişierul *lex_meu* ca executabil şi apoi, pentru a executa analiza lexicală:

$ ./lex_meu prog_sursa

Pentru şabloanele recunoscute se aplică acţiunile corespunzătoare din specificaţie (secţiunea de reguli), care pot genera text specific în fişierul *yyout* sau pot returna coduri lexicale. Textul din fişierul sursă care nu se potriveşte cu nici un şablon este copiat automat în *yyout* (în cazul în care *yyout* nu a fost redefinit, el este implicit stdout). Dacă pentru un text se potrivesc mai multe şabloane, se alege potrivirea cea mai lungă. Dacă pentru un text se potrivesc mai multe şabloane de aceeaşi lungime, se alege prima potrivire din specificaţie, în ordine textuală. Textul corespunzător potrivirii este copiat în variabila globală *yytext* iar lungimea acestuia este memorată în variabila globală *yyleng*. Când analizorul întâlneşte marcajul de sfârşit de fişier, el verifică rezultatul funcţiei *yywrap*(). Rezultat fals (zero) înseamnă că analiza trebuie să continue cu noul fişier desemnat de *yyin*. Un rezultat nenul al funcţiei *yywrap*() determină sfârşitul analizei şi returnarea valorii 0 spre apelantul lui *yylex*().

Exemplul 2.4. Următorul fişier de specificaţii va analiza un fişier sursă C şi va detecta atomii lexicali, returnând clasa căruia îi aparţine. În plus, va detecta erorile lexicale, tipărind un mesaj corespunzător şi va elimina spaţiile albe.

```
LITERA [a-zA-Z_]
CIFRA [0-9]
%%
" = "|"! = "|" < "|" > "|" == "|"/"|" - "|" * "|" + "|" << "|" >> " {
    printf("%s (%s)\n", yytext, "Operator");
    getch();
}
";"|"["|"]"|"("|")"|"{"|"}" {
    printf("%s (%s)\n", yytext, "Separator");
    getch();
}
true|false {
    printf("%s (%s)\n", yytext, "Constanta booleana");
    getch();
}
main|if|then|else|endif|while|do|endwhile|bool|int|array|cin|cout {
    printf("%s (%s)\n", yytext, "Cuvant rezervat");
}
{LITERA}+{CIFRA}* {
    if(strlen(yytext)>8) printf("%s (%s)\n", yytext, "Lungime id > 8");
    else printf("%s (%s)\n", yytext, "Identificator");
}
{CIFRA}+{LITERA}+{
    printf("%s (%s)\n", yytext, "Identificator inceput cu cifra");
}
0—[1-9]{DIGIT}* {
    printf("%s (%s) %d\n", yytext, "Constanta intreaga", atoi(yytext));
}
[\t\n ]+
. printf("%s (%s)\n", yytext,"Caracter necunoscut");
%%
main( argc, argv )
int argc;
char **argv;
{ ++argv, --argc;
    if ( argc > 0 ) yyin = fopen( argv[0], "r" );
    else
    {
    yyin = fopen( "test", "r" );
    }
```

```
    yyout=fopen("rez","w");
    yylex();
}
```

2.6. Relaţia dintre analizor lexical şi analizor sintactic

Rolul analizei lexicale este asemănător cu cel al analizei sintactice: detectare, clasificare, traducere. Separarea analizei lexicale de analiza sintactică este justificată de următoarele aspecte:

1. Separarea conduce la o proiectare mai eficientă a fiecărei părţi (principiul proiectării modulare).
2. Prelucrarea şirului de caractere necesită operaţii ce consumă mult timp (preluarea textului de pe suport extern, accesul la fiecare caracter, căutări în tabele, etc.), spre deosebire de celelalte faze; astfel, pentru o analiză lexicală eficientă se va prefera un limbaj specific acestor operaţii de prelucrare de caractere, iar pentru celelalte faze limbaje de nivel înalt.
3. Specificarea atomilor lexicali este foarte simplă, ea poate fi exprimată printr-o gramatică regulară, simplificând procesul de analiză. În cadrul acestui proces se va simula un automat finit şi nu unul push-down ca în cadrul analizei sintactice.
4. În urma analizei lexicale textul este "curăţat" din mai multe puncte de vedere:
   - numărul atomilor lexicali este, de regulă, mult mai mic decât numărul caracterelor textului;
   - se elimină spaţiile inutile şi alte caractere corespunzătoare alinierii textului sursă;
   - se elimină comentariile.
5. Asigură portabilitatea algoritmului de compilare.

În proiectarea compilatoarelor, interdependenţa dintre cele două faze duce la implementarea uneia dintre următoarele decizii:

- analizor lexical independent de analizorul sintactic - un pas separat al procesului de compilare, legătura fiind realizată printr-un fişier sau zonă de memorie ce conţine rezultatele analizei lexicale;

- analizor lexical comandat de analizorul sintactic - ca o rutină a analizorului sintactic, fiind apelat ori de câte ori analizorul sintactic are nevoie de un nou atom.

3. ANALIZA SINTACTICĂ

Analiza sintactică[1] se defineşte, la modul cel mai general, ca fiind procesul de structurare a unei secvenţe în raport cu nişte reguli. Este o denumire generică întâlnită în foarte multe domenii: lingvistică (analiza textelor, traduceri), informatică (construcţia compilatoarelor, descrierea bazelor de date, inteligenţă artificială), chimie, pentru a numi doar câteva.

Ca fază a procesului de compilare, analiza sintactică se poate defini ca acea componentă care are ca intrare secvenţa de atomi lexicali şi determină dacă aceasta este corectă din punct de vedere sintactic, caz în care se construieşte *arborele de analiză sintactică*.

Această definiţie sugerează elementele de care vom avea nevoie în această fază:

- un limbaj de descriere a secvenţelor valide (corecte);
- o metodă pentru a distinge între secvenţe corecte şi incorecte.

Este evident că instrumentul folosit în analiza lexicală, şi anume gramaticile regulare, nu sunt suficient de puternice pentru a exprima constrângerile pe care sintaxa unui limbaj le impune asupra atomilor. De aceea, analiza sintactică se bazează pe gramatici independente de context, $G = (N, \Sigma, P, S)$, în care [AU72]:

- Σ - alfabetul gramaticii va cuprinde elementele limbajul (atomii);
- N - mulţimea de neterminale reprezintă intuitiv construcţiile sintactice ale limbajului;
- P - mulţimea de producţii reprezintă reguli de formare a construcţiilor sintactice, unde producţiile au forma $A \rightarrow \alpha, A \in N, \alpha \in (N \cup \Sigma)^*$;
- S - simbolul de start, corespunde intuitiv generării unui program în limbajul ales.

Astfel, problema determinării corectitudinii sintactice a secvenţei de atomi lexicali se reduce la a verifica dacă o anumită secvenţă $w \in \Sigma^*$ aparţine limbajului: $w \in L(G)$ [AU72]. Funcţionarea analizei sintactice poate fi descrisă prin intermediul automatelor push-down, datorită

[1] în engleză *parsing*

echivalenţei lor cu gramatici independente de context. Deşi reprezintă un instrument matematic puternic, automatele push-down sunt relativ complexe, greu de reprezentat şi manipulat. Din aceste considerente, pentru analizoarele sintactice prezentate vom ocoli definirea exactă a automatelor corespunzătoare, prezentând o formă simplificată a lor, care conţine strict informaţiile necesare în analiza sintactică, simplificând atât reprezentarea cât şi funcţionarea lor.

În funcţie de strategia aleasă pentru construcţia arborelui, analizoarele sintactice se clasifică în felul următor:

- direcţia de construire a arborelui: *descendentă* - de la rădăcină spre frunze sau *ascendentă* - de la frunze spre rădăcină;
- număr de posibilităţi la un pas al construcţiei:
  - recursiv: există mai multe posibilităţi care se iau în considerare, se alege una, dacă analiza nu se termină cu succes, atunci se revine şi se face o altă alegere;
  - liniar (sau predictiv): la un moment dat o singură alegere este posibilă.

Clasificarea celor mai cunoscute tipuri de analizoare sintactice este prezentată în tabelul următor:

	descendent	ascendent
recursiv	descendent cu reveniri	ascendent cu reveniri
liniar	LL(k)	de precedenţă LR(k)

Analizorul sintactic descendent cu reveniri este foarte simplu din punct de vedere al algoritmului şi singura restricţie impusă gramaticii este să nu fie recursivă la stânga (eliminarea recursivităţii la stânga se poate realiza aplicând un algoritm simplu). Din păcate, datorită recursivităţii, eficienţa lui este scăzută şi nu se aplică pentru gramatici mari. Analizorul sintactic ascendent cu reveniri are aceleaşi principii, direcţia de construire a arborelui diferă, dar este mult mai puţin cunoscut.

Algoritmii liniari sunt cei mai performanţi şi cei mai des folosiţi în implementarea compilatoarelor pentru limbaje de programare:

- analizor sintactic LL(k) este o abreviere pentru *Left-Left (Stânga-Stânga)* şi semnifică: secvenţa de intrare este citită de la *stânga* spre dreapta şi arborele se construieşte folosind derivări de *stânga*;

- analizor sintactic bazat pe relaţii de precedenţă - este o clasă de algoritmi care lucrează pe gramatici de precedenţă, dar restricţiile acestor gramatici fac ineficientă aplicarea lor pentru limbaje de programare; sunt algoritmi foarte eficienţi pentru verificarea corectitudinii sintactice a expresiilor aritmetice;

- analizor sintactic LR(k) este o abreviere pentru *Left-Right (Stânga-Dreapta)* şi semnifică: secvenţa de intrare este citită de la *stânga* spre dreapta şi arborele se construieşte folosind derivări de *dreapta*; este o clasă de algoritmi având aceeaşi strategie, dar care diferă prin unele detalii sau optimizări, incluzând: LR(0), SLR, LR(1), LALR.

Arborele de analiză sintactică nu este o structură uşor de manipulat. De aceea, se folosesc următoarele două alternative de rezultate, pe baza cărora arborele poate fi complet construit:

1. şirul derivărilor directe succesive care formează derivarea $S \overset{*}{\Rightarrow} w$:
   $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow w$;
2. în condiţiile în care producţiile gramaticii sunt numerotate, şirul numerelor de producţii folosite la fiecare pas de derivare în $S \overset{*}{\Rightarrow} w$.

3.1. Analizor sintactic descendent cu reveniri

Să încercăm să simulăm analiza sintactică descendentă pentru a înţelege care sunt situaţiile şi acţiunile care apar în construcţia arborelui de analiză sintactică prin această metodă. Vom considera gramatica:
$G = (\{S\}, \{a, b, c\}, \{S \rightarrow aSbS|aS|c\}, S)$
şi secvenţa: $w = aacbc$.
Figura 3.1.a)-k) conţine aceste situaţii, iar pentru fiecare arbore, nodul curent este cel încercuit. Fiind un analizor sintactic descendent vom începe construirea arborelui de la rădăcină (figura 3.1.a)). Având un simbol neterminal ca nod curent în arbore, trebuie construiţi descendenţii lui, conform unei reguli de producţie a gramaticii. Se observă că neterminalul S va apărea de mai multe ori în arbore. Pentru a putea scrie un algoritm, avem nevoie de un "şablon" după care aplicăm producţiile pentru un neterminal, astfel că vom considera o ordine între producţiile pentru acelaşi neterminal şi vom aplica întotdeauna producţiile în această ordine.

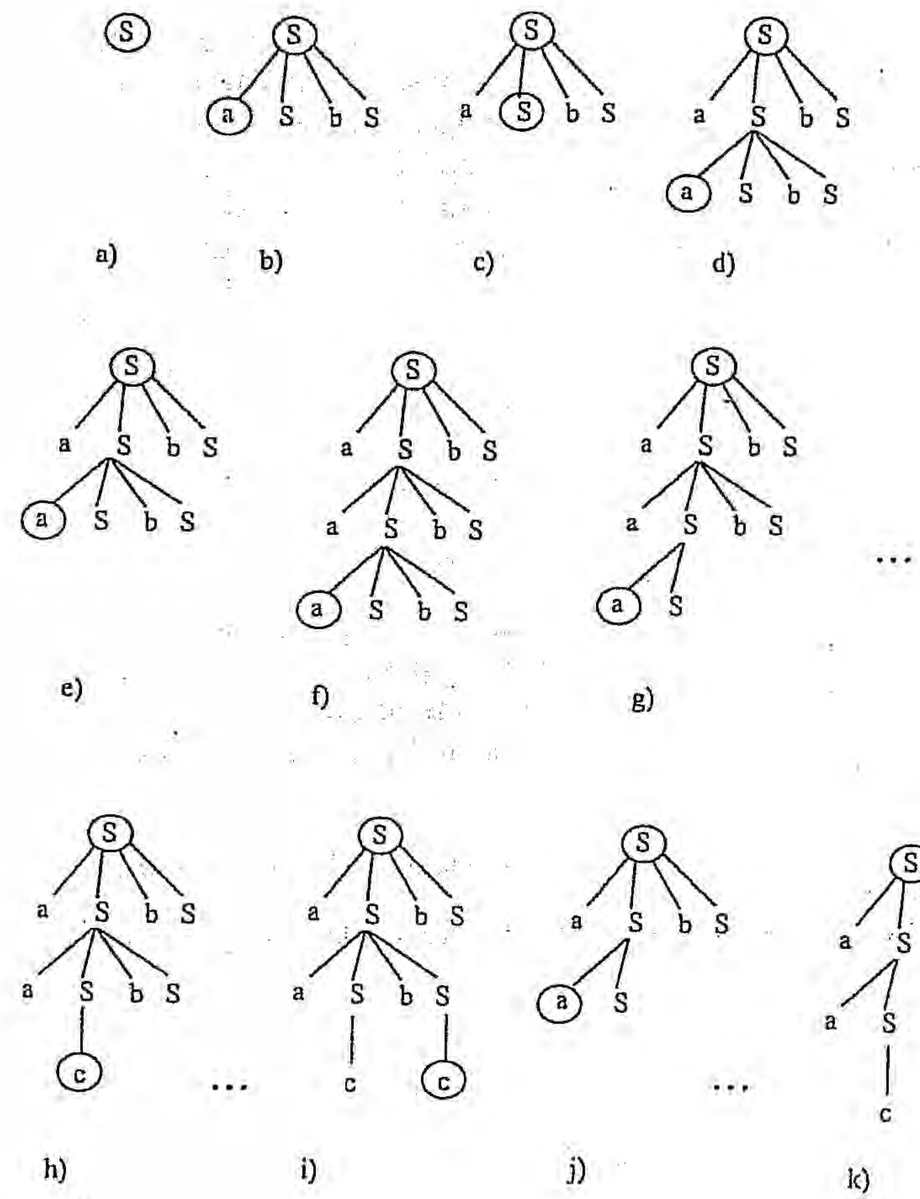

Figura 3.1: Analiza sintactică descendentă a secvenţei *aacbc*

Pentru exemplul nostru, fie prima producţie $S \rightarrow aSbS$. În momentul în care am construit descendenţii nodului neterminal, nodul curent se schimbă, devenind cel mai la stânga descendent al lui, fapt ilustrat în figura 3.1.b). Nodul curent este a, un simbol terminal. Pentru acest caz, vom verifica dacă el poate face parte din frontul arborelui, adică dacă se potriveşte cu simbolul curent din secvenţa de intrare, care în acest moment este primul simbol. Este vorba de acelaşi terminal, ceea ce înseamnă că putem continua cu construcţia arborelui: simbolul curent din secvenţă este al doilea simbol (tot a) şi nodul curent devine vecinul la dreapta, adică S, după cum se observă în figura 3.1.c).

Avem din nou simbol neterminal, construim subarborele corespunzător, nodul curent devine cel mai la stânga descendent, ca în figura 3.1.d). Nodul curent este terminalul a, care se potriveşte cu al doilea simbol, deci putem avansa în construcţie: simbolul curent din secvenţă este c, nodul curent este S (figura 3.1.e)). Aplicăm din nou expandarea nodului S, nodul curent devine a, situaţie ilustrată în figura 3.1.f). În acest caz avem o nepotrivire între nodul curent care este a şi simbolul curent din secvenţă care este c, ceea ce blochează temporar construcţia arborelui. Aici intervine situaţia de revenire: trebuie să ne întoarcem la pasul precedent şi să verificăm dacă putem face altă încercare de a rescrie S, adică dacă există altă producţie pentru S. Răspunsul este afirmativ şi construim subarborele corespunzător lui S pe baza acestei producţii, ca în figura 3.1.g). Din nou avem o nepotrivire, vom reveni şi vom încerca producţia $S \rightarrow c$ în construcţia subarborelui pentru S (figura 3.1.h).

Paşii deja descrişi se aplică cu succes până la arborele de derivare din figura 3.1.i). În această situaţie frontul va conţine mai multe simboluri decât secvenţa de analizat, deci, avem din nou o situaţie de insucces de moment. Pentru ultimele noduri neterminale considerate am epuizat posibilităţile de rescriere, ceea ce impune ca revenirea să se facă cu un nivel mai sus în arbore, la primul nod care conţine S pe nivelul 1, unde vom aplica altă încercare de construire a subarborelui, ca în figura 3.1.j).

Construcţia restului arborelui nu mai ridică alte probleme în afara situaţiilor explicate şi ilustrate în figură, astfel că în final vom obţine arborele de analiză sintactică din figura 3.1.k), care indică faptul că secvenţa este acceptată de gramatică, cu alte cuvinte analiza sintactică se termină cu succes.

3.1.1. Modelul analizorului sintactic descendent cu reveniri

Automatul push-down corespunzător analizorului sintactic descendent cu reveniri poate fi caracterizat de următoarea configuraţie[Şer87]:

$$(s, i, \alpha, \beta)$$

unde:

- s reprezintă starea automatului şi are una din posibilele valori:
  - q - stare normală;
  - r - stare de revenire;
  - t - stare de terminare;
  - e - stare de eroare;
- i reprezintă poziţia simbolului curent din secvenţa de intrare;
- α este conţinutul stivei de lucru, care memorează modul în care se obţine arborele;
- β este conţinutul stivei de intrare, reprezentând partea din arbore care nu a fost încă dezvoltată.

Configuraţia iniţială este $(q, 1, \epsilon, S)$, caracterizând automatul prin: stare normală, primul simbol din secvenţă, stiva de lucru vidă şi stiva de intrare conţine simbolul de start. Configuraţia finală este $(t, n+1, \alpha, \epsilon)$, adică automatul este în starea t, au fost detectate toate simbolurile din secvenţă, inclusiv al n-lea, deci indicele este $n+1$ şi arborele este complet construit, $\beta = \epsilon$.

Pe mulţimea configuraţiilor se definesc următoarele tranziţii (notate $|-$):

1. **Expandare**: are loc atunci când vârful stivei de intrare este un neterminal şi constă în rescrierea (expandarea) neterminalului respectiv:

   $(q, i, \alpha, A\beta) | - (q, i, \alpha A_1, \gamma_1\beta)$

   unde $A \rightarrow \gamma_1|\gamma_2|...$ sunt producţiile neterminalului A, iar indicele 1 ataşat în momentul exapandării indică faptul că a fost luată în considerare prima producţie pentru A (în ordinea prestabilită). Acest indice este util pentru a şti la un moment dat, pentru un anumit neterminal, care producţie a fost folosită şi care se mai poate folosi.

2. **Avans**: se realizează atunci când vârful stivei de intrare coincide cu simbolul curent (al i-lea) din secvenţa de intrare w:

   $(q, i, \alpha, a_i\beta) \vdash (q, i+1, \alpha a_i, \beta)$

3. **Insucces de moment**: se constată atunci când simbolul terminal din vârful stivei de intrare şi simbolul curent din secvenţa de intrare diferă. Se marchează astfel trecerea într-o stare de revenire:

   $(q, i, \alpha, a\beta) \vdash (r, i, \alpha, a\beta), a \neq a_i$

4. **Revenire**: reprezintă situaţia când e necesar să refacem partea dreaptă a unei producţii care nu a fost bine aleasă:

   $(r, i, \alpha a, \beta) \vdash (r, i-1, \alpha, a\beta)$

5. **Altă încercare**: reprezintă "ieşirea" dintr-o stare de revenire şi constă din mai multe situaţii posibile: încercarea unei alte producţii pentru un acelaşi neterminal (1), sau rămânerea în stare de revenire pentru a reface partea dreaptă a producţiei (2), sau situaţia de eroare, când analiza sintactică se termină cu răspuns negativ: secvenţa nu este acceptată (3). Această ultimă situaţie apare atunci când simbolul curent din secvenţa de intrare este primul simbol, neterminalul A este simbolul de start şi au fost încercate toate producţiile pentru el.

$$(r, i, \alpha A_j, \gamma_j\beta) \vdash \begin{cases} (q, i, \alpha A_{j+1}, \gamma_{j+1}\beta), \text{dacă } \exists A \rightarrow \gamma_{j+1} \\ (r, i, \alpha, A\beta) \text{în caz contrar, cu excepţia} \\ (e, i, \alpha, \beta), \text{dacă } i = 1, A = S, \textbf{EROARE} \end{cases}$$

6. **Succes**: reprezintă terminarea cu succes, acceptarea secvenţei de către gramatică:

   $(q, n+1, \alpha, \epsilon) \vdash (t, n+1, \alpha, \epsilon)$

Algoritmul 3.1 descrie analiza sintactică descendentă cu reveniri pe baza modelului prezentat anterior. Cât timp analiza nu s-a terminat cu succes sau nu s-a ajuns într-o situaţie de eroare, configuraţia curentă determină acţiunea următoare, în primul rând în funcţie de stare - stare normală (liniile 5 - 23) sau stare de revenire (liniile 24-44).

Dacă starea este normală atunci vârful stibei β este luat în considerare: pentru un simbol neterminal în vârful stivei vom aplica expandare (liniile 9-12), iar pentru simbol terminal, se verifică dacă corespunde cu simbolul curent din secvenţa de intrare (linia 14) şi atunci se aplică avans (liniile 15-17), altfel se aplică insucces de moment (linia 19).

Dacă starea este de revenire atunci acţiunea este decisă de vârful stivei α - pentru un simbol terminal în vârful stivei se aplică revenire (liniile 26-28), iar pentru un simbol neterminal se aplică altă încercare (liniile 30-40). Când nici o altă încercare nu se mai poate aplica, analiza va intra într-o stare de eroare (liniile 35-36). Dacă s-a parcurs toată secvenţa de intrare şi starea este normală atunci analiza va ajunge în stare de terminare (liniile 6-7).

Am precizat că în cazul în care programul supus analizei este corect din punct de vedere sintactic, este necesar să reţinem şi modul în care este construit, şi anume arborele de analiză sintactică sau o modalitate de reprezentare a lui. În cazul acceptării secvenţei, conţinutul stivei α din configuraţia finală poate fi prelucrată pentru a obţine şirul de producţii folosit. Algoritmul 3.2 procesează stiva α şi pentru fiecare neterminal va reţine indicele (numărul producţiei) pentru acel neterminal.

Exemplul 3.1. Considerând din nou gramatica de la începutul paragrafului, $G = (\{S\}, \{a, b, c\}, \{S \rightarrow aSbS, S \rightarrow aS, S \rightarrow c\}, S)$ aplicăm algoritmul de mai sus şi vom obţine următoarea secvenţă de tranziţii în analiza sintactică descendentă cu reveniri a şirului de intrare $aacbc$.

Pentru fiecare tranziţie vom specifica între paranteze rotunde ce acţiune se aplică (de exemplu: (1) semnifică faptul că la acest pas de analiză se aplică o acţiune de expandare).

$(q, 1, S) \vdash_{(1)} (q, 1, S_1, aSbS) \vdash_{(2)} (q, 2, S_1a, SbS) \vdash_{(1)} (q, 2, S_1aS_1, aSbSbS)$
$\vdash_{(2)} (q, 3, S_1aS_1a, SbSbS) \vdash_{(1)} \underline{(q, 3, S_1aS_1aS_1, aSbSbSbS)} \vdash_{(3)} (b, 3,$
$S_1aS_1aS_1, aSbSbSbS) \vdash_{(5)} \underline{(q, 3, S_1aS_1aS_2, aSbSbS)} \vdash_{(3)} (b, 3, S_1aS_1aS_2,$
$aSbSbS) \vdash_{(5)} (q, 3, S_1aS_1aS_3, cbSbS) \vdash_{(2)} (q, 4, S_1aS_1aS_3c, bSbS) \vdash_{(2)}$
$(q, 5, S_1aS_1aS_3cb, SbS) \vdash_{(1)} (q, 5, S_1aS_1aS_3cbS_1, aSbSbS) \vdash_{(3)} (b, 5,$
$S_1aS_1aS_3cbS_1, aSbSbS) \vdash_{(5)} (q, 5, S_1aS_1aS_3cbS_2, aSbS) \vdash_{(3)} (b, 5,$
$S_1aS_1aS_3cbS_2, aSbSbS) \vdash_{(5)} (q, 5, S_1aS_1aS_3cbS_3, cbS) \vdash_{(2)} (q, 6,$
$S_1aS_1aS_3cbS_3c, bS) \vdash_{(3)} \underline{(b, 6, S_1aS_1aS_3cbS_3c, bS)} \vdash_{(4)} (b, 5,$
$S_1aS_1aS_3cbS_3, cbS) \vdash_{(3)} (b, 5, S_1aS_1aS_3cbS_3, cbS) \vdash_{(5)} (b, 5, S_1aS_1aS_3cb,$
$SbS) \vdash_{(4)} (b, 4, S_1aS_1aS_3c, bSbS) \vdash_{(4)} (b, 3, S_1aS_1aS_3, cbSbS) \vdash_{(5)} (b, 3,$
$S_1aS_1a, SbSbS) \vdash_{(4)} (b, 2, S_1aS_1, aSbSbS) \vdash_{(5)} (q, 2, S_1aS_2, aSbS) \vdash_{(2)}$
$(q, 2, S_1aS_2, aSbS) \vdash_{(2)} (q, 3, S_1aS_2a, SbS) \vdash_{(1)} (q, 3, S_1aS_2aS_1, aSbSbS)$
$\vdash_{(3)} (b, 3, S_1aS_2aS_1, aSbSbS) \vdash_{(5)} (q, 3, S_1aS_2aS_2, aSbS) \vdash_{(3)} (b, 3,$
$S_1aS_2aS_2, aSbS) \vdash_{(5)} (q, 3, S_1aS_2aS_3, cbS) \vdash_{(2)} (q, 4, S_1aS_2aS_3c, bS) \vdash_{(2)}$
$(q, 5, S_1aS_2aS_3cb, S) \vdash_{(1)} (q, 5, S_1aS_2aS_3cbS_1, aSbS) \vdash_{(3)} (b, 5, S_1aS_2aS_3c$
$bS_1, aSbS) \vdash_{(5)} (q, 5, S_1aS_2aS_3cbS_2, aS) \vdash_{(3)} (b, 5, S_1aS_2aS_3cbS_2, aS)$
$\vdash_{(5)} (q, 5, S_1aS_2aS_3cbS_3, c) \vdash_{(2)} (q, 6, S_1aS_2aS_3cbS_3c, \epsilon) \vdash_{(6)}$
$(t, 5, S_1aS_2aS_3cbS_3c, \epsilon)$

Algoritmul 3.1 Descendent_reveniri

```
INPUT: G, w = x1x2...xn;
OUTPUT: mesaj (acceptare sau nu), sir_prod
s := q; i := 1; α := ε; β := S : { configuraţia iniţială}
while (s ≠ t) and (s ≠ e) do
  if s = q then
    if (β = ε) and (i = n+1) then
      s:=t
    else
      if varf(β) = A then
        push(α, A1); {fie A → γ prima producţie a lui A }
        pop(β,A); push(β, γ);
      else
        if varf(β) = xi then
          i := i+1; push(α,a); pop(β,a)
        else
          s:=r
  else
    if s=r then
      if varf(α) = a then
        i := i-1; pop(α,a); push(β,a)
      else
        if ∃A → γj+1, dacă A → γj a fost ultima folosită then
          s:= q;
          pop(α, Aj); push(α, Aj+1);
          pop(β, γj); push(β, γj+1);
        else
          if (i=1) and (A=S) then
            s:= e
          else
            pop(α, Aj); push(β,A)
if s=e then
  mesaj: "EROARE"
else
  mesaj: "Secvenţa este acceptată"
  Construire_sir_prod
```


Algoritmul 3.2 Construire_sir_prod

```
sir_prod := "";
while not vida(α) do
  if varf(α) ∈ N then
    sir_prod := indice(varf(α)) + sir_prod;
  end if
  pop(α);
end while
```

Am subliniat situaţiile mai complicate care apar în alegerea tranziţiei care se aplică la un moment dat: insuccesul de moment urmat de o altă încercare, respectiv insuccesul de moment urmat de revenire. În acest caz, configuraţia obţinută după insuccesul de moment, datorită faptului că vârful stivei de lucru este un terminal, nu permite o altă încercare. Astfel că se va aplica o revenire, reconstituind partea dreaptă a ultimei producţii aplicate, ceea ce specifică, de fapt, revenirea pe un nivel superior în arborele de analiză.

Pentru construirea şirului de producţii folosit vom prelucra conţinutul stivei de lucru din ultima configuraţie: $S_1aS_2aS_3cbS_3c$, obţinând $1, 2, 3, 3$.

3.2. Analizor sintactic LL(k)

Analizorul sintactic LL(k) [ASU86] se caracterizează prin faptul că *în orice retranscriere alegerea producţiei este unic determinată* de:

- partea închisă ($w' = a_1 \dots a_i$);
- simbolul de retranscris (A);
- predicţia de lungime k: $a_{i+1} \dots a_{i+k}$,

după cum se observă şi din alegerea producţiei $A \to \beta$ în figura 3.2.

Predicţia de lungime k reprezintă următoarele k simboluri care ar trebui generate din configuraţia curentă. Pentru aceasta se introduce o funcţie $FIRST_k$ [ASU86], care calculează primele k simboluri ce se pot obţine prin derivări succesive dintr-o anumită formă propoziţională:

$FIRST_k : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma^k)$

$FIRST_k(\alpha) = \{u | u \in \Sigma^k, \alpha \overset{*}{\Rightarrow} ux, |u| = k$ sau $\alpha \overset{*}{\Rightarrow} u, |u| \leq k\}$

(primele k simboluri ale lui α)

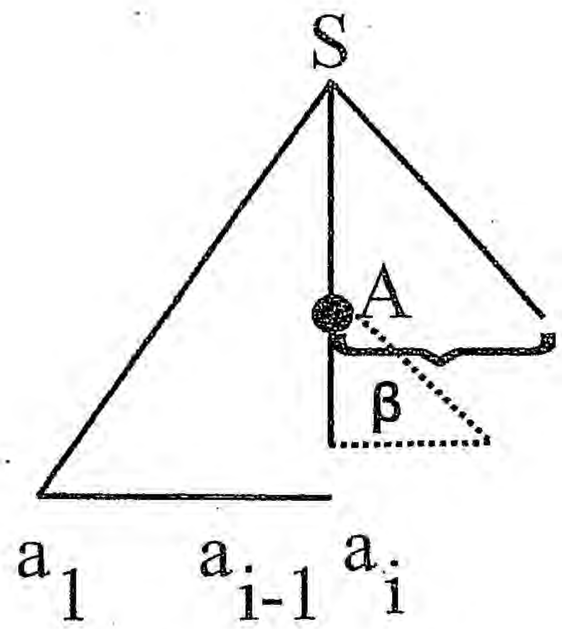

Figura 3.2: Construcţia arborelui prin analiza sintactică LL(1)

3.2.1. Gramatici de tip LL(k)

Definiţia 3.1. *[AU73] O gramatică $G = (N, \Sigma, P, S)$ este de tip LL(k) dacă pentru oricare două derivări de stânga:*

1. $S \overset{*}{\Rightarrow} wA\alpha \Rightarrow_{st} w\beta\alpha \overset{*}{\Rightarrow} wx$;
2. $S \overset{*}{\Rightarrow} wA\alpha \Rightarrow_{st} w\gamma\alpha \overset{*}{\Rightarrow} wy$;

astfel încât $FIRST_k(x) = FIRST_k(y)$ avem că: $\beta = \gamma$.

Definiţia poate fi reformulată astfel: pentru orice formă propoziţională $wA\alpha$, primele k simboluri derivabile din $A\alpha$ definesc în mod unic o producţie care se poate aplica lui A pentru a obţine o derivare a unui cuvânt (= secvenţă de simboluri terminale) care începe cu w şi se continuă cu aceste k simboluri. Această condiţie este uneori dificil de verificat şi în majoritatea cazurilor se face apel la următoarea teoremă.

Teorema 3.1. *[AU73] Condiţia necesară şi suficientă pentru ca o gramatică să fie de tip LL(k) este ca pentru orice pereche de producţii distincte ale aceluiaşi neterminal ($A \to \beta, A \to \gamma, \beta \neq \gamma$) să fie verificată condiţia:*

$FIRST_k(\beta\alpha) \cap FIRST_k(\gamma\alpha) = \emptyset, \forall\alpha$ *astfel încât* $S \overset{*}{\Rightarrow} uA\alpha$.

3.2.2. Analizorul sintactic LL(1)

Considerând cazul în care predicţia este de lungime unu ($k = 1$) se obţine un algoritm eficient de analiză sintactică şi anume analizorul sintactic LL(1), care se caracterizează prin faptul că la fiecare pas:

- simbolul A - neterminalul situat cel mai la stânga în forma propoziţională;
- predicţia a_i - următorul simbol de pe banda de intrare;

determină în mod unic alegerea unei producţii $A \to \alpha$.

Teorema 3.2. *[Şer87] O gramatică este de tip LL(1) dacă şi numai dacă pentru fiecare neterminal A cu producţiile $A \to \alpha_1|\alpha_2|\ldots|\alpha_n$, $FIRST_k(\alpha_i) \cap FIRST_k(\alpha_j) = \emptyset$ şi dacă $\alpha_i \overset{*}{\Rightarrow} \epsilon$, $FIRST(\alpha_i) \cap FOLLOW(A) = \emptyset, \forall i, j = \overline{1,n}, i \neq j$.*

După cum sugerează teorema, o situaţie mai specială apare atunci când primul simbol din rescrierea lui A, $FIRST(A)$, este secvenţa vidă. Analizând arborele deja construit, se poate observa că în acest caz trebuie să luăm în considerare simbolul care se obţine din ceea ce "urmează" după A. Pentru aceasta se introduce o nouă funcţie [AU73]:

$FOLLOW : (N \cup \Sigma)^* \to \mathcal{P}(\Sigma)$

$FOLLOW(\beta) = \{w \in \Sigma | S \overset{*}{\Rightarrow} \alpha\beta\gamma, w \in FIRST(\gamma)\}$.

Pentru a construi un analizor sintactic LL(1) avem nevoie de predicţiile care pot apărea pe parcursul analizei şi care se memorează într-un tabel, numit *tabel de analiză LL(1)*.

3.2.3. Construirea tabelului de analiză LL(1)

Calculul elementelor din tabel depinde de valorile funcţiei $FIRST$. Pentru a putea descrie o metodă de calcul a lui $FIRST$ avem nevoie de următoarea proprietate:

Observaţii [GJ90]:

- Dacă L_1, L_2 sunt două limbaje peste alfabetul Σ, atunci: $L_1 \oplus L_2 = \{w | x \in L_1, y \in L_2, xy = w, |w| \leq 1$ sau $xy = wz, |w| = 1\}$ şi
- $FIRST(\alpha\beta) = FIRST(\alpha) \oplus FIRST(\beta)$

  $FIRST(X_1 \ldots X_n) = FIRST(X_1) \oplus \ldots \oplus FIRST(X_n)$

Algoritmul 3.3 FIRST

INPUT: G
OUTPUT: $FIRST(X), \forall X \in N \cup \Sigma$
for $\forall a \in \Sigma$ do
  $F_i(a) = \{a\}, \forall i \geq 0$
end for
i := 0;
$F_0(A) = \{x | x \in \Sigma, A \rightarrow x\alpha$ sau $A \rightarrow x \in P\}$; {*iniţializare*}
repeat
  i := i+1;
  for $\forall X \in N$ do
    if F_{i-1} au fost calculate $\forall X \in N \cup \Sigma$ then
      {*dacă* $\exists Y_j, F_{i-1}(Y_j) = \emptyset$ *atunci nu se poate aplica*}
      $F_i(A) = F_{i-1}(A) \cup$
      $\{x | A \rightarrow Y_1 \ldots Y_n \in P, x \in F_{i-1}(Y_1) \oplus \ldots \oplus F_{i-1}(Y_n)\}$
    end if
  end for
until $F_{i-1}(A) = F_i(A)$
$FIRST(X) := F_i(X), \forall X \in N \cup \Sigma$

Exemplul 3.2. Fie gramatica $G = (\{S, A, B, C, D\}, \{=, *, (,), a\}, P, S)$, unde P este:

$S \rightarrow BA$
$A \rightarrow +BA$
$A \rightarrow \epsilon$
$B \rightarrow DC$
$C \rightarrow *DC$
$C \rightarrow \epsilon$
$D \rightarrow (S)$
$D \rightarrow a$

Aplicând algoritmul obţinem:

$$\begin{aligned}
F_0(S) &= \emptyset, F_0(A) = \{a, \epsilon\}, F_0(B) = \emptyset, F_0(C) = \{*, \epsilon\}, F_0(D) = \{(, a\} \\
F_1(S) &= \emptyset, F_1(A) = \{a, \epsilon\}, F_1(B) = \{(, a\}, F_1(C) = \{*, \epsilon\}, \\
&\quad F_1(D) = \{(, a\} \\
&\quad (F_1(S) \text{nu s-a putut calcula deoarece } F_0(B) \text{ nu era calculat}) \\
F_2(S) &= \{(, a\}, F_2(A) = \{a, \epsilon\}, F_2(B) = \{(, a\}, F_2(C) = \{*, \epsilon\}, \\
&\quad F_2(D) = \{(, a\}
\end{aligned}$$

$$\begin{aligned}
F_3(S) &= \{(, a\}, F_3(A) = \{a, \epsilon\}, F_3(B) = \{(, a\}, F_3(C) = \{*, \epsilon\}, \\
&\quad F_3(D) = \{(, a\}
\end{aligned}$$

După cum se observă din teorema 3.2 în cazul în care $\epsilon \in FIRST(A)$, pentru un neterminal oarecare A, vom avea nevoie de mulţimea $FOLLOW(A)$. Nu este necesar să construim mulţimile $FOLLOW$ de la început, ci doar în timpul construirii tabelului dacă situaţia de mai sus apare. Algoritmul 3.4 descrie construcţia mulţimii $FOLLOW$.

Algoritmul 3.4 FOLLOW

INPUT: G, FIRST(X), $\forall X \in N \cup \Sigma$
OUTPUT: FOLLOW(X), $\forall X \in N \cup \Sigma$
$F(X) = \emptyset, \forall X \in N - \{S\}$; {*iniţializare*}
$F(S) = \{\epsilon\}$; {*corespunzător simbolului \$ folosit în analiză*}
repeat
  for $B \in N$ do
    for $A \rightarrow \alpha By \in P$ do
      if $\epsilon \in FIRST(y)$ then
        $F'(B) = F(B) \cup F(A)$
      else
        $F'(B) = F(B) \cup FIRST(y)$
      end if
    end for
  end for
until $F'(X) = F(X), \forall X \in N$
$FOLLOW(X) = F(X), \forall X \in N$.

Tabelul de analiză LL(1) trebuie să cuprindă acţiunile posibile luând în considerare simbolul curent $X \in N \cup \Sigma$ şi predicţiile posibile $u \in \Sigma$. Astfel vom avea un tabel bidimensional, care va conţine câte o linie pentru fiecare simbol din gramatică (terminale şi neterminale) şi câte o coloană pentru fiecare predicţie posibilă. În plus, se adaugă un caracter special, de obicei notat '\$' ($\notin N \cup \Sigma$), al cărui scop este să marcheze sfârşitul secvenţei şi căruia i se alocă o linie şi o coloană în tabel. Efectul acestui simbol în faza de analiză propriu-zisă este de a elimina verificările de stivă goală. Regulile de completare a tabelului sunt:

1. $M(A, a) = (\alpha, i), \forall a \in FIRST(\alpha), a \neq \epsilon, A \rightarrow \alpha$ producţie în P cu numărul i;
   $M(A, b) = (\alpha, i)$, dacă $\epsilon \in FIRST(\alpha), \forall b \in FOLLOW(A), A \rightarrow \alpha$ producţie în P cu numărul i;

2. $M(a,a) = pop, \forall a \in \Sigma$;[2]

3. $M(\$,\$) = acc$;

4. M(x,a)=err (eroare) în celelalte cazuri.

Pentru gramatica din exemplul precedent, construcţia tabelului de analiză LL(1) necesită şi calculul mulţimilor *FOLLOW* pentru neterminalele A şi C, deoarece $\epsilon \in FIRST(A)$ şi $\epsilon \in FIRST(C)$. Aplicarea algoritmului FOLLOW se poate observa în tabelul următor:

	F	FOLLOW
S	$\{), \epsilon\}$	$\{), \epsilon\}$
A	F(S)	$\{), \epsilon\}$
B	$FIRST(A) \cup F(S) \cup F(A)$	$\{+,),\epsilon\}$
C	$F(B) \cup F(C)$	$\{+,),\epsilon\}$
D	$FIRST(C) \cup F(B) \cup F(C)$	$\{+,*,),\epsilon\}$

Tabelul de analiză LL(1) pentru gramatica din exemplul 3.2 este:

	+	*	(	)	a	$
S			BA,1		BA,1	
A	$+BA, 2$			ϵ,3		ϵ, 3
B			DC,4		DC,4	
C	ϵ, 6	*DC,5		ϵ, 6		ϵ,6
D			(S),7		a,8	
+	pop					
*		pop				
(			pop			
)				pop		
a					pop	
$						acc

3.2.4. Funcţionarea analizorului sintactic LL(1)

Putem descrie acum modul în care va funcţiona analiza sintactică LL(1) după modelul unui automat push-down, având ca date de intrare: gramatica limbajului, tabelul de analiză LL(1) şi secvenţa de analizat. Automatul push-down are o bandă de intrare, numită *stivă de intrare*, o *stivă de lucru*, unitatea centrală al cărei comportament este definit prin tranziţii şi o bandă de ieşire [AU73, Şer87].

[2]pop reprezină operaţia de scoatere din stivă, în acest caz se va aplica pentru două stive; detalii în paragraful următor.

Configuraţia: se defineşte ca un triplet (α, β, π), cu următoarele semnificaţii:
α - stiva de intrare;
β - stiva de lucru;
π - banda de ieşire.

Configuraţia iniţială: $(w\$, S\$, \epsilon)$ semnifică faptul că: banda de intrare conţine toată secvenţa de analizat, terminată cu simbolul \$, stiva de lucru conţine marcatorul de sfârşit de stivă \$ şi simbolul S (simbolul de start al gramaticii), iar banda de ieşire este vidă.

Configuraţia finală: $(\$, \$, \pi)$ specifică faptul că atât stiva de intrare, cât şi stiva de lucru sunt vide, iar banda de ieşire va conţine şirul de producţii folosit. Se observă că acceptarea unei secvenţe se face pe baza criteriului stivei vide.

Tranziţiile se definesc în felul următor:

1. **push** - operaţia de punere în stivă:

   $(ux, A\alpha\$, \pi) \vdash (ux, \beta\alpha\$, \pi i)$, dacă $M(A,u) = (\beta, i)$;

   de fapt, în stiva de lucru se efectuează următoarele operaţii: se scoate A din stivă şi se pune β în stivă;

2. **pop** - operaţia de scoatere din stivă, se elimină vârfurile ambelor stive (dacă ele coincid):

   $(ux, a\alpha\$, \pi) \vdash (x, \alpha\$, \pi)$, dacă M(a,u)=pop ;

3. tranziţia de acceptare, dacă s-a obţinut configuraţia finală, notată acc:

   $(\$, \$, \pi) \vdash acc$;

4. în celelalte cazuri eroare, notată err:

   $(n\alpha, x\beta\$, \pi) \vdash err$.

Corespunzător tranziţiilor de mai sus, analiza sintactică LL(1) se face conform algoritmului 3.5.

Pentru gramatica considerată în exemplul 3.2, considerând producţiile numerotate în ordinea în care sunt date în exemplu, pentru care avem construit tabelul de analiză LL(1), analiza secvenţei $a * (a + a)$ se face în felul următor:

$(a * (a + a)\$, S\$, \epsilon) \vdash (a * (a + a)\$, BA\$, 1) \vdash (a * (a + a)\$, DCA\$, 14) \vdash (a * (a + a)\$, aCA\$, 148) \vdash (*(a + a)\$, CA\$, 148) \vdash (*(a + a)\$, *DCA\$, 1485) \vdash ((a + a)\$, DCA\$, 1485) \vdash ((a + a)\$, (S)CA\$, 14857) \vdash ((a + a)\$,$

Algoritmul 3.5 Anal_sintLL1

INPUT:

- tabelul de analiză LL(1) fără conflicte;
- producţiile gramaticii numerotate;
- secvenţa de analizat $w = a_1 a_2 \ldots a_n$.

OUTPUT: este sau nu secvenţa acceptată; dacă da, şirul de producţii folosit.

```
alfa := w$; beta := S$; pi := ε;
go := true;
while go do
  if M(varf(beta),varf(alfa))=(b,i) then
    pop(beta);
    push(beta,b);
    push(pi,i)
  else
    if M(varf(beta),varf(alfa))=pop then
      pop(beta);
      pop(alfa);
    else
      if M(varf(beta),varf(alfa))=acc then
        go:=false; s:="acc";
      else
        go:=false; s:="err";
      end if
    end if
  end if
end while
if s="'acc"' then
  write("Secventa e acceptată");
  print(pi)
else
  write("Secventa nu e acceptată");
end if
```


$S)CA\$, 14857) \vdash (a+a)\$, BA)CA\$, 148571) \vdash (a+a)\$, DCA)CA\$, 145714) \vdash (a+a\$, aCA)CA\$, 14857148) \vdash (\epsilon+a)\$, A)CA\$, 14857148) \vdash (\epsilon+a)\$, \epsilon A)CA\$, 148571486) \vdash (+a)\$, A)CA\$, 148571486) \vdash (+a)\$, +BA)CA\$, 1485714862) \vdash (a)\$, BA)CA\$, 1485714862) \vdash (a)\$, DCA)CA\$, 14857148624) \vdash (a)\$, aCA)CA\$, 14857148628) \vdash ()\$, CA)CA\$, 14857148628) \vdash ()\$, A)CA\$, 148571486286) \vdash ()\$,)CA\$, 148571486283) \vdash (\$, CA\$, 1485714862836) \vdash (\$, \$, 14857148628363) \vdash$ acc.

Deci, secvenţa este corectă din punct de vedere sintactic, iar arborele de analiză sintactică se poate construi pornind de la simbolul de start ca rădăcină şi aplicând producţiile 1,4,8,5,7,1,4,8,6, 2,8,3,6,3, cu condiţia ca în orice moment producţia se aplică pentru cel mai din stânga neterminal.

Exemplul 3.3. Vom considera o gramatică (restrânsă) corespunzătoare instrucţiunii *if* din limbajul Pascal:

$G = (\{S, C, E, R, I\}, \{if, then, else, id, const, :=, =, <>\}, P, S)$, unde producţiile sunt:

$S \rightarrow if\ C\ then\ I \mid if\ C\ then\ I\ else\ I$

$C \rightarrow ERE$

$E \rightarrow id \mid const$

$R \rightarrow = \mid <> \mid ...$

$I \rightarrow id := E \mid ...$

Se observă că dacă construim mulţimile *FIRST* pentru cele două părţi dreapta de producţie corespunzătoare lui S, intersecţia este nevidă, deci această gramatică nu este de tip LL(1) şi nu se poate aplica analiza sintactică LL(1). Presupunând că această condiţie nu se verifică la început şi se trece la etapa următoare de construcţie a tabelului, la completarea celulei M(S,if) vor apărea două valori: (if ..., 1) şi (if ... else ..., 2).

Se poate face următoarea observaţie importantă pentru implementare:

Observaţie: La proiectarea unui analizor sintactic LL(1), dacă nu se verifică condiţia relativă la gramatică atunci construcţia tabelului se va face verificând să nu existe două valori distincte pentru o aceeaşi poziţie. Într-un astfel de caz, algoritmul nu va putea funcţiona pentru că s-a distrus proprietatea de unicitate a alegerii producţiei şi trebuie emis un mesaj corespunzător de eroare.

Producţiile gramaticii se pot modifica prin următorul artificiu: se factorizează prefixul comun din cele două producţii şi se introduce un nou neterminal pentru restul părţii drepte. Producţiile noului neterminal descriu partea diferită din producţiile iniţiale. Gramatica din exemplul 3.3 devine:

$G = (\{S, C, E, R, I, T\}, \{if, then, else, id, const, :=, =, <>\}, P, S)$, unde producţiile sunt:

(1) $S \rightarrow if\ C\ then\ I\ T$
(2) $T \rightarrow \epsilon$
(3) $T \rightarrow else\ I$
(4) $C \rightarrow ERE$
(5) $E \rightarrow id$
(6) $E \rightarrow const$
(7) $R \rightarrow =$
(8) $R \rightarrow <>$
(9) $I \rightarrow id := E$

Calculul mulţimii $FIRST$ are ca rezultat:

	F_0	F_1	F_2
S	if	if	if
T	ϵ, else	ϵ, else	ϵ, else
C	$\emptyset$	id, const	id, const
E	id, const	id, const	id, const
R	=, <>	=, <>	=, <>
I	id	id	id

Se observă că $\epsilon \in FIRST(T)$, deci trebuie să calculăm mulţimea $FOLLOW$ pentru neterminalul T. Ţinând cont de faptul că algoritmul FOLLOW se bazează pe studierea părţilor drepte ale producţiilor, singura producţie în care apare T în partea dreaptă este prima producţie, pe ultima poziţie, deci se poate concluziona că $FOLLOW(T) = \{\epsilon\}$.

Tabelul de analiză LL(1) va fi:

	if	then	else	id	const	:=	=	<>	$
S	ifCthenIT,1								
T			else,3						ϵ,2
C				ERE,4	ERE,4				
E				id,5	const,6				
R							=,7	<>,8	
I				id=E,9					
if	pop								
then		pop							
else			pop						
id				pop					
const					pop				
:=						pop			
=							pop		
<>								pop	
$									acc

Dorim acum să analizăm secvenţa *if x=z then x:=1*. S-a efectuat analiza lexicală a acestei secvenţe şi deci intrarea pentru analizorul sintactic LL(1) este: *if id=id then id:=const*. Tranziţiile corespunzătoare sunt:

(if id=id then id:=const$, S, ϵ) ⊢(if id=id then id:=const$,if C then IT$, 1) ⊢
(id=id then id:=const$,C then IT$,1)⊢(id=id then id:=const$,ERE then IT$,14) ⊢(id=id then id:=const$,id RE then IT$,145)⊢(=id then id:= const$,RE then IT$,145)⊢(=id then id:=const $, =E then I T $,1457)⊢ (id then id:=const$,E then IT$,1457)⊢(id then id:=const$,E then IT$, 1457)⊢(id then id:=const$,id then IT$,14575)⊢(then id:=const$,then IT$, 14575)⊢(id:=const$,IT$,14575)⊢(id:=const$,id:= ET$,145759)⊢ (:=const$,:= ET$,145759)⊢(const$,ET$,145759)⊢(const$,const T$, 1457596)⊢($,T$,1457596)⊢($,$\epsilon$$,14575962)⊢ ($,$,14575962)⊢ acc.

Deci secvenţa este acceptată.

3.2.5. Calculul funcţiilor FIRST şi FOLLOW prin matrice booleene

În [Boi83] se propune o metodă alternativă pentru calculul funcţiilor *FIRST* şi *FOLLOW* care se bazează pe algoritmii relativ simpli de lucru cu matrice booleene. Vom încerca să prezentăm succint această metodă, fără a o argumenta (pentru detalii, [Boi83]).

Funcţia $FIRST : N \cup \Sigma \rightarrow P(\Sigma)$ este transformată într-o relaţie binară: $X \cdot FIRST\ a$ dacă şi numai dacă $a \in FIRST(X), \forall X \in N \cup \Sigma, \forall a \in \Sigma$.

Pentru a putea aplica această metodă vom face o scurtă introducere a matricelor booleene, ca metodă eficientă de reprezentare a relaţiilor binare şi a operaţiilor definite asupra lor.

Fie ρ o relaţie definită pe $A \times B$. Atunci M_ρ notează matricea asociată relaţiei ρ, definită astfel: $\forall i = 1, |A|, \forall j = 1, |B|$

$$M_\rho = \begin{cases} 1, \text{ dacă } a_i\ \rho\ b_j \\ 0, \text{ altfel} \end{cases}$$

Cardinalul mulţimiilor pe care sunt definite relaţiile determină dimensiunea matricelor.

Compunerea relaţiilor corespunde înmulţirii booleene a matricelor relaţiilor. Înmulţirea booleeană are ca rezultat tot o matrice booleană, astfel că singura modificare care va trebui să o facem înmulţirii uzuale a matricelor va fi: dacă $c_{i,j} \geq 1$ atunci $c_{i,j} = 1$.

Matricea corespunzătoare închiderii tranzitive ρ^+ a unei relaţii se calculează aplicând algoritmul lui Warshall [AU73] (algoritmul 3.6).

Algoritmul 3.6 Warshall

```
INPUT: M -matricea corespunzătoare relaţiei ρ
OUTPUT: R -matricea corespunzătoare relaţiei ρ+
R := M;
for k:=1,n do
  for i:=1,n do
    for j:=1,n do
      if (r[i,k] = 1) and (r[k,j] = 1) then
        r[i,j] := 1;
      end if
    end for
  end for
end for
```

Închiderea reflexiv-tranzitivă ρ^* a unei relaţii presupune că avem o închidere ρ^+ şi în plus $a\ \rho\ a, \forall a \in A$, deci pentru a obţine matricea corespunzătoare M_{ρ^*} vom aplica algoritmul lui Warshall şi vom seta elementele de pe diagonala principală la valoarea 1.

Pentru determinarea relaţiei $FIRST$ sunt necesare:

- construirea mulţimii N_ϵ;
- construirea relaţiilor auxiliare FNE şi FTE.

Mulţimea $N_\epsilon = \{A \in N | A \Rightarrow^* \epsilon\}$ se calculează pe baza algoritmului 3.7.

Algoritmul 3.7 Calcul N_ϵ

INPUT: $G = (N, \Sigma, P, S)$ gramatică independentă de context
OUTPUT: N_ϵ
$N_0 = \{A | A \rightarrow \epsilon\}$;
i := 1;
repeat
  $N_i := N_{i-1} \cup \{A | A \rightarrow \alpha \in P, \alpha \in N_{i-1}^*\}$
until $N_i = N_{i-1}$;
$N_\epsilon := N_i$.

Relaţia FNE (First Nonterminal without ϵ-derives) se defineşte în felul următor:

$FNE \subset N \times N : \forall A, B \in N$
A FNE B $\Leftrightarrow \exists A \rightarrow \alpha B \beta \in P, \alpha \in N_{epsilon}^*$.

Similar, relaţia FTE ((First Terminal without ϵ-derives) este:
$FNE \subset N \times \Sigma : \forall A \in N, \forall a \in \Sigma$
A FNE a $\Leftrightarrow \exists A \rightarrow \alpha a \beta \in P, \alpha \in N_{epsilon}^*$.
Atunci: $M_{FIRST} = M_{FNE^*} \circ M_{FTE}$.

În mod asemănător, vom găsi o relaţie de calcul pentru relaţia FOLLOW: $\forall A \in N, \forall a \in \Sigma \cup \{\epsilon\}, A\ FOLLOW\ a$ dacă şi numai dacă $a \in FOLLOW(A)$. Pentru aceasta, sunt necesare următoarele relaţii auxiliare:

- LNE (Last Nonterminal): $LNE \subset N \times N$
  $\forall A, B \in N, A\ LNE\ B \Leftrightarrow \exists A \rightarrow \alpha B \beta, \beta \in N_{epsilon}^*$;
- EQN (EQuality Nonterminal): $EQN \subset N \times N$
  $\forall A, B \in N, A\ EQN\ B \Leftrightarrow \exists C \rightarrow \alpha A \beta B \gamma, \beta \in N_{epsilon}^*$;
- EQT (EQuality Terminal): $EQT \subset N \times \Sigma$
  $\forall A \in N, \forall a \in \Sigma, A\ EQT\ a \Leftrightarrow \exists C \rightarrow \alpha A \beta a \gamma, \beta \in N_{epsilon}^*$;

Atunci: $M_{FOLLOW} = (M_{LNE^*})^T \circ ((M_{EQN} \circ M_{FIRST}) + M_{EQT})$.

3.3. Analizoare sintactice LR(k)

Analiza sintactică ascendentă construieşte arborele de analiză sintactică începând de la frunze şi avansând spre rădăcină. Strategia de construcţie se bazează pe reducerea secvenţei de intrare la simbolul de start al gramaticii. La fiecare pas, forma propoziţională intermediară este comparată cu partea dreaptă a unei producţii din gramatică. Dacă sunt identice atunci este înlocuită cu partea stângă a producţiei respective. Deci, un algoritm eficient, care la un pas să determine în mod unic acţiunea, trebuie să garanteze că nu vor exista mai multe potriviri între forma propoziţională intermediară şi producţiile gramaticii. Se observă că aplicând o astfel de strategie construcţia arborelui se bazează pe derivări de dreapta.

Analizoarele sintactice ascendente liniare se bazează pe aşa numita metodă ***deplasare-reducere***[3] [Şer87]. Acest model se regăseşte la analizoarele sintactice bazate pe relaţii de precedenţă şi LR. Metoda se bazează pe un automat push-down şi are următoarul principiu de funcţionare: deplasarea - simbolurile de pe banda de intrare sunt introduse în stiva de lucru; când se obţine în vârful stivei de lucru o formă propoziţională identică cu partea dreaptă a unei producţii se aplică o reducere - se înlocuieşte

[3] în engleză *shift-reduce*

în stivă această parte dreaptă (numită *manşă*) cu partea stângă corespunzătoare. Acceptarea secvenţei presupune ca stiva de intrare să fie vidă iar stiva de lucru să conţină doar simbolul de start al gramaticii. Caracteristicile de bază ale metodei deplasare-reducere se pot sintetiza în:

- configuraţia automatului este un triplet (α, β, π), unde: α este conţinutul stivei de lucru, β este conţinutul stivei de intrare şi π este banda de ieşire, care va conţine şirul producţiilor folosite; pentru o mai bună înţelegere a funcţionării analizorului, stiva de lucru α conţine simbolurile în ordine inversă ("cu faţa spre" stiva de lucru);
- tranziţiile posibile într-un astfel de analizor sunt în număr de patru: deplasare, reducere, acceptare şi eroare.

Putem spune că eficienţa unui algoritm deplasare-reducere se bazează pe găsirea cât mai rapidă a manşei.

3.3.1. Gramatici de tip LR(k)

Denumirea de LR(k) provine de la modul în care se desfăşoară analiza sintactică şi anume:

- secvenţa de intrare este parcursă de la stânga la dreapta (*left*);
- se aplică derivări de dreapta (*right*);
- următoarele k simboluri de pe banda de intrare sunt luate în considerare pentru luarea unei decizii unice (*predicţie de lungime k*).

Dacă considerăm automatul push-down corespunzător acestui analizor sintactic atunci putem spune că în orice moment acţiunea este unic determinată de: simbolul din vârful stivei, simbolul curent de pe banda de intrare şi de predicţia de ordin k.

Definiţia 3.2. *[Şer87] Dacă într-o gramatică $G = (N, \Sigma, P, S)$ avem $S \overset{*}{\Rightarrow} \alpha A w \Rightarrow \alpha\beta w$, unde $\alpha \in (N \cup \Sigma)^*, A \in N, w \in \Sigma^*$, atunci orice prefix al secvenţei $\alpha\beta$ se numeşte* **prefix viabil** *în G.*

Definiţia 3.3. *[Şer87]* **Elementul de analiză LR(k)** *se defineşte ca fiind $[A \to \alpha.\beta, u]$, unde $A \to \alpha\beta$ e o producţie a gramaticii, $u \in \Sigma^k$ şi descrie stadiul în care, având în vedere producţia $A \to \alpha\beta$, a fost detectat α (α e în vârful stivei) şi se aşteaptă să fie detectat β:*
$S \overset{*}{\Rightarrow} \gamma A w \Rightarrow_{dr} \gamma\alpha\beta w$
$u = FIRST_k(w)$ *valabil pentru prefixul viabil $\gamma\alpha$.*

Analizoarele sintactice LR(k) lucrează cu gramatica îmbogăţită: $G = (N \cup \{S'\}, \Sigma, P \cup \{S' \to S\}, S'), S' \notin N$, pentru a evita ca simbolul de start S să apară în partea dreaptă a unei producţii. Se observă că cele două gramatici sunt echivalente (generează acelaşi limbaj) şi în această nouă gramatică S' nu apare în partea dreaptă a nici unei producţii.

Definiţia 3.4. *[Şer87] O gramatică $G = (N, \Sigma, P, S)$ este* de tip LR(k) *pentru $k \geq 0$ dacă din:*

1. $S' \overset{*}{\Rightarrow}_{dr} \alpha A w \Rightarrow_{dr} \alpha\beta w$;
2. $S' \overset{*}{\Rightarrow}_{dr} \gamma B x \Rightarrow_{dr} \alpha\beta y$;
3. $FIRST_k(w) = FIRST_k(y)$

rezultă că $\alpha = \beta, A = B, x = y$.

Pornind de la definiţia elementului de analiză să considerăm un caz particular $[A \to \alpha\beta., u]$. Semnificaţia acestui element de analiză este că $\alpha\beta$ a fost deja detectat în arbore şi astfel s-ar putea reduce subarborele cu frontul $\alpha\beta$ la neterminalul A. Astfel, am determinat când (în ce stare a automatului push-down) se poate face reducerea şi care este manşa. Deci, când se ajunge într-o stare ce conţine element de analiză de forma $[A \to \alpha\beta., u]$ acţiunea va fi de reducere, altfel va fi deplasare. Deoarece starea automatului decide acţiunea, ea va fi de asemenea depusă în stiva de lucru. Configuraţia stivei va fi:
$\$ s_{init} X_1 s_1 \ldots X_m s_m$
unde X_m este simbolul din vârful stivei, iar s_m este starea curentă a automatului. Cele două elemente împreună cu predicţia de lungime k vor determina comportamentul analizorului, caracterizat prin:

- acţiunea care se va efectua şi
- tranziţia în altă stare.

De aceea tabelele de analiză LR(k) au două componente: de acţiune şi de deplasare, numită "goto".

Care sunt şi cum se determină aceste stări? Pentru a răspunde să considerăm elementul de analiză $[A \to \alpha.B\beta, u]$ care, conform definiţiei, implică:
$S \overset{*}{\Rightarrow}_{dr} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w$ şi
$u = FIRST_k(w)$ valabil pentru prefixul viabil $\gamma\alpha$.
Dacă în gramatică există o producţie $B \to \delta$ atunci elementul de analiză $[B \to .\delta, u]$ are, de asemenea, u valid pentru prefixul viabil $\gamma\alpha$:

$S \overset{*}{\Rightarrow} \gamma A w \Rightarrow_{dr} \gamma\alpha B\beta w \Rightarrow_{dr} \gamma\alpha\delta\beta w$.

Această observaţie sugerează faptul că elementele de analiză corespunzătoare unui acelaşi prefix viabil ar trebui grupate într-o mulţime şi că această mulţime caracterizează un pas al analizei, deci va forma o stare. Mulţimea care va conţine toate elementele de analiză corespunzătoare unui prefix viabil va forma o *stare* a automatului.

Starea iniţială este cea care corespunde prefixului vid (încă nu s-a citit nimic de pe banda de intrare), deci va conţine elementul de analiză $[S' \rightarrow .S, u]$. Starea finală, corespunzătoare acceptării unei secvenţe, va trebui să conţină elementul de analiză $[S' \rightarrow S., \$]$ pentru a impune o acţiune de reducere la S', iar predicţia este vidă, adică la rădăcina arborelui, ceea ce corespunde construirii complete a arborelui de analiză sintactică.

Toate analizoarele sintactice LR au aceste caracteristici generale. Modul de calcul al stărilor automatului şi construcţia tabelului de analiză vor fi particularizate pentru diferite tipuri de analizoare LR, şi anume: LR(0), SLR, LR(1), LALR.

Prezentarea fiecărui analizor va urma paşii:

- prezentarea elementului de analiză;
- construirea mulţimii de stări;
- construirea tabelului de analiză;
- funcţionarea analizorului.

3.3.2. Analizor sintactic LR(0)

Elementul de analiză LR(0) are forma $[A \rightarrow \alpha.\beta]$ (predicţia fiind vidă, ea dispare din elementul de analiză), ceea ce înseamnă că vom aplica principiile analizei sintactice LR de determinare a manşei, deplasare şi reducere, dar fără să ţinem seama de predicţie (de ce simbol urmează pe banda de intrare).

Construcţia mulţimii de stări: După cum am mai precizat o stare este o mulţime care conţine toate elementele de analiză corespunzătoare unui prefix viabil. Vom nota cu $\mathcal{E}_0$ mulţimea elementelor de analiză LR(0). Vom folosi o *funcţie de închidere* pentru a construi această mulţime [ASU86]:

$closure : \mathcal{P}(\mathcal{E}_0) \rightarrow \mathcal{P}(\mathcal{E}_0)$

care are următoarele două proprietăţi importante, pe baza cărora vom elabora algoritmul 3.8:

1. $I \in \mathcal{P}(\mathcal{E}_0), I \subseteq closure(I)$

2. $closure(I) = I \cup \{[B \rightarrow .\delta] | [A \rightarrow \alpha.B\beta] \in I\}$, conform observaţiei din paragraful anterior.

Algoritmul 3.8 ClosureLR0

```
INPUT: I-element de analiză; G'- gramatica îmbogăţită
OUTPUT: C = closure(I);
C := {I};
repeat
  for ∀[A → α.Bβ] ∈ C do
    for ∀B → γ ∈ P do
      if [B → .γ] ∉ C then
        C = C ∪ [B → .γ]
      end if
    end for
  end for
until C nu se mai modifică
```

Pentru a determina stările şi cum se deplasează automatul dintr-o stare în altă stare, ceea ce înseamnă că se procesează un simbol, vom folosi o *funcţie notată goto* [ASU86]:

$goto : \mathcal{P}(\mathcal{E}_0) \times (N \cup \Sigma) \rightarrow \mathcal{P}(\mathcal{E}_0)$

Dacă $[A \rightarrow \alpha.X\beta]$ este un element din starea s_i, procesarea lui X înseamnă că X a fost detectat şi partea neprelucrată va conţine doar β, deci vom ajunge într-o nouă stare care va conţine elementul $[A \rightarrow \alpha X.\beta]$:

$goto(s, X) = closure(\{[A \rightarrow \alpha X.\beta] | [A \rightarrow \alpha.X\beta] \in s\})$.

Stările automatului formează o colecţie canonică:

- colecţie - pentru că trebuie să calculăm toate posibilităţile de "deplasare" ale automatului: pentru toate stările s şi pentru toate simbolurile gramaticii $X \in N \cup \Sigma$ vom calcula $goto(s, X)$;
- canonică - pentru că dacă printr-un calcul $goto(s, X)$ obţinem o stare q deja existentă, ea nu va mai fi adăugată la colecţie, în schimb trebuie memorată această tranziţie a automatului: $goto(s, X) = q$.

Colecţia de stări se construieşte după algoritmul 3.9.

Exemplul 3.4. Fie $G = (\{S', S, A\}, \{a, b, c\}, P, S')$, unde producţiile sunt:

$S' \rightarrow S$
$S \rightarrow aA$
$A \rightarrow bA$

Algoritmul 3.9 Col_stariLR0

INPUT: G'- gramatica îmbogăţită
OUTPUT: C - colecţia canonică de stări
$C := \emptyset$;
$s_0 := closure(\{[S' \rightarrow .S]\})$
$C := C \cup \{s_0\}$;
repeat
  for $\forall s \in C$ **do**
    for $\forall X \in N \cup \Sigma$ **do**
      if $goto(s, X) \neq \emptyset$ and $goto(s, X) \notin C$ **then**
        $C = C \cup goto(s, X)$
      end if
    end for
  end for
until C nu se mai modifică

$A \rightarrow c$

$s_0 = closure(\{[S' \rightarrow .S]\}) = \{[S' \rightarrow .S], [S \rightarrow .aA]\}$ { iniţializarea}

Pentru a construi $goto(s, X)$ va trebui să căutăm în starea s un element de analiză care să conţină '.' înaintea lui X. Construim elementul de analiză cu '.' după X şi aplicăm algoritmul de închidere:

$$
\begin{aligned}
s_1 &= goto(s_0, S) = closure(\{S' \rightarrow S.\}) = \{[S' \rightarrow S.]\} \\
s_2 &= goto(s_0, a) = closure(\{[S \rightarrow a.A]\} = \{[S \rightarrow a.A], \\
&\quad [A \rightarrow .bA], [A \rightarrow .c]\} \\
&\quad goto(s_0, b) = \emptyset \\
&\quad goto(s_0, c) = \emptyset \\
&\quad goto(s_1, X) = \emptyset, \forall X \in N \cup \Sigma \\
s_3 &= goto(s_2, A) = closure(\{[S \rightarrow aA.]\}) = \{[S \rightarrow aA.]\} \\
s_4 &= goto(s_2, b) = closure(\{[A \rightarrow b.A]\}) = \{[A \rightarrow b.A], \\
&\quad [A \rightarrow .bA], [A \rightarrow .c]\} \\
s_5 &= goto(s_2, c) = closure(\{[A \rightarrow c.]\}) = \{[A \rightarrow c.]\} \\
s_6 &= goto(s_4, a) = closure(\{[A \rightarrow bA.]\}) = \{[A \rightarrow bA.]\} \\
&\quad goto(s_4, b) = s_4 \\
&\quad goto(s_4, c) = s_5
\end{aligned}
$$

$C = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$. Automatul are 7 stări.

Figura 3.3 descrie automatul finit corespunzător tranziţiilor determinate de funcţia *goto*.

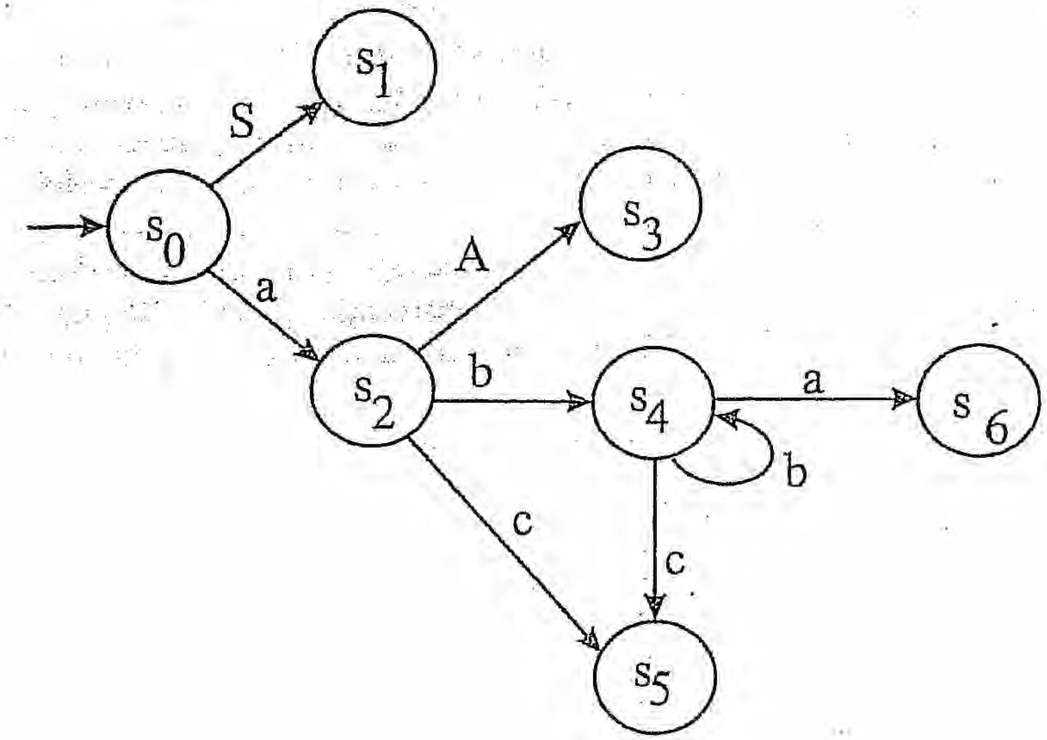

Figura 3.3: Automatul de generare al predicţiilor

Construcţia tabelului LR(0): Tabelul de analiză LR(0) va conţine câte o linie pentru fiecare stare, o coloană pentru acţiune (pentru o anumită stare acţiunea este unică deoarece nu se ţine cont de predicţie), iar în partea de goto câte o coloană pentru fiecare simbol al gramaticii $X \in N \cup \Sigma$.

Regulile de completare a tabelului se bazează pe colecţia de stări şi pe rezultatele funcţiei goto:

1. dacă $[A \rightarrow \alpha.a\beta] \in s_i$ atunci acţiune(i)=s;[4]
2. dacă $[A \rightarrow \beta.] \in s_i$ şi $A \neq S'$ atunci acţiune(i)=r l, unde l - numărul producţiei $A \rightarrow \beta$;[5]
3. dacă $[S' \rightarrow S.] \in s_i$ atunci acţiune(i)=acc;
4. dacă $goto(s_i, X) = s_j$ atunci $goto(i, X) = j$;

[4] s = shift - deplasare
[5] r = reducere

5. starea iniţială a analizorului corespunde mulţimii din colecţie care conţine $[S' \rightarrow .S]$;

6. toate celelalte valori = eroare.

Tabelul de analiză LR(0) corespunzător gramaticii din exemplul 3.4 este:

Nr	acţiune	goto				
		S	A	a	b	c
0	s	1		2		
1	acc					
2	s		3		4	5
3	r1					
4	s		6		4	5
5	r3					
6	r2					

Observaţii:

1. Din starea de acceptare nu există deplasare: dacă s e stare de acceptare, $goto(s, X) = \emptyset, \forall X \in N \cup \Sigma$.

2. Dacă într-o anumită stare s acţiunea este de reducere, atunci pentru starea respectivă $goto(s, X) = \emptyset, \forall X \in N \cup \Sigma$.

3. Putem acum argumenta de ce se lucrează cu gramatica îmbogăţită: presupunem că lucrăm cu o gramatică obişnuită şi considerăm următorul exemplu:

   $G = (\{S\}, \{a, b, c\}, \{S \rightarrow aSbS, S \rightarrow c\}, S)$

   Stările de acceptare ar trebui să fie cele care conţin elementele de analiză $[S \rightarrow aSbS.]$ şi $[S \rightarrow c.]$, dar aceste stări sunt şi cele corespunzătoare acţiunii de reducere. Astfel se creează un conflict între acceptare şi reducere, care poate fi uşor evitat folosind gramatica îmbogăţită şi admiţând ca stare de acceptare doar cea care conţine elementul de analiză $[S' \rightarrow S.]$.

4. Dacă gramatica nu este de tip LR(0), atunci tabelul de analiză va conţine conflicte, care se clasifică în:

   - *conflict deplasare - reducere*: când o stare include elemente de analiză de forma $[A \rightarrow \alpha.\beta]$ şi $[B \rightarrow \gamma.]$, ceea ce ar implica două acţiuni pentru această stare;

   - *conflict reducere - reducere:* când o stare include elemente de analiză de forma $[A \rightarrow \alpha\beta.]$ şi $[B \rightarrow \gamma.]$, în care acţiunea este de reducere, dar se creează conflict relativ la producţia cu care se face reducerea.

   Analiza sintactică LR(0) nu se poate efectua dacă tabelul conţine conflicte.

Funcţionarea analizorului sintactic LR(0): Analizorul sintactic LR(0) este un analizor sintactic de tip deplasare - reducere, în care configuraţiile au forma (α, β, π), unde α e stiva de lucru, cu capul stivei înspre β şi conţine o succesiune de stări şi simboluri din gramatică, β e stiva de intrare, conţinând secvenţa de analizat, iar π este banda de ieşire ce va conţine şirul de producţii folosit în derivarea secvenţei din simbolul de start.

Configuraţia iniţială este $(\$s_0, w\$, \epsilon)$, iar configuraţia finală corespunzătoare acceptării secvenţei este $(\$s_{acc}, \$, \pi)$. Se observă folosirea simbolului special $\$ \notin N \cup \Sigma$ pentru delimitarea celor două stive, ca în cazul analizorului sintactic LL(1).

Tranziţiile se definesc în felul următor:

1. **deplasare**: dacă acţiunea corespunzătoare stării s_m din vârful stivei de lucru este deplasare şi simbolul din vârful stivei de intrare este a_i, atunci: simbolul a_i este mutat din stiva de intrare în stiva de lucru şi noua stare din vârful stivei de lucru va fi $s_j = goto(s_m, a_i)$.

   $(\$s_0x_1 \ldots x_m s_m, a_i \ldots a_n\$, \pi) \vdash (\$s_0x_1 \ldots x_m s_m a_i s_j, a_{i+1} \ldots a_n\$, \pi)$, dacă *acţiune*$(s_m)$=s şi $goto(s_m, a_i) = s_j$;

2. **reducere**: dacă acţiunea corespunzătoare stării s_m din vârful stivei de lucru este reducere cu producţia t, atunci se caută începând cu vârful stivei de lucru, partea dreapta a producţiei respective; se scoate din stivă şi se introduce în stivă partea stânga a producţiei t, după care se calculează starea în care s-a ajuns: valoarea funcţiei *goto* pentru starea din vârful stivei după scoaterea manşei s_{m-p} şi partea stângă a producţiei t şi se depune în stivă; în final se actualizează banda de ieşire cu producţia t; ţinând cont de faptul că avem un analizor ascendent producţiile se folosesc în derivare în ordine inversă faţă de cum sunt detectate; de aceea, adăugarea pe banda de ieşire se face în faţă:

   $(\$s_0 \ldots x_m s_m, a_i \ldots a_n\$, \pi) \vdash (\$s_0 \ldots x_{m-p} s_{m-p} A s_j, a_i \ldots a_n\$, t\pi)$, dacă *acţiune*$(s_m)$=r t, t - numărul producţiei $A \rightarrow \beta$, $\beta = x_{m-p+1} \ldots x_m$ şi $goto(s_{m-p}, A) = s_j$;

3. **acceptare**: $(\$s_i, \$, \pi) = acc$ dacă *acţiune(i)=acc*;

4. **eroare**, în celelalte cazuri.

Analiza sintactică LR(0) funcţionează după algoritmul 3.10.

Considerând gramatica din exemplul 3.4, pentru care am construit deja tabelul de analiză LR(0), dorim să analizăm secvenţa "abbc".

Stiva de lucru	Stiva de intrare	Banda de ieşire	Acţiunea
$0	abbc$		deplasare
$0a2	bbc$		deplasare
$0a2b4	bc$		deplasare
$0a2b4b4	c$		deplasare
$0a2b4b4c5	$	3	reducere
$0a2b4b4A6	$	2,3	reducere
$0a2b4A6	$	2,2,3	reducere
$0a2A3	$	1,2,2,3	reducere
$0S1	$		acceptare
acc			

Exemplul 3.5. Fie gramatica

$G = (\{S', E, T, F\}, \{+, *, (,), id, const\}, P, S')$, cu următoarele producţii:
$S' \to E$
$E \to E + T | T$
$T \to T * F | F$
$F \to (E) | id | const$

care e des întâlnită în descrierea sintaxei limbajelor de programare pentru generarea expresiilor aritmetice (am considerat doar cazul expresiilor aritmetice care conţin adunarea şi înmulţirea). Construind mulţimea de stări corespunzătoare acestei gramatici obţinem:

$$\begin{aligned} s_0 &= closure(\{[S' \to .E]\}) = \{[S' \to .E], [E \to .E + T], [E \to .T], \\ &\quad [T \to .T + F], [T \to .F], [F \to .(E)], [F \to .id], [F \to .const]\} \\ s_1 &= goto(s_0, E) = closure(\{S' \to E.\}) = \{[S' \to E.], [E \to E. + T]\} \end{aligned}$$

Observăm că la completarea tabelului de analiză LR(0), pentru starea s_1 vom avea conflict; primul element de analiză implică acţiune de acceptare, iar al doilea acţiune de deplasare. Ca urmare, această gramatică nu este de tip LR(0).

Algoritmul 3.10 Anal_sintLR0

INPUT:

- tabelul de analiză LR(0) fără conflicte;
- producţiile gramaticii numerotate;
- şirul de analizat $w = a_1 a_2 ... a_n a_{n+1}$, unde $a_{n+1} = \$$.

OUTPUT: este sau nu secvenţa acceptată; dacă da, şirul de producţii folosit.

```
j:=1; stare:=0; stiva := '$';out:='';
terminare := false;
repeat
  if actiune(stare)=s then
    push(a_j);
    j := j+1
  else
    if actiune(stare)=r t then
      cauta_prod(t,pd,ps);
      pop(pd);
      temp:=varf(stiva);
      push(ps);
      push(goto(temp,ps));
      out:=t+out;
    else
      if actiune(stare)=acc then
        write("succes"); write(out);
        terminare := true;
      end if
      if actiune(stare)=err then
        write("eroare")
        terminare := true;
      end if
    end if
  end if
until terminare
```

Gramaticile de tip LR(0) sunt destul de restrictive şi de aceea, va fi necesar să elaborăm un algoritm mai performant, în care să ţinem cont de predicţie.

3.3.3. Analizor sintactic SLR

Denumirea acestui analizor provine de la Simple LR(1), adică un analizor LR, în care predicţia este de lungime 1, *simplificat*. Această simplificare se bazează pe următoarea idee: predicţia va fi luată în considerare în momentul în care se face o reducere, în rest analizorul funcţionează ca şi LR(0) [DeR71].

Care este beneficiul acestei metode? Dacă ne imaginăm tabelul de analiză SLR pentru o anumită gramatică şi ţinem cont de predicţie, atunci pentru o stare în care se efectuează reducere, pe linia respectivă am putea avea: reducere cu producţia t dacă predicţia este a şi reducere cu producţia t' dacă predicţia este b, ceea ce înseamnă că astfel se poate elimina un conflict reducere - reducere dintr-un analizor sintactic LR(0).

Elementul de analiză SLR este identic cu cel LR(0). De asemenea, **construcţia mulţimii de stări** este aceeaşi. Vom obţine aceeaşi colecţie canonică. Diferenţele între cele două analizoare apar la construcţia tabelului de analiză SLR şi funcţionarea analizorului.

Construcţia tabelului de analiză SLR: conform caracteristicii definitorii a acestui analizor sintactic reducerea se face ţinând cont de următorul simbol de pe banda de intrare (simbolul imediat următor), ceea ce ne indică faptul că va trebui să folosim funcţia FOLLOW (aceeaşi funcţie folosită şi în cadrul analizorului sintactic LL(1), a cărui definiţie şi mod de calcul sunt date în paragraful 3.2.3).

Tabelul de analiză SLR "moşteneşte" unele caracteristici de la tabelul LR(0): vom avea câte o linie pentru fiecare stare şi ar trebui să avem câte o coloană pentru fiecare simbol din gramatică $X \in N \cup \Sigma$ în partea de *goto*. Partea de *acţiune* a tabelului SLR ar trebui să conţină câte o coloană pentru fiecare predicţie posibilă $u \in \Sigma \cup \{\$\}$. Se observă că pentru fiecare simbol terminal al gramaticii ar trebui să se aloce câte două coloane, ceea ce ar duce la o gestiune ineficientă a spaţiului.

Putem îmbunătăţi această situaţie făcând următoarea convenţie: pentru simbolurile terminale, părţile de tabel corespunzătoare acţiunii şi deplasării se unifică. Cu alte cuvinte, pentru fiecare simbol terminal, celula de tabel va conţine după acţiunea corespunzătoare stării respective şi valoarea funcţiei goto. Ţinând cont de observaţiile 1 şi 2 din paragraful precedent, putem deduce faptul că funcţia goto va furniza valori doar în stările care corespund deplasării. Astfel, această unificare de coloane nu

creează confuzii în construcţia tabelului.

Regulile de completare a tabelului SLR sunt:

1. Dacă $[A \to \alpha.\beta] \in s_i$ atunci acţiune(i,u)=s, $\forall u \in \Sigma$.
2. Dacă $[A \to \beta.] \in s_i, A \neq S'$ atunci acţiune(i,u)=r l, unde l este numărul producţiei $A \to \beta, \forall u \in FOLLOW(A)$.
3. Dacă $[S' \to S.] \in s_i$ atunci acţiune(i,$)=acc.
4. Dacă $goto(s_i, X) = s_j$ atunci $goto(i, X) = j$.
5. Starea iniţială a analizorului corespunde mulţimii din colecţie care conţine $[S' \to .S]$.
6. Toate celelalte valori = eroare.

Observaţii:

1. Comparând cu construcţia tabelului de analiză LR(0), regulile 1, 2, 3 sunt uşor modificate, iar regulile 4, 5, 6 rămân identice.
2. Să presupunem că într-o stare j acţiunea este de deplasare şi $goto(j, a) = \emptyset$. În acest caz, celula (j, a) ar conţine doar s (acţiunea). Datorită faptului că dacă se ajunge în această stare, automatul corespunzător analizorului se blochează (nu se mai poate deplasa în altă stare), nu are sens să reţinem deplasarea pentru o astfel de stare. De aceea, regula 1 se aplică în felul următor:

   1' Dacă $[A \to \alpha.a\beta] \in s_i$ atunci actiune(i,u)=s, $\forall u \in \Sigma$, pentru care $goto(i, u) \neq \emptyset$.
3. O gramatică este de tip SLR dacă nu conţine conflicte.

Exemplul 3.6. Fie gramatica $G = (\{S', E, T, \}, \{+, (,), id, const\}, P, S')$, cu următoarele producţii (o variantă simplificată a gramaticii care generează expresii aritmetice, generând doar adunări):

$S' \to E$
$E \to T$
$E \to E + T$
$T \to (E)$
$T \to id$
$T \to const$

Aplicând algoritmul pentru FOLLOW obţinem:
$FOLLOW(E) = \{+,), \epsilon\}$

$FOLLOW(T) = \{+,), \epsilon\}$

$s_0 = closure(\{[S' \to .E]\}) = \{[S' \to .E], [E \to .T], [E \to .E+T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_1 = goto(s_0, E) = closure(\{[S' \to E.], [E \to E.+T]\}) = \{[S' \to E.], [E \to E.+T]\}$

$s_2 = goto(s_0, T) = closure(\{[E \to T.]\}) = \{[E \to T.]\}$

$s_3 = goto(s_0, id) = closure(\{[T \to id.]\}) = \{[T \to id.]\}$

$s_4 = goto(s_0, const) = closure(\{[T \to const.]\}) = \{[T \to const.]\}$

$s_5 = goto(s_0, () = closure(\{[T \to (.E)]\}) = \{[T \to (.E)], [E \to .T], [E \to .E+T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_6 = goto(s_1, +) = closure(\{[E \to E+.T]\}) = \{[E \to E+.T], [T \to .(E)], [T \to .id], [T \to .const]\}$

$s_7 = goto(s_5, E) = closure(\{[T \to (E.)], [E \to E.+T]\}) = \{[T \to (E.)], [E \to E.+T]\}$

$goto(s_5, T) = s_2; goto(s_5, () = s_5; goto(s_5, id) = s_3;$

$goto(s_5, const) = s_4;$

$s_8 = goto(s_6, T) = closure(\{[E \to E+T.]\}) = \{[E \to E+T.]\}$

$goto(s_6, () = s_5; goto(s_6, id) = s_3; goto(s_6, const) = s_4;$

$s_9 = goto(s_7,)) = closure(\{[T \to (E).]\}) = \{[T \to (E).]\}$

$goto(s_7, +) = s_6.$

acţiune							goto	
Nr	id	const	+	(	)	$	E	T
0	s3	s4		s5			1	2
1			s6			acc		
2			1		1	1		
3			4		4	4		
4			5		5	5		
5	s3	s4		s5			7	2
6	s3	s4		s5				8
7			s6		s9			
8			2		2	2		
9			3		3	3		

Se poate observa că dacă am fi considerat analizorul sintactic LR(0),

am fi avut conflict în starea s_1, între deplasare şi acceptare.

Funcţionarea analizorului sintactic SLR este asemănătoare cu cea a analizorului LR(0), cu două mici modificări:

- condiţiilor de execuţie a acţiunii de reducere se adaugă condiţia de verificare a predicţiei: adică reducerea va avea loc doar dacă în tabel avem $actiune(s_m, u) = r\ t$ şi u este vârful stivei de intrare;
- **acceptare:** $(\$s_i, \$, \pi) = acc$ dacă $actiune(i, \$) = acc$ (stiva de intrare conţine doar simbolul \$).

Folosind tabelul de analiză SLR pentru gramatica din exemplul 3.6, efectuăm analiza secvenţei $id + const$:

Stiva de lucru	Stiva de intrare	Banda de ieşire	Acţiunea
\$0	id+const\$		deplasare
\$0id3	+const\$	4	reducere
\$0T2	+const\$	1,4	reducere
\$0E1	+const\$	1,4	deplasare
\$0E1+6	const\$	1,4	deplasare
\$0E1+6const4	\$	1,4	reducere
\$0E1+6T8	\$	2,5,1,4	reducere
\$0E1	\$	2,5,1,4	acceptare
acc			

3.3.4. Analizor sintactic LR(1)

Analizorul sintactic LR(1) foloseşte un simbol (predicţia este de lungime 1) pentru a lua o decizie [Cha87, Spe88]. Se obţine astfel un algoritm mai puternic, deoarece se micşorează posibilitatea de apariţie a unui conflict. Analizoarele sintactice LR(1) reprezintă o opţiune eficientă de implementare pentru compilatoarele reale. Studiul acestui analizor se va face comparativ cu metodele LR prezentate anterior.

Elementul de analiză LR(1) are următoare formă $[A \to \alpha.\beta, u]$, conţinând o producţie a gramaticii $A \to \alpha\beta$ îmbogăţită cu punct şi un simbol u ca predicţie şi are următoarea semnificaţie: α se află în vârful stivei de lucru, iar prefixul secvenţei rămase poate fi derivat din βu.

Construcţia mulţimii de stări LR(1): Stările analizorului sunt, ca în cazul analizorului sintactic LR(0), specificate de mulţimi de elemente de analiză. Funcţiile folosite în construcţie, *closure* şi *goto*, trebuie adaptate corespunzător formei elementului de analiză.

Funcţia de închidere closure va fi calculată folosind următoarele două observaţii:

1. $C \in closure(C)$.

2. Considerăm elementul de analiză $[A \rightarrow \alpha.B\beta, a]$, care are următoarea semnificaţie:

   $S \overset{*}{\Rightarrow}_{dr} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w$ şi $a = FIRST_k(w)$

   valabil pentru prefixul viabil $\gamma\alpha$. Vom folosi o producţie pentru B în derivările ulterioare, fie ea $B \rightarrow \delta$ şi vom obţine:

   $S \overset{*}{\Rightarrow}_{dr} \gamma Aw \Rightarrow_{dr} \gamma\alpha B\beta w \Rightarrow_{dr} \gamma\alpha\delta\beta w$

   astfel că elementul de analiză $[B \rightarrow .\delta, ?]$ va fi valabil pentru prefixul viabil $\gamma\alpha$. Care este predicţia corespunzătoare acestul element de analiză? Primul simbol care urmează după δ în forma propoziţională şi anume $FIRST(\beta w)$. Dar în acest şir de derivări s-a presupus deja că $FIRST(w) = a$, obţinând predicţia $FIRST(\beta a)$. Deci, vom avea următoarea regulă de calcul pentru *closure*, aplicată în algoritmul 3.11:

   $\forall [A \rightarrow \alpha.B\beta, a] \in closure(C), \forall B \rightarrow \delta \in P, [B \rightarrow .\delta, b] \in closure(C)$ pentru $\forall b \in FIRST(\beta a)$

Algoritmul 3.11 ClosureLR1

INPUT: I-element de analiză; G'- gramatica îmbogăţită;
$FIRST(X), \forall X \in N \cup \Sigma$;
OUTPUT: C_1 = closure(I);
$C_1 := \{I\}$;
repeat
  for $\forall [A \rightarrow \alpha.B\beta, a] \in C_1$ do
    for $\forall B \rightarrow \gamma \in P$ do
      for $\forall b \in FIRST(\beta a)$ do
        if $[B \rightarrow .\gamma, b] \notin C_1$ then
          $C_1 = C_1 \cup [B \rightarrow .\gamma, b]$
        end if
      end for
    end for
  end for
until C_1 nu se mai modifică

Definiţia funcţiei *goto* se actualizează în:

$goto(s, X) = closure(\{[A \rightarrow \alpha X.\beta, a] | [A \rightarrow \alpha.X\beta, a] \in s\})$.

Pentru calculul colecţiei canonice de stări LR(1) modificăm algoritmul corespunzător de la LR(0) (algoritmul 3.9), obţinând algoritmul 3.12.

Algoritmul 3.12 Col_stariLR1

INPUT: G'- gramatica îmbogăţită
OUTPUT: C - colecţia canonică de stări
$C_1 := \emptyset$;
$s_0 := closure(\{[S' \rightarrow .S, \$]\})$
$C_1 := C_1 \cup \{s_0\}$;
repeat
  for $\forall s \in C_1$ do
    for $\forall X \in N \cup \Sigma$ do
      T:=goto(s, X);
      if $T \neq \emptyset$ and $T \notin C_1$ then
        $C_1 = C_1 \cup T$
      end if
    end for
  end for
until C_1 nu se mai modifică

Exemplul 3.7. Fie gramatica $G = (\{S', S, A\}, \{a, b\}, P, S')$ unde producţiile sunt:

$S' \rightarrow S$
$S \rightarrow AA$
$A \rightarrow aA$
$A \rightarrow b$

pentru care dorim să calculăm colecţia de stări. Vom avea nevoie de valoarea funcţiei $FIRST$ pentru neterminalul A. Se observă uşor că $FIRST(A) = \{a, b\}$.

$$s_0 = closure(\{[S' \rightarrow .S, \$]\}) = \{[S' \rightarrow .S, \$], [S \rightarrow .AA, \$], [A \rightarrow .aA, a], [A \rightarrow .aA, b], [A \rightarrow .b, a], [A \rightarrow .b, b]\}$$

Să analizăm construcţia stării s_0, pentru a înţelege mecanismul de calcul al predicţiilor din stările LR(1). Iniţializăm colecţia cu $[S' \rightarrow .S, \$]$. Avem un element de analiză cu punct înaintea neterminalului S, deci vom adăuga elementul $[S \rightarrow .AA, u]$, unde u se calculează ca fiind $FIRST(\$)$ (după S în partea dreaptă a producţiei $S' \rightarrow S$ nu mai avem nimic, iar predicţia în respectivul element de analiză este \$).

Luăm acum în considerare elementul de analiză $[S \rightarrow .AA, \$]$. Avem punct înaintea lui A, deci vom adăuga elemente de analiză corespunzătoare

producţiilor lui A, iar predicţia este $FIRST(A\$)$ (după A, în partea dreaptă urmează A). $FIRST(A) = \{a, b\}$, deci vom adăuga pentru fiecare producţie câte un element de analiză cu predicţia a, respectiv b: $[A \to .aA, a], [A \to .aA, b]$ şi $[A \to .b, a], [A \to .b, b]$.

$$
\begin{aligned}
s_1 &= goto(s_0, S) = closure(\{[S' \to S., \$]\}) = \{[S' \to S., \$]\} \\
s_2 &= goto(s_0, A) = closure(\{[S \to A.A, \$]\}) = \{[S \to A.A, \$], \\
&\quad [A \to .aA, \$], [A \to .b, \$]\} \\
s_3 &= goto(s_0, a) = closure(\{[A \to a.A, a], [A \to a.A, b]\}) = \\
&\quad \{[A \to a.A, a], [A \to a.A, b], [A \to .aA, a], [A \to .aA, b], \\
&\quad [A \to .b, a], [A \to .b, b]\} \\
s_4 &= goto(s_0, b) = closure(\{[A \to b., a], [A \to b., b]\}) = \\
&\quad \{[A \to b., a], [A \to b., b]\} \\
&\quad goto(s_1, X) = \emptyset, \forall X \in N \cup \Sigma \\
&\quad (\textit{vom considera doar multimile nevide in continuare}) \\
s_5 &= goto(s_2, A) = closure(\{[S \to AA., \$]\}) = \{[S \to AA., \$]\} \\
s_6 &= goto(s_2, a) = closure(\{[A \to a.A, \$]\}) = \{[A \to a.A, \$], \\
&\quad [A \to .aA, \$], [A \to .b, \$]\} \\
s_7 &= goto(s_2, b) = closure(\{[A \to b., \$]\}) = \{[A \to b., \$]\} \\
s_8 &= goto(s_3, A) = closure(\{[A \to aA., a], [A \to aA., b]\}) = \\
&\quad \{[A \to aA., a], [A \to aA., b]\} \\
&\quad goto(s_3, a) = s_3 \\
&\quad goto(s_3, b) = s_4 \\
s_9 &= goto(s_6, A) = closure(\{[A \to aA., \$]\}) = \{[A \to aA., \$]\} \\
&\quad goto(s_6, a) = s_6 \\
&\quad goto(s_6, b) = s_7
\end{aligned}
$$

Construcţia tabelului de analiză LR(1): Tabelul LR(1) are acelaşi format ca şi tabelul SLR, iar completarea lui se face pe baza următoarelor reguli:

1. Dacă $[A \to \alpha.\beta, u] \in s_i$ atunci $actiune(i, v) = s, \forall v \in \Sigma$, cu aceeaşi observaţie ca în cazul tabelului de analiză SLR, şi anume că nu se vor completa decât celulele de tabel pentru care $goto(i, v) \neq \emptyset$.

2. Dacă $[A \to \beta., u] \in s_i, A \neq S'$ atunci $actiune(i, u) = rl$, unde l este numărul producţiei $A \to \beta$.

3. Dacă $[S' \to S., \$] \in s_i$ atunci $actiune(i, \$) = acc$.

4. Dacă $goto(s_i, X) = s_j$ atunci $goto(i, X) = j$.

5. Starea iniţială a analizorului corespunde mulţimii din colecţie care conţine $[S' \to .S, \$]$.

6. Toate celelalte valori = eroare.

Pentru gramatica şi colecţia de stări din exemplul 3.7, tabelul de analiză LR(1) este:

	a	b	$	S	A
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	3	3			
5			1		
6	s6	s7			9
7			3		
8	2	2			
9			2		

Funcţionarea analizorului LR(1) este similară cu cea a analizorului SLR.

Principalul dezavantaj al acestei metode este acela că se creează un număr mare de stări, fapt care se poate observa şi din exemplul următor, în care am luat în calcul o gramatică foarte mică, care generează doar expresii aritmetice bazate pe adunare între identificatori. Se vor obţine, pentru această gramatică de dimensiuni reduse, 16 stări.

Exemplul 3.8. Fie gramatica $G = (\{S', E, T, \}, \{+, (,), id\}, P, S')$, cu următoarele producţii:

$S' \to E$
$E \to E + T$
$E \to T$
$T \to (E)$
$T \to id$
FIRST(E) + FIRST(T)={id, (}

Mulţimea de stări LR(1) este:

$s_0 = closure\{[S' \to .E, \$]) = \{[S' \to .E, \$], [E \to .E+T, \$],$
$[E \to .E+T, +], [E \to .T, \$], [E \to .T, +], [T \to .id, \$],$
$[T \to .id, +], [T \to .(E), \$], [T \to .(E), +]\}$

$s_1 = goto(s_0, E) = closure(\{[S' \to E., \$], [E \to E.+T, \$],$
$[E \to E.+T, +]\}) = \{[S' \to E., \$], [E \to E.+T, \$],$
$[E \to E.+T, +]\}$

$s_2 = goto(s_0, T) = closure(\{[E \to T., \$], [E \to T., +]\}) =$
$\{[E \to T., \$], [E \to T., +]\}$

$s_3 = goto(s_0, id) = closure(\{[T \to id., \$], [T \to id., +]\}) =$
$\{[T \to id., \$], [T \to id., +]\}$

$s_4 = goto(s_0, () = closure(\{[T \to (.E), \$], [T \to (.E), +]\}) =$
$\{[T \to (.E), \$], [T \to (.E), +], [E \to .E+T,)], [E \to .T,)],$
$[E \to .E+T, +], [E \to .T, +], [T \to .id, +], [T \to .id,)],$
$[T \to .(E), +], [T \to .(E),)]\}$

$s_5 = goto(s_1, +) = closure(\{[E \to E+.T, \$], [E \to E+.T, +]\}) =$
$\{[E \to E+.T, \$], [E \to E+.T, +], [T \to .id, \$], [T \to .(E), \$],$
$[T \to .id, +], [T \to .(E), +]\}$

$s_6 = goto(s_4, E) = closure(\{[T \to (E.), \$], [T \to (E.), +],$
$[E \to E.+T,)], [E \to E.+T, +]\}) = \{[T \to (E.), \$],$
$[T \to (E.), +], [E \to E.+T,)], [E \to E.+T, +]\}$

$s_7 = goto(s_4, T) = \{[E \to T.,)], [E \to T., +]\}$

$s_8 = goto(s_4, id) = \{[T \to id.,)], [T \to id., +]\}$

$s_9 = goto(s_4, () = closure(\{[T \to (.E), +], [T \to (.E),)]\}) =$
$\{[T \to (.E), +], [T \to (.E),)], [E \to .E+T,)], [E \to .E+T, +],$
$[E \to .T, +], [E \to .T,)][T \to .id, +], [T \to .id,)],$
$[T \to .(E), +], [T \to .(E), +]\}$

$s_{10} = goto(s_5, T) = \{[E \to E+T., \$], [E \to E+T., +]\}$
$goto(s_5, id) = s_3$
$goto(s_5, () = s_4$

$s_{11} = goto(s_6,)) = \{[T \to (E)., \$], [T \to (E)., +]\}$

$s_{12} = goto(s_6, +) = closure(\{[E \to E+.T,)], [E \to E+.T, +]\}) =$
$\{[E \to E+.T,)], [E \to E+.T, +], [T \to .id,)], [T \to .id, +],$

$[T \to .(E),)], [T \to .(E), +]\}$

$s_{13} = goto(s_9, E) = \{[T \to (E.), +], [T \to (E.),)], , [E \to E.+T,)],$
$[E \to E.+T, +]\}$
$goto(s_9, T) = s_7$
$goto(s_9, id) = s_8$
$goto(s_9, () = s_9$

$s_{14} = goto(s_{12}, T) = \{[E \to E+T.,)], [E \to E+T., +]\}$
$goto(s_{12}, id) = s_8$
$goto(s_{12}, () = s_9$

$s_{15} = goto(s_{13},)) = \{[T \to (E)., +], [T \to (E).,)]\}$
$goto(s_{13}, +) = s_{12}$

Tabelul LR(1) este:

acţiune						goto	
Nr	id	(	)	+	$	E	T
0	s3	s4				1	2
1				s5	acc		
2				r2	r2		
3				r3	r3		
4	s8	s9				6	7
5	s3	s4					10
6			s11	s12			
7			r2	r2			
8			r3	r3			
9	s8	s9				13	7
10				r1	r1		
11				r4	r4		
12	s8	s9					14
13			s15	s12			
14			r1	r1			
15			r4	r4			

Analiza LR(1) a secvenţei $id+(id+id)$ se desfăşoară în paşii următori:

Stiva de lucru	Stiva de intrare	Banda de ieşire	Acţiune
$0	id+(id+id)$		deplasare
$0id3	+(id+id)$		reducere
$0T2	+(id+id)$	3	reducere
$0E1	+(id+id)$	2,3	deplasare
$0E1+5	(id+id)$	2,3	deplasare
$0E1+5(4	id+id)$	2,3	deplasare
$0E1+5(4id8	+id)$	2,3	reducere
$0E1+5(4T7	+id)$	3,2,3	reducere
$0E1+5(4E6	+id)$	2,3,2,3	deplasare
$0E1+5(4E6+12	id)$	2,3,2,3	deplasare
$0E1+5(4E6+12id8	)$	2,3,2,3	reducere
$0E1+5(4E6+12T14	)$	3,2,3,2,3	reducere
$0E1+5(4E6	)$	1,3,2,3,2,3	deplasare
$0E1+5(4E6)12	$	1,3,2,3,2,3	reducere
$0E1+5T10	$	4,1,3,2,3,2,3	reducere
$0E1	$	1,4,1,3,2,3,2,3	acceptare
acc			

3.3.5. Analizor sintactic LALR

Denumirea acestui analizor reprezintă o abreviere pentru Look-Ahead-LR(1)[6], iar principiul de funcţionare are la bază următoarele observaţii [LLH71]:

În primul rând, în mulţimea de stări LR(1) există multe elemente asemănătoare. Dacă am ignora predicţiile atunci multe stări ar fi egale. Într-un element de analiză LR(1) de forma $[A \rightarrow \alpha.\beta, u]$, $A \rightarrow \alpha.\beta$ se numeşte *nucleu*, iar u reprezintă *predicţia*. Nucleele elementelor de analiză LR(1) reprezintă, de fapt, elemente de analiză LR(0).

A doua observaţie se remarcă în următoarea situaţie: fie două stări s_i şi s_j. Starea s_i conţine un element de analiză $[A \rightarrow \alpha., u]$, ceea ce va determina o reducere cu producţia $A \rightarrow \alpha$ (actiune[i,u] = r t, t fiind numărul producţiei) şi apoi, automatul va ajunge din starea s_i într-o stare s_m. Starea s_j conţine un element de analiză $[A \rightarrow \alpha., v]$, care, la rândul lui, va determina reducerea cu *aceeaşi* producţie $A \rightarrow \alpha$ (actiune[i,v] = r t) şi apoi analiza va ajunge într-o stare s_n, în majoritatea cazurilor diferită de s_m.

Totuşi, pare relativ bizar ca după efectuarea reducerii cu o aceeaşi producţie să se ajungă în stări diferite. Această situaţie se explică prin faptul că reducerea este decisă de tabel, iar căutarea în tabel ţine cont, în primul rând, de predicţie şi nu de producţia cu care se face reducerea.

[6] Look-Ahead = privire înainte

Analizorul sintactic LALR propune "să se privească înainte la acţiune şi apoi la predicţie" şi, prin urmare, astfel de stări să nu fie diferenţiate: s_i şi s_j ar trebui să reprezinte o singură stare $s_{i,j}$, în care predicţia să cuprindă toate predicţiile posibile din stările unificate.

Starea unificată $s_{i,j}$ va conţine elementul de analiză $[A \rightarrow \alpha., u/v]$, în care caracterul '/' este folosit pentru separarea predicţiilor.

Principiul LALR [LLH71] constă în unificarea tuturor stărilor LR(1) care au acelaşi nucleu, cu predicţiile corespunzătoare, cu restricţia de a nu se crea conflicte.

Construcţia unui analizor sintactic LALR urmează aceiaşi paşi ca orice analizor LR(k), putându-se alege una din variante:

1. Se pleacă de la mulţimea de stări şi tabelul LR(1) şi se construieşte mulţimea de stări şi tabelul LALR prin unificarea stărilor LR(1): pentru fiecare stare s_i se verifică dacă elementele de analiză din s_i şi $s_j, \forall j = 1, i-1$, au acelaşi nucleu şi nu se creează conflicte. Atunci atât starea s_i, cât şi starea s_j se înlocuiesc cu starea $s_{i,j}$ peste tot în tabel (inclusiv în partea de goto).

2. Se construieşte direct mulţimea de stări LALR prin adaptarea algoritmului LR(1) astfel:

Acest algoritm diferă de *colecţie_stăriLR1* prin faptul că înainte de a adăuga o stare la colecţie verifică în liniile 11–17 dacă nucleele acestei stări nu apar în altă stare precedentă. Funcţia *egal_nuclee(T,s')* are scopul de a returna true, dacă există o corespondenţă biunivocă între nucleele din stările T şi s' (şi nu se creează conflicte) si false în caz contrar. Dacă rezultatul este true, atunci elementele de analiză corespunzătoare din s' sunt modificate astfel încât să conţină predicţiile din starea T, operaţie realizată în algoritm prin apelul *modifica(s',T)* din linia 16. Doar dacă se construieşte o stare T cu nuclee diferite sau condiţia de evitare a conflictelor nu a fost îndeplinită se adaugă starea T la colecţia canonică.

Funcţionarea analizorului LALR este identică cu a analizorului LR(1).

Exemplul 3.9. Considerând gramatica din exemplul 3.8, mulţimea de stări calculată şi tabelul LR(1) corespunzător, folosim prima strategie de construcţie pentru LALR. În primul rând vom determina stările care pot fuziona şi observăm în colecţia de stări LR(1) următoarele perechi de stări cu nuclee identice: (2,7), (3,8), (4,9), (6,13), (10,14), (11,15). Se observă că numărul de stări se reduce simţitor, în loc de 16 stări avem doar 10. Figura 3.4 ilustrează automatul rezultat din fuziunea stărilor. Tabelul de analiză LALR este:

Algoritmul 3.13 colecţie_stăriLALR

```
INPUT: G'
OUTPUT: C1
C1 := ∅;
s0 = closure([S' → .S, $]);
C1 := C1 ∪ {s0};
repeat
   for ∀s ∈ C1 do
      for ∀X ∈ N ∪ Σ do
         T:=goto(s, X);
         if (T ≠ ∅)and(T ∉ C1) then
            flag := true;
            for ∀s' ∈ C1, s' ≠ T do
               if egal_nuclee(T,s') then
                  modifica(s',T); flag := false;
               end if
            end for
            if flag then
               C1 := C1 ∪ T
            end if
         end if
      end for
   end for
until C1 nu se modifică
```

acţiune						goto	
Nr	id	(	)	+	$	E	T
0	s[3,8]	s[4,9]				1	[2,7]
1				s5	acc		
[2,7]			r2	r2	r2		
[3,8]			r3	r3	r3		
[4,9]	s[3,8]	s[4,9]				[6,13]	[2,7]
5	s[3,8]	s[4,9]					[10,14]
[6,13]			s[11,15]	s12			
[10,14]			r1	r1	r1		
[11,15]			r4	r4	r4		
12	s[3,8]	s[4,9]					[10,14]

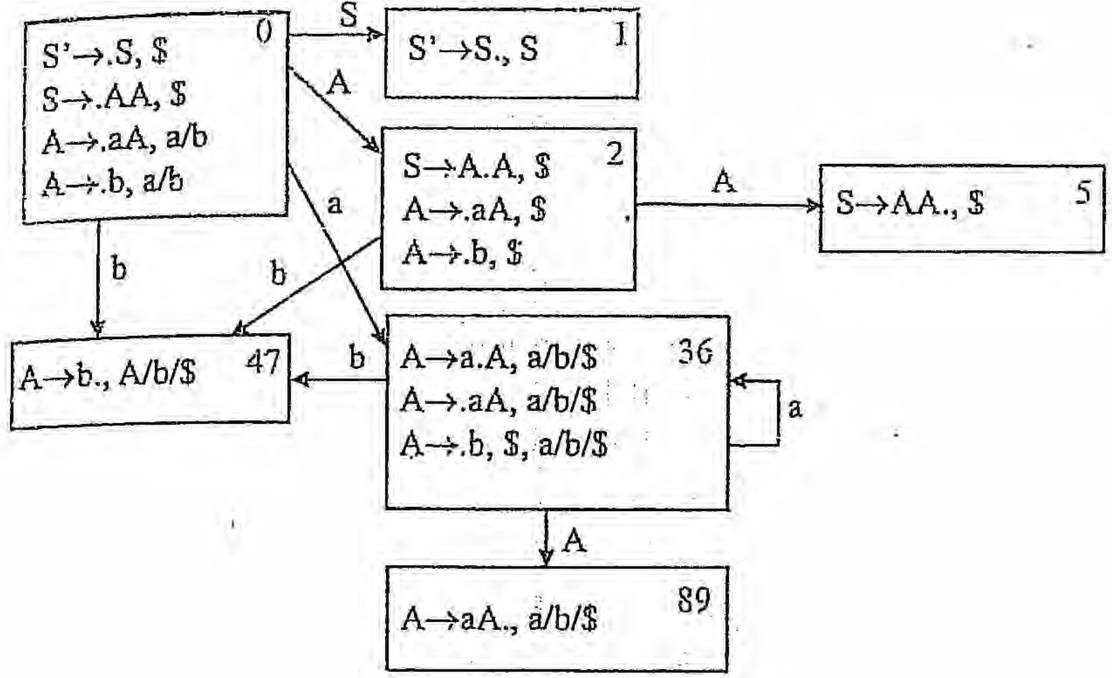

Figura 3.4: Automatul de generare a predicţiilor după fuziunea stărilor

Exemplul 3.10. Fie gramatica $G = (\{S', S, A\}, \{a, b\}, P, S')$ unde producţiile sunt:

$S' \to S$
$S \to AA$
$A \to aA$
$A \to b$

pentru care dorim să calculăm colecţia de stări LALR. Vom avea nevoie de valoarea funcţiei $FIRST$ pentru neterminalul A: $FIRST(A) = \{a, b\}$.

$$
\begin{aligned}
s_0 &= closure(\{[S' \to .S, \$]\}) = \{[S' \to .S, \$], [S \to .AA, \$], [A \to .aA, a], \\
&\quad [A \to .aA, b], [A \to .b, a], [A \to .b, b]\} \\
s_1 &= goto(s_0, S) = closure(\{[S' \to S., \$]\}) = \{[S' \to S., \$]\} \\
s_2 &= goto(s_0, A) = closure(\{[S \to A.A, \$]\}) = \{[S \to A.A, \$], \\
&\quad [A \to .aA, \$], [A \to .b, \$]\} \\
s_3 &= goto(s_0, a) = closure(\{[A \to a.A, a], [A \to a.A, b]\}) = \\
&\quad \{[A \to a.A, a], [A \to a.A, b], [A \to .aA, a], [A \to .aA, b], [A \to .b, a], \\
&\quad [A \to .b, b]\} \\
s_4 &= goto(s_0, b) = closure(\{[A \to b., a], [A \to b., b]\}) = \{[A \to b., a],
\end{aligned}
$$

$[A \to b., b]\}$

$goto(s_1, X) = \emptyset, \forall X \in N \cup \Sigma$

$s_5 = goto(s_2, A) = closure(\{[S \to AA., \$]\}) = \{[S \to AA., \$]\}$

$goto(s_2, a) = closure(\{[A \to a.A, \$]\}) = \{[A \to a.A, \$],$

$[A \to .aA, \$], [A \to .b, \$]\}$

Avem aceleaşi nuclee ca şi în s_3, deci s_3 devine:

$s_3 = \{[A \to a.A, a/b/\$], [A \to .aA, a/b/\$], [A \to .b, a/b/\$]\}$

$goto(s_2, b) = closure(\{[A \to b., \$]\}) = \{[A \to b., \$]\}$

Avem aceleaşi nuclee ca şi în s_4, deci s_4 devine:

$s_4 = \{[A \to b., a/b/\$]\}$

$s_6 = goto(s_3, A) = closure(\{[A \to aA., a/b/\$]\}) = \{[A \to aA., a/b/\$],$

$goto(s_3, a) = s_3$

$goto(s_3, b) = s_4$

Tabelul de analiză LALR corespunzător este:

acţiune				goto	
	a	b	$	S	A
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	3		
5			r1		
6	r2	r2	r2		

3.4. Studiul comparativ al analizoarelor sintactice

Putem în acest moment să tragem anumite concluzii relative la metoda de analiză sintactică pe care o vom alege pentru a scrie un compilator pentru un limbaj de programare. Este evident că acesta va trebui ales dintre:

- analizor sintactic LL(1), dacă dorim o analiză sintactică descendentă;

- familia analizoarelor sintactice LR(k), dacă dorim o analiză sintactică ascendentă.

Un aspect important pe care trebuie să îl luăm în considerare este legat de restricţiile impuse de metoda de analiză asupra gramaticii limbajului. Eliminarea conflictelor nu este întotdeauna uşor de realizat şi de aceea se doreşte evitarea lor. Cea mai puţin restrictivă clasă este cea a gramaticilor LR(1), dar analizorul sintactic are alte dezavantaje, asupra cărora vom reveni. Figura 3.4 ilustrează incluziunea dintre tipurile de gramatici luate în considerare în analiza sintactică. Se observă că nu există o corelaţie evidentă între gramatici LL(1) şi gramaticile LR(k), o gramatică LL(1) poate să fie LR(1), LALR, SLR sau chiar LR(0), dar orice gramatică LL(1) este LR(1).

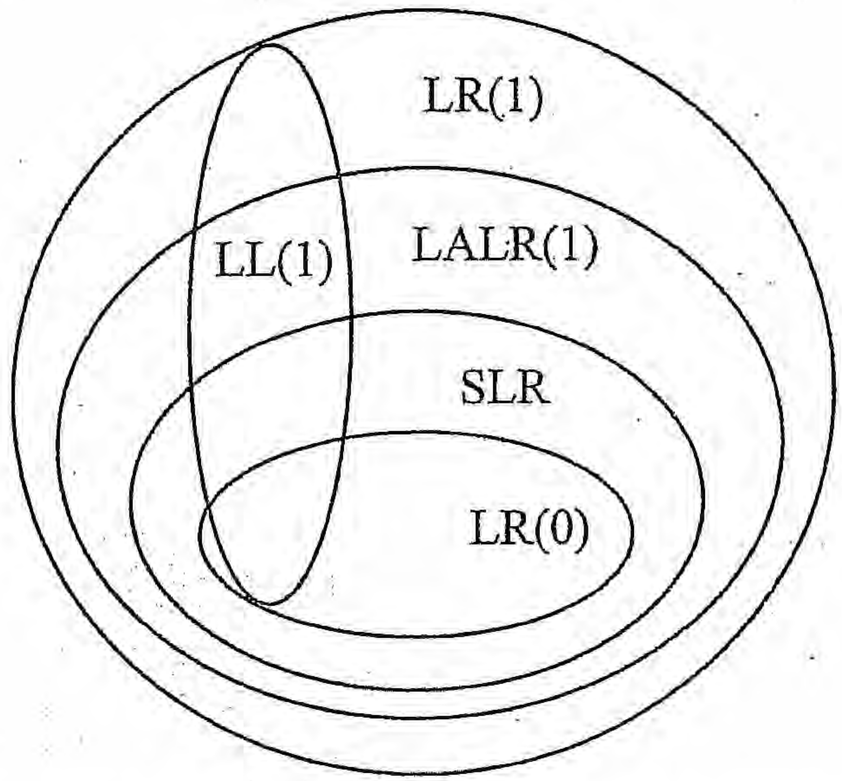

Figura 3.5: Relaţia dintre diferite clase de gramatici în funcţie de metoda de analiză sintactică

Diferenţele importante dintre cele două mari clase de analizoare sintactice se pot sintetiza în modul de construire al arborelui, tipul de derivări folosite şi semnificaţia tabelelor de analiză, după cum se observă şi în tabelul următor.

	LL	LR
construcţie arbore	descendent	ascendent
derivări	de stânga	de dreapta
tabel	producţia folosită la expandare	actiune (deplasare/reducere) si starea rezultat

Relativ la analizoarele sintactice LR(k) putem evidenţia următoarele deosebiri:

- LR(0): elementele de analiză nu iau în considerare predicţia; ca urmare, reducerea poate fi efectuată doar în stări singulare (care conţin un singur element de analiză). Se generează multe conflicte.
- SLR: foloseşte aceleaşi elemente de analiză ca şi LR(0), dar în momentul reducerii se iau în considerare anumite restricţii: se aplică producţia $A \rightarrow \beta$ pentru predicţia u doar dacă $u \in FOLLOW(A)$. Astfel se vor elimina unele cazuri de conflict care apăreau la LR(0).
- LR(1): este un algoritm performant de construire a stărilor, se generează conflicte puţine, dar prea multe stări.
- LALR: unifică stările LR(1) corespunzătoare unui aceluiaşi nucleu. Este cel mai des folosit algoritm, inclusiv de către generatoarele de analiză sintactică.

3.5. Generatoare de analiză sintactică

După cum am menţionat şi în paragraful 2.5, analiza sintactică, fiind un algoritm general valabil, poate fi implementat generic pentru mai multe tipuri de limbaje, furnizând gramatica de descriere a limbajului ca un parametru al acesteia. Celor două mari clase de analizoare sintactice, ascendent şi descendent, le corespund diferite încercări de generatoare automate bazate pe modelul analizei sintactice LL(1), respectiv LALR.

3.5.1. Generator de analiză sintactică descendentă - *LLgen*

Cel mai cunoscut şi utilizator generator din această clasă este *LLgen* (Amsterdam Compiler Kit), care are ca intrare o gramatică de specificare a unui limbaj. În cazul în care această gramatică nu este de tip LL(1), *LLgen* este capabil să aplice anumite transformări pentru a o aduce la forma cerută şi produce ca rezultat un program C care realizează analiză

sintactică descendentă pe baza algoritmului LL(1). Principalele caracteristici ale acestui generator pot fi sintetizate în:

- operatori pentru repetiţie în specificarea gramaticii limbajului;
- tratarea avansată a erorilor;
- control asupra acţiunilor semantice prin inserarea de cod C producţiilor;
- rezolvarea dinamică a conflictelor.

Exemplul 3.11. Vom considera ca exemplu un interpretor de expresii aritmetice simple, folosind gramatica: $G = (\{E, T, F\}, \{c, +, *,), (\}, P, E)$, unde producţiile sunt:

$E \rightarrow E + T | T$
$T \rightarrow T * F | F$
$F \rightarrow (E) | c$

Sintaxa fişierului de specificaţii permite folosirea parantezelor drepte ('[' şi ']') pentru repetiţii în producţii, iar unele producţii au ataşat cod C, cuprins între acolade care are rolul de acţiune semantică: în momentul în care analiza va folosi producţia respectivă se vor executa instrucţiunile asociate.

```
%token DIGIT;
        main   :  [line] +
               ;
        line   :  expr '\n'
               ;
        expr   :  term [ '+' term ] *
               ;
        term   :  factor [ '*' factor ] *
               ;
      factor   :  '(' expr ')'
               |  DIGIT
               ;
        main   :  [line] +
               ;
line{int e;}
               :  expr(&e) '\n' {printf("%d\n", e);}
```

```
                    ;
expr(int *e){int t;}
                    : term(e)
                      ['+' term(&t) {*e+ = t;}
                      ]*
                    ;
term(int *t){int f;}
                    : factor(t)
                      ['*' factor(&f) {*t* = f;}
                      ]*
                    ;
factor(int *f)
                    : '(' expr(f) ')'
                    | DIGIT {*f = yylval;}
                    ;
```

Interfaţa LLgen cu analizorul lexical este asigurată implicit astfel: se invocă *yylex*() pentru a obţine următorul atom - "valoarea" unui atom poate fi stocată în orice variabilă globală (*yylval*) de orice tip (implict fiind, *int*).

```
yylex()
{
int c;
c = getchar();
if (isdigit(c)) {
    yylval = c - '0';
    return DIGIT;
}
return c;
}
```

LLgen generează o funcţie utilizator (*parse*). Tratarea erorilor se realizează invocând o funcţie *LLmessage*(), care încearcă inserarea de atomi lipsă sau ştergerea de atomi nedoriţi.

```
%start parse, main; LLmessage(int class)
{
switch (class) {
case -1:printf("expecting EOF, ");
```



```
case 0: printf("deleting token (%d) \ n",LLsymb);break;
default: /* push back token LLsymb */
    printf("inserting token (%d)  \ n",class);
    break;
}
}
```

3.5.2. Generator de analiză sintactică ascendentă -*yacc*

Generatorul de analiză sintactică cel mai folosit este *yacc* [LMB92, LY, Joh74], apărut ca şi un utilitar al sistemului de operare Unix pe la mijlocul anilor '70 (în majoritatea cazurilor, lex şi yacc sunt folosite împreună, documentaţiile sunt comune şi există o bună compatibilitate între cele două programe). *Yacc* [7] este un generator de analiză sintactică LALR şi a fost folosit cu succes în multe proiecte de compilare şi conversie, existând diferite versiuni actualizate. Dintre aceste merită să menţinăm *bison* (produs de GNU), care remediază un neajuns al utilitarului original: *bison* generează cod standard ANSI C, spre deosebire de *yacc* care generează cod C.

Aplicând aceeaşi strategie ca şi *lex*-ul, *yacc* foloseşte un fişier de specificaţii, care are de obicei extensia *y*, cu următoarea structură:

secţiune declaraţii
%%

secţiune reguli ale gramaticii

%%

secţiune cod C

Execuţia acestui fişier de specificaţii are ca rezultat un program C cu numele *y.tab.c* care realizează analiza sintactică.

Secţiunea de declaraţii: cuprinde atât declaraţii C cuprinse între %{ şi %} cât şi declaraţii specifice *yacc*. Secţiunea de declaraţii C poate cuprinde:

- definiţii de macro-uri şi declaraţii de funcţii şi variabile care sunt folosite în acţiunile corespunzătoare regulilor din gramatică;
- "#include" pentru a folosi declaraţiile dintr-un fişier header.

[7] *Yet Another Compiler Compiler*

Secţiunea declaraţii *yacc* se construieşte după următoarele reguli:

- conţine declaraţii care definesc simbolurile terminale, neterminale;
- se specifică precedenţa operatorilor;
- toate tipurile de atomi lexicali trebuie declarate (excepţie atomii caracter ca '+' sau '*'). Simbolurile neterminale trebuie declarate dacă trebuie specificat tipul de date folosit pentru valoarea semantică.

Secţiunea de reguli ale gramaticii: conţine una sau mai multe reguli ale gramaticii corespunzătoare limbajului, implicit prima regulă din fişier specificând simbolul de start, cu următoarea formă generală:

$$
\begin{array}{lcl}
rezultat & : & regula1 - componente... \\
 & | & regula2 - componente... \\
 & & ... \\
 & ; &
\end{array}
$$

unde rezultat este neterminalul pe care această regulă îl descrie, iar componente reprezintă rescrierea neterminalul respectiv, ceea ce corespunde unor producţii $rezultat \rightarrow regula1 - componente|regula2 - componente|....$

Între componente pot fi inserate acţiuni care determină semantica regulii:

{instrucţiuni C}

şi care determină execuţia respectivelor instrucţiuni C în momentul folosirii regulii de producţie căreia îi este asociată în analiza sintactică.

Secţiunea de cod C: se copiază la sfârşitul fişierului analizorului sintactic, fiind locul cel mai convenabil pentru a insera tot ce se doreşte în analizorul sintactic, dar care nu trebuie să apară înaintea definirii lui *yyparse*.

Flexibilitatea generatorului *yacc* se datorează şi faptului că nu impune restricţii asupra gramaticii de specificare a limbajului. În cazul în care se crează un conflict, el este rezolvat automat, după următoarele principii:

- într-un conflict deplasare - reducere se va prefera întotdeauna acţiunea de deplasare (se încearcă determinarea celei mai lungi manşe posibile);
- într-un conflict reducere - reducere se va prefera prima regulă, în ordinea în care sunt scrise în secţiunea de reguli.

Exemplul 3.12. Vom considera, din nou, exemplul interpretorului de expresii aritmetice, dar, în acest caz, vom lucra cu gramatica simplificată, *yacc* permiţând să specificăm prioritatea operatorilor. Fişierul de specificaţii corespunzător gramaticii $G = (\{E\}, \{c, +, *,), (\}, E \rightarrow E + E|E * E|(E)|c, E)$ este:

```
%token DIGIT
%left '+'
%left '*'
%%
line  :  expr '\n' {printf("%d\n", $1); }
      ;
expr  :  expr '+' expr {$$ = $1 + $3; }
      |  expr '*' expr {$$ = $1 * $3; }
      |  '('expr')' {$$ = $2; }
      |  DIGIT
      ;
%%
```

Codul C corespunzător fiecărei producţii cuprinde acţiunea semantică: evaluarea expresiei rezultat $$ pe baza subexpresiilor din partea dreaptă a producţiei, referite prin $ concatenat cu indicele simbolului din partea dreaptă. Strategia LALR asigură faptul că în momentul în care se face o reducere manşa (simbolurile din partea stângă şi implicit parametrii corespunzători lor) este cunoscută şi deci valoarea parametrului corespunzător părţii drepte poate fi calculat.

Yacc asigură o bună compatibilitate cu *lex* şi în continuare cu celelalte faze ale compilării. Funcţia *yylex()* poate fi apelată de către *yacc* şi, în plus, deţine o funcţie *yyerror()* pentru a detecta erorile sintactice, care poate fi redefinită de utilizator. Pentru a folosi cele două utilitare secvenţa de comenzi este:

```
$ lex sspascal.lxi
$ yacc -d sspascal.y
$ cc lex.yy.c y.tab.c -o exec -lfl
$./exec<test1
```

unde *exec* este numele fişierului executabil care va realiza analiza lexicală şi sintactică, iar *test1* este numele fişierului care conţine programul sursă supus analizei.

4. ANALIZA SEMANTICĂ

4.1. Scopul şi instrumentele analizei semantice

Pornind de la arborele de sintaxă abstractă, analiza semantică efectuează mai multe acţiuni producând ca şi rezultat o formă intermediară care se reprezintă ca arbore atribuit sau program rezultat (în cazul translatoarelor).

Acţiunile care se execută pot fi clasificate în:

- verificarea proprietăţilor entităţilor de program;
- verificarea de tip;
- identificarea operaţiilor;
- unele transformări.

Instrumentele care pot fi folosite în această fază a procesului de compilare sunt:

- gramatici de atribute,
- metode manuale

şi constau, în principal, în anumite operaţii apelate pentru nodurile din arborele de sintaxă abstractă.

4.2. Arborele de sintaxă abstractă

Arborele de analiză sintactică este o structură de date care reprezintă modul exact în care diferite fragmente de program pot fi obţinute conform regulilor sintactice ale limbajului considerat. Forma exactă a arborelui, aşa cum este ea necesară gramaticii limbajului, nu este întotdeauna cea mai convenabilă formă pentru prelucrările ulterioare, în primul rând analiza semantică, şi de aceea se foloseşte arborele de sintaxă abstractă.

În realitate, arborele de analiză sintactică conţine prea multe detalii şi ramuri neinteresante, astfel că arborele de sintaxă abstractă este o structură mai eficientă. Diferenţa dintre cei doi arbori este de ordin practic (şi nu conceptual), iar detaliile depind de deciziile luate în proiectarea compilatorului.

Exemplul 4.1. Să considerăm expresia $b*b-4*a*c$. Figura 4.1a) conţine arborele de analiză sintactică, iar 4.1b) arborele de sintaxă abstractă corespunzător.

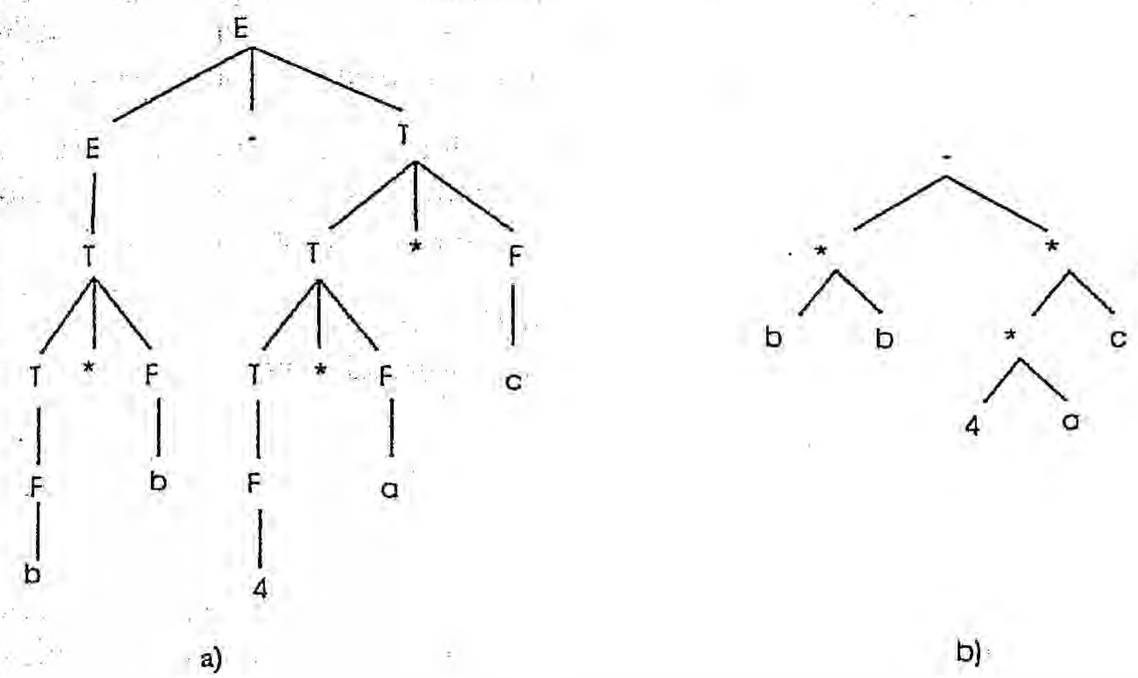

Figura 4.1: Arborele de analiză sintactică (a) şi arborele de sintaxă abstractă (b) pentru $b*b-4*a*c$

Simplificările sunt posibile bazându-ne pe următoarele observaţii [GBJL02]:

- anumite construcţii sintactice sunt folosite doar ca informaţii, nu au nici o influenţă în timpul execuţiei; de exemplu, declaraţii de variabile, declaraţii de constante;
- anumite construcţii sintactice (ca identificator, expresie, factor, termen) sunt triviale, scopul lor fiind acela de a transmite valoare şi nu e necesar să fie reţinute.

4.3. Identificarea

Faza de verificare a tipurilor testează dacă programul satisface nişte condiţii dependente de context, şi anume dacă programul foloseşte corect tipurile limbajului, dacă expresiile care apar în program sunt corecte din

punct de vedere al folosirii tipurilor [ASU86]. Sarcina verificării de tip este de a asigura că orice folosire a unui identificator are o declarație corespunzătoare a acelui identificator și de a verifica dacă contextul în care apare identificatorul este permis.

Găsirea identificatorilor este necesară verificării tipurilor. O construcție nu poate fi considerată corectă din punctul de vedere al folosirii tipurilor decât după ce tipurile componentelor au fost determinate. De exemplu, pentru ca o instrucțiune de atribuire să fie corectă, tipurile expresiei și variabilei trebuie să fie cunoscute și, mai mult, să fie echivalente. Pentru a determina tipul variabilei va trebui să știm cum a fost declarat identificatorul respectiv în program.

Aparițiile unui identificator se clasifică în: definire și folosire [1]. Apariția de definire constă în declararea identificatorului, iar toate celelalte apariții sunt considerate de folosire. Identificarea poate fi executată complet sau parțial în timpul analizei sintactice. Depinde dacă apariția de folosire a unui identificator poate fi plasată înaintea apariției de definire sau nu. Dacă sintaxa limbajului impune ca toate aparițiile de definire (de fapt, blocul de declarații), să preceadă aparițiile de folosire, cum este cazul limbajului Pascal, atunci identificarea se poate executa în timpul analizei sintactice. Altfel, cum este cazul limbajului C, care permite declarații în interiorul programului, printre instrucțiuni, identificarea completă este posibilă numai la pasul următor. În ambele cazuri, în momentul procesării apariției de definire este necesară memorarea informațiilor legate de tip, care se face prin completarea pentru identificator a câmpului asociat tipului, din tabela de simboluri.

Putem spune că procesul de identificare constă din două rutine:

- procesarea aparițiilor de definire a identificatorilor - care va trebui să completeze în tabela de simboluri câmpurile corespunzătoare tipului (cu tipul determinat din declarație) și valorii (cu valoarea inițială implicită sau cea impusă de declarație);

- procesarea aparițiilor de folosire a identificatorilor - care va verifica dacă tabela de simboluri conține informațiile necesare (de tip și valoare) unei apariții de folosire, actualizând eventual unele informații din tabelă.

[1] În unele limbaje mai vechi se considera existența a trei tipuri de apariții: declarare, definire și folosire. Datorită faptului că limbajele moderne realizează la momentul declarării și inițializarea variabilei la o valoare implicită (0 pentru întregi, etc.) cele două faze de declarare și definire au fuzionat.

Construirea tipurilor

Limbajele de programare divid tipurile în două categorii principale:

- primitive: întreg (int, integer), real (real, float), caracter (char), boolean și

- tipuri structurate - construite folosind constructorii de tip: șir (array), structură (record, struct), referință, funcție etc.

Aceste construcții de tipuri trebuie stocate într-un anumit mod, pentru a putea furniza această informație procesărilor ulterioare. Se pot construi arbori sau grafe aciclice care conțin ca frunze tipurile primitive. De exemplu, declarația de tip:

type s = array[1..10,1..10] of integer;

va fi reprezentată ca în figura 4.2

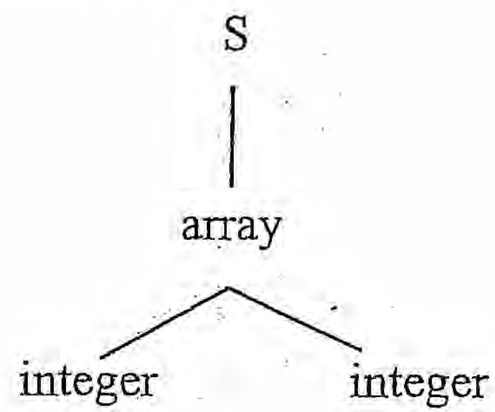

Figura 4.2: Reprezentarea unei construcții de tip

Merită să menționăm metoda folosită de compilatorul C, dezvoltată de D.M. Ritchie [KR88]: sunt permiși doar trei constructori de tip - șir, referință și funcție:

- *array(t)* - notează un șir de elemente de tipul t (lungimea fiind nedefinită);

- *pointer(t)* - notează pointerul spre o valoare de tipul t;

- *freturns(t)* - notează o funcție care returnează o valoare de tipul t.

Fiecare din aceste tipuri poate fi reprezentat de un vector de biți. Deoarece avem doar 3 constructori, o codificare pe 2 biți este suficientă:

pointer 01
array 10
freturns 11

Tipurile primitive sunt codificate pe 4 biţi:

boolean 0000
char 0001
integer 0010
real 0011

Astfel orice tip structurat poate fi descris prin concatenearea acestor codificări, ca în următoarele exemple:

integer 000000 0010
array (integer) 000010 0010
array (array (integer)) 001010 0010
freturns (array (array (integer))) 111010 0010

4.4. Verificarea tipurilor

Pentru a asigura că un program nu conţine erori de tipizare, compilatorul trebuie să atribuie tipuri şi apoi să le verifice. Atribuirea de tip constă în a determina tipul unei construcţii din program (variabilă, expresie, funcţie), pentru ca apoi verificarea să determine dacă programul foloseşte corect aceste atribuiri de tip.

Verificarea tipurilor poate fi clasificată în două componente mari: *verificare statică*, ce se efectuează la momentul compilării şi *verificare dinamică*, ce se efectuează în timpul execuţiei programului. În principiu, verificarea tipurilor se poate executa întotdeauna în momentul execuţiei, dacă informaţia relativă la tipuri este accesibilă în codul executabil. Verificarea dinamică de tip măreşte timpul de execuţie al programului obiect şi micşorează fiabilitatea codului compilat. Un limbaj de programare se numeşte *limbaj puternic tipizat* sau limbaj cu verificarea statică de tip dacă, pentru orice expresie, tipul ei poate fi determinat la momentul compilării, asigurându-se astfel că nu apar erori de tipizare în programul obiect. Limbajul Pascal este un astfel de exemplu, chiar dacă unele verificări se pot efectua doar dinamic. De exemplu, pentru următoarele declaraţii:

sir:array[1..10] of integer;

i:integer;

un compilator nu poate garanta că argumentul i din expresia $sir[i]$ este în limitele impuse de definiţie, adică între 1 şi 10. Astfel de verificări pot fi efectuate doar prin tehnici speciale, de tipul analizei fluxului de date şi nu întotdeauna cu succes; de aceea, practic ele sunt amânate pentru momentul execuţiei.

Echivalenţa tipurilor

O parte importantă a verificării de tip este echivalenţa tipurilor. Majoritatea limbajelor de programare nu definesc noţiunea de tipuri identice, ci de echivalenţă a lor, care poate fi privită din două puncte de vedere: *echivalenţa struturală* şi *echivalenţa de nume*. Diferite limbaje adoptă una sau alta dintre perspective: echivalenţa structurală este implementată în limbajele Algol68, Modula3, C şi C++ cu excepţia tipului *struct*, în timp ce limbajul Ada şi unele implementări ale limbajului Pascal folosesc echivalenţa de nume.

Diferenţa dintre cele două abordări se poate observa în efectul următoarelor declaraţii:

```
type sir= array[1..10] of integer;
var a:sir;
    p,q: array[1..10] of integer;
    r: array[1..10] of integer;
```

Într-o implementare a limbajului Pascal bazată pe echivalenţa de nume, variabilele p, q, r sunt de acelaşi tip, dar variabilele p şi a au tipuri diferite. Dacă implementarea este bazată pe echivalenţa structurală atunci toate variabilele declarate sunt considerate de acelaşi tip.

Echivalenţa de nume se caracterizează prin faptul că fiecare nume asociat unui aceluiaşi tip este tratat ca un tip unic, astfel că două nume de tipuri sunt echivalente dacă şi numai dacă au nume identic.

Două tipuri se numesc *echivalente structural* dacă şi numai dacă sunt acelaşi tip primitiv sau au fost construite aplicând acelaşi constructor unor tipuri echivalente structural. Verificarea se poate face folosind un algoritm de tipul algoritmului 4.1 [ASU86], în care am considerat că modalităţile de construire a tipurilor constau în:

- şir: *array(s,t) = array s of t*,
- referinţă: *^s* şi
- funcţie: *func(s,t)*;

dar care poate fi uşor extins pentru a include şi alte structuri.

Algoritmul 4.1 Echiv_struct(s,t)

```
INPUT: s,t - tipuri;
OUTPUT: true - dacă s şi t sunt echivalente structural, altfel false
if s,t - primitive and s=t then
  return true
else
  if s = array(s1, s2) and t = array(t1, t2) then
    return Echiv_struct(s1, t1) and Echiv_struct(s2, t2)
  else
    if s = ^s1 and t = ^t1 then
      return Echiv_struct(s1, t1)
    else
      if s = func(s1, s2) and t = func(t1, t2) then
        return Echiv_struct(s1, t1) and Echiv_struct(s2, t2)
      else
        return false
      end if
    end if
  end if
end if
```

Conversii de tip

Să considerăm următoarele declaraţii în limbajul Pascal:

var i:integer; r:real;

Expresia $r + i$ este corectă şi tipul ei este real. Compilatorul asigură acest rezultat pe baza unor conversii de tip implicite care se efectuează. De fapt, reprezentările numerelor întregi şi reale sunt diferite, operatorii care lucrează cu valori întregi, respectiv reale sunt implementaţi diferit (chiar dacă se foloseşte o aceeaşi notaţie) şi, de fapt, nu se poate aplica operaţia unor operatori de tipuri diferite. Astfel, compilatorul trebuie să convertească un operand la tipul celuilalt sau ambii operatori la tipul corespunzător (de exemplu, în instrucţiunea $r := i + i$).

O conversie se numeşte *implicită* dacă este inserată automat de către compilator. Definirea limbajului specifică conversiile implicite permise de limbaj: de exemplu, de la întreg la real, de la caracter la string. Principiul care trebuie respectat în aceste conversii este ca informaţia să nu fie pierdută. Conversia unui întreg la real respectă această condiţie, pe când conversia inversă de la real la întreg duce la pierderea informaţiei (valoarea părţii fracţionare) şi poate genera situaţii eronate.

O conversie se numeşte *explicită* dacă este specificată explicit de programator prin intermediul unui apel al unei funcţii definite între tipuri. Limbajele de programare posedă funcţii predefinte pentru realizarea acestor conversii (de exemplu, limbajul Pascal defineşte funcţiile *trunc* şi *round* pentru conversia unui întreg la un real).

4.5. Gramatici de atribute

Introduse de D. Knuth [Knu68] ca o metodă de specificare a semanticii limbajelor de programare, au ca principiu de bază ideea de a atribui o semnificaţie asociind atribute simbolurilor din arborele de analiză sintactică. Gramaticile de atribute reprezintă o modalitate eficientă de specificare a calculelor şi a dependenţelor dintre aceste calcule dintr-un arbore de sintaxă abstractă, stabilind ordinea corectă de execuţie a lor.

Gramaticile independente de context sunt extinse cu două caracteristici: fiecărui simbol $X \in N \cup \Sigma$ i se asociază atribute şi fiecărei producţii i se asociază reguli de calcul a atributelor. Se defineşte astfel informaţia din arborele de sintaxă abstractă necesară analizei semantice.

Definiţia 4.1. *[Knu68, Knu71]$GA = (G, A, R)$ se numeşte gramatică de atribute (sau gramatică atributivă) dacă şi numai dacă:*

- *$G = (N, \Sigma, P, S)$ este o gramatică independentă de context;*
- *$A = \bigcup_{X \in N \cup \Sigma} A(X)$ este o mulţime finită de atribute: pentru fiecare simbol al gramaticii se definesc zero sau mai multe atribute, care sunt folosite pentru a memora informaţii semantice ataşate nodurilor din arborele de sintaxă abstractă; toate nodurile corespunzătoare unui acelaşi simbol vor avea aceleaşi atribute, cu valori diferite;*
- *$R = \bigcup_{p \in P} R(p)$ este o mulţime finită de reguli, numite reguli de evaluare a atributelor; pentru fiecare producţie $A \to X_1 ... X_n$ se asociază o mulţime de reguli de evaluare a atributelor, conţinând expresii ale valorilor atributelor asociate simbolurilor A şi $X_i, i = \overline{1,n}$.*

Atributele asociate unui simbol se clasifică în:

- *atribut sintetizat* - dacă regula de evaluare pentru producţia $A \to X_1 ... X_n$ atribuie valoare atributului lui A şi
- *atribut moştenit* - dacă regula de evaluare corespunzătoare producţiei $A \to X_1 ... X_n$ atribuie valoare atributelor lui $X_i, i = \overline{1,n}$ (valoarea atributului este moştenită de la părinţi).

O gramatică de atribute este *completă* dacă pentru orice simbol $X \in N$: 1) pentru orice producţie $X \rightarrow \alpha$ a gramaticii, mulţimea R(p) conţine o regulă de evaluare a atributelor sintetizate ale lui X şi 2) pentru orice producţie $Y \rightarrow \alpha X \beta$, mulţimea R(p) conţine o regulă de evaluare a atributelor moştenite ale lui X.

Presupunând că o anumită construcţie a unui limbaj a fost specificată prin reguli de producţie, ataşarea atributelor nu este o sarcină uşoară. Atributele trebuie astfel alese încât să modeleze exact caracteristica a cărei evoluţie se doreşte a fi urmărită în program. Exemplul 4.2 pleacă de la o gramatică pentru generarea de numere binare $G = (\{B, N\}, \{0, 1\}, P, N)$:
$N \rightarrow NB|B$
$B \rightarrow 0|1$

Acestei gramatici dorim să-i ataşăm atribute care să ne furnizeze valoarea decimală a numerelor binare generate. Pentru un număr binar $b_n ... b_1 b_0$ formula de calcul este:
$v = b_n \cdot 2^n + \cdots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Vom avea nevoie de un atribut v pentru a reţine valoarea. Aplicând schema lui Horner în acest calcul[2], putem deduce şi regulile de calcul a atributelor.

Exemplul 4.2.

$$\begin{array}{ll} N_1 \rightarrow N_2 B & \{N_1.v = 2 \cdot N_2.v + B.v\} \\ N \rightarrow B & \{N.v = B.v\} \\ B \rightarrow 0 & \{B.v = 0\} \\ B \rightarrow 1 & \{B.v = 1\} \end{array}$$

(În producţia $N \rightarrow NB$ neterminalul N apare de două ori. Pentru a nu crea confuzii relativ la valorile atributelor unui aceluiaşi simbol, acestea se indexează pentru a putea fi referite diferit în cadrul regulilor.)

Numărului binar 101 îi corespunde arborele din figura 4.3 iar valoarea lui zecimală este valoarea atributului v corespunzătoare rădăcinii (v=5).

4.5.1. Evaluator de atribute

Sarcina analizei semantice poate fi asimilată unui evaluator de atribute care: *aplică o strategie de traversare* a arborelui de sintaxă abstractă, *execută apeluri de calcul* astfel încât dependenţele specificate (relative la tip şi operaţie) sunt îndeplinite şi *propagă valorile calculate* în arbore.

[2] $n = ((b_n \cdot 2 + b_{n-1}) \cdot 2 + \cdots + b_1) \cdot 2 + b_0$

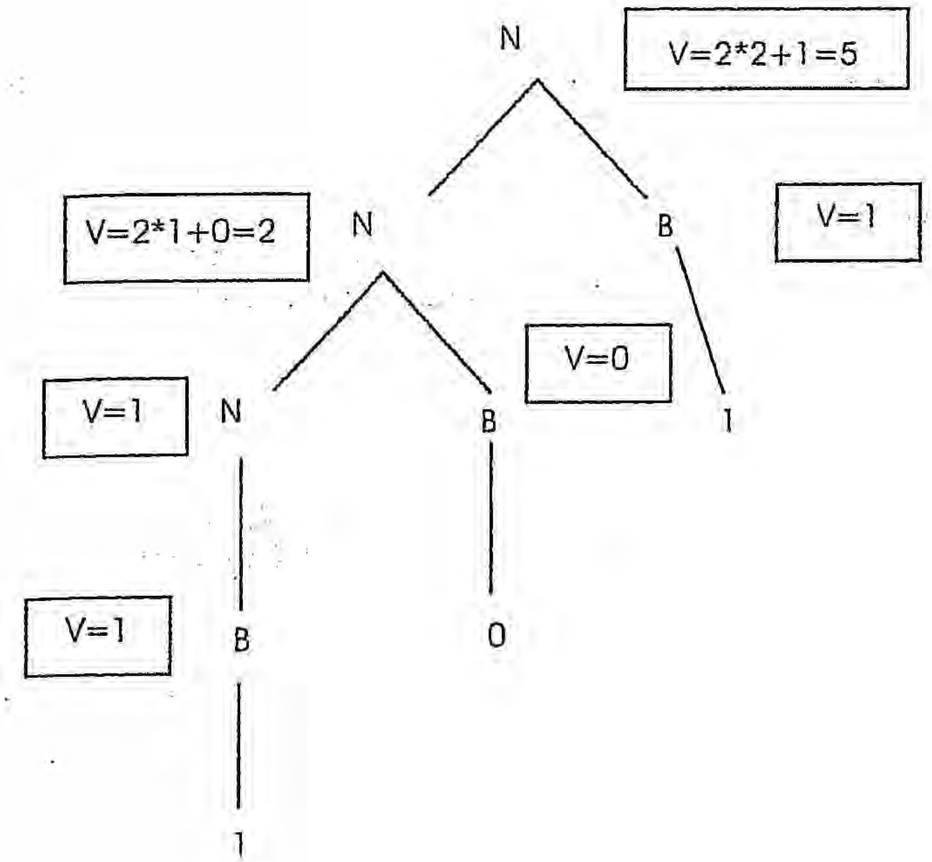

Figura 4.3: Calculul valorii zecimale a unui număr binar folosind gramatici de atribute

Construirea unui evaluator de atribute se face respectându-se următoarele principii [Eng84]:

- să aloce spaţiu pentru atributele corespunzătoare fiecărui nod din arborele de sintaxă abstractă;
- să se poată aplica pentru orice arbore care respectă sintaxa abstractă specificată de regulile gramaticii de atribute considerate;
- să execute o traversare a arborelui însoţită de calcularea atributelor în momentul vizitării unui nod;
- să respecte ordinea de execuţie a calculelor: nici o valoare de atribut să nu fie folosită înainte de a fi disponibilă, orice atribut primeşte valoare o singură dată şi se ţine cont de dependenţele dintre atribute;
- să detecteze cazul în care nu se pot calcula toate atributele (evitarea ciclurilor infinite de calcul).

O implementare naivă ar proceda astfel: vizitează pe rând toate nodurile, în mod repetat, la fiecare vizitare execută regulile de evaluare astfel încât valorile de atribute folosite sunt disponibile, iar valorile calculate la acest pas nu erau disponibile. Un astfel de evaluator de atribute se numeşte evaluator dinamic, dar reprezintă o metodă ineficientă şi, în plus, nu se termină dacă există un ciclu între dependenţele de atribute.

Un evaluator static determină ordinea de evaluare la momentul compilării, este mai eficient şi detectează ciclurile între dependenţe. Ordinea de evaluare se bazează pe determinarea dependenţelor de atribute: de la atribute moştenite la atribute sintetizate şi de la atribute sintetizate spre cele moştenite.

Un alt tip de evaluator cu rezultate eficiente, *evaluatorul multi-vizită* [Eng84], se bazează pe vizitarea fiecărui nod neterminal din arbore de una sau mai multe ori. Deoarece un atribut nu trebuie calculat înainte de a fi necesar, evaluarea atributelor se face doar la terminarea vizitării unui nod pentru părinte sau descendent. În principiu, există două tipuri de vizitare pentru un subarbore corespunzător unei producţii $A \rightarrow X_1 ... X_n$:

- pune la dispoziţie valori ale atributelor moştenite pentru un descendent X_i
- vizitează descendentul X_i
- calculează atributele sintetizate care depind de X_i

şi

- pune la dispoziţie valori ale atributelor sintetizate pentru părintele A
- vizitează părintele A
- calculează atributele moştenite care depind de A.

La fiecare a i-a vizitare se compară dacă sunt valori noi de atribute calculate şi pe baza dependenţelor dintre atribute se decide dacă s-a încheiat evaluarea tuturor atributelor sau dacă, datorită unui ciclu între dependenţe, procesul nu se va termina.

Nici una dintre aceste metode de evaluare nu reprezintă o soluţie acceptabilă, ceea ce a condus la a încerca găsirea unor clase speciale de gramatici de atribute particulare, care să permită evaluatori performanţi.

4.5.2. Clase speciale de gramatici de atribute

Gramatici L-atributate

Gramaticile L-atributate[3] permit evaluarea atributelor într-o singură traversare de la stânga spre dreapta a arborelui de sintaxă abstractă. Ca rezultat, evaluarea atributelor într-o gramatică L-atributată poate fi uşor incorporată într-un analizor sintactic descendent. În plus, majoritatea gramaticilor corespunzătoare limbajelor de programare îndeplinesc condiţia de L-atributate.

Definiţia 4.2. *[ASU86] $GA = (G, A, R)$ se numeşte gramatică L-atributată dacă şi numai dacă GA este o gramatică de atribute şi pentru orice producţie $A \rightarrow X_1 X_2 ... X_i ... X_n$ regulile de calcul a atributelor respectă următoarele condiţii:*

- *orice atribut moştenit al lui X_i depinde doar de atribute moştenite ale lui A şi atribute ale simbolurilor $X_1 ... X_{i-1}$ (noduri situate la stânga în arbore);*
- *orice atribut sintetizat al lui A nu depinde de alte atribute sintetizate ale lui A.*

Exemplul 4.3. Un exemplu de gramatică L-atributată se poate asocia declaraţiei de variabile din limbajul C:

$$
\begin{array}{rcl}
Decl \rightarrow & DeclTip\ ListId & \{ListId.tip = DeclTip.tip\} \\
ListId \rightarrow & Id & \{Id.tip = ListId.tip\} \\
ListId_1 \rightarrow & ListId_2, Id & \{ListId_2.tip = ListId_1.tip; \\
 & & Id.tip = ListId_1.tip\}
\end{array}
$$

Declaraţiei *int* i, j îi corespunde arborele din figura 4.4. Săgeata punctată indică ordinea de evaluarea a atributului *tip* care în final va actualiza câmpurile destinate tipului în tabela de simboluri pentru identificatorii i şi j.

Atributul *tip* este un atribut moştenit şi este transmis pentru a completa informaţia legată de tipul unui identificator în tabela de simboluri.

Să considerăm subarborele din figura 4.5. Un evaluator de atribute care calculează atributele nodului C_2 va lua în considerare doar două mulţimi de atribute [GBJL02]:

[3] L-attributed (left-attributed) grammars, în engleză

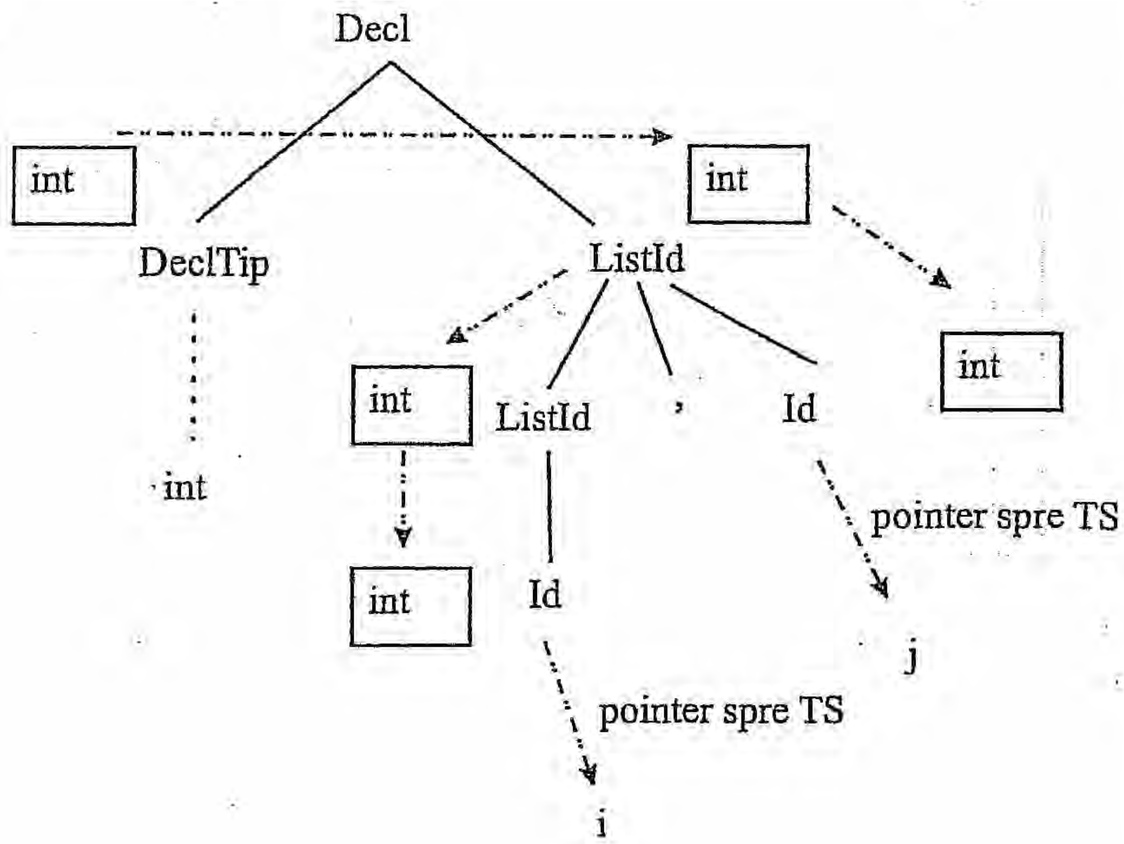

Figura 4.4: Gramatică L-atribuită pentru determinarea tipului identificatorilor

1. atributele nodurilor aflate pe drumul de la rădăcină la C_2 şi anume A, B_2, C_2;

2. atributele sintetizate ale nodurilor situate la stânga de nodul curent C_1, B_1, B_2.

Chiar dacă unul dintre nodurile C_1, B_1, B_2 ar avea descendenţi, aceştia nu intervin în calculul atributelor lui C_2, pentru că toate calculele asociate nodurilor respective au fost deja efectuate. De asemenea, atributele lui C_2 nu depind de atributele nodurilor B_4 şi C_3, conform definiţiei gramaticilor L-atribuite.

În acest moment, prelucrarea nodurilor aflate la stânga lui C_2 este terminată şi se prelucrează nodurile C_2, B_3, A, reţinând valori ale atributelor în ordine de la stânga spre dreapta, după ce analiza sintactică a creat nodul respectiv şi evaluatorul a calculat valoarea atributelor corespunzătoare nodului. Astfel, nu este necesar să construim întreg arborele de analiză, ci ne putem restrânge la nodurile ce se află pe drumul de la rădăcină la nodul curent. Tot ce se află la stânga a fost procesat şi calculat şi tot ce se află

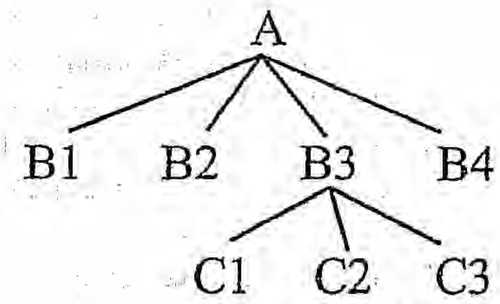

Figura 4.5: Subarbore dintr-o gramatică L-atribuită

la dreapta nu are importanţă. Putem concluziona că evaluarea atributelor se poate intercala într-un analizor sintactic descendent (LL(1)), direct, ca o procedură ce se apelează la crearea unui nou nod.

Figura 4.6 ilustrează ordinea în care se realizează calcularea atributelor într-un nod din arbore.

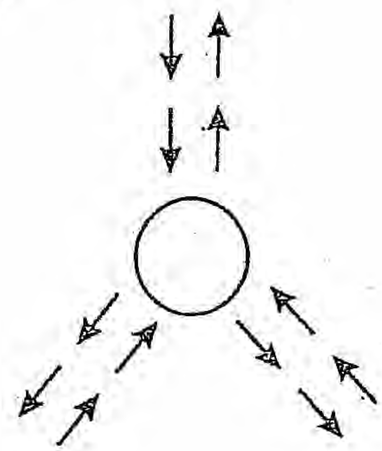

Figura 4.6: Calculul atributelor într-un nod pentru o gramatică L-atribuită

Pentru analizoare sintactice ascendente situaţia este îngreunată de atributele moştenite care trebuie transmise de la părinte spre descendenţi. Pentru a rezolva această problemă se realizează un artificiu de calcul, care este implementat în *yacc*, bazat pe analizorul sintactic LALR(1) şi se bazează pe următoarea idee [Joh74]: analizoarele sintactice LR(k) folosesc o stivă de lucru care conţine terminale (ce au fost deplasate) şi neterminale (obţinute în urma unei reduceri). În paralel cu această stivă se consideră o stivă adiţională, numită *stivă de atribute*, care va conţine atributele fiecărui simbol din stiva de lucru, păstrând ordinea. Problema importantă

este de a completa atributele moştenite: când suntem la sfârşitul unei părţi drepte de producţie, se face o reducere şi se poate completa valoarea atributului, când suntem în interiorul unei părţi drepte de producţie, se efectuează următorul artificiu:

Să presupunem următoarea situaţie: $A \rightarrow B\{C.c = B.b\}C$, unde c este un atribut moştenit al lui C şi b e un atribut sintetizat al lui B. Vom introduce o ϵ-producţie astfel:

$A \rightarrow BA'C$

$A' \rightarrow \epsilon\{C.c = B.b\}$

Yacc face acest artificiu automat şi memorează contextul lui A' astfel încât B.b şi C.c pot fi identificate chiar dacă au fost scoase din contextul lor obişnuit.

Gramatici S-atributate

Gramaticile S-atributate sunt o clasă specială de gramatici de atribute, caracterizate prin faptul că nu conţin atribute moştenite. Toate atributele definite sunt sintetizate, de unde se explică şi denumirea acestei clase. Evaluarea atributelor într-o gramatică S-atributată poate fi implementată uşor atât în analiza sintactică ascendentă, cât şi în cea descendentă. Figura 4.7 ilustrează ordinea în care se realizează calcularea atributelor într-un nod din arbore, pentru o gramatică S-atributată.

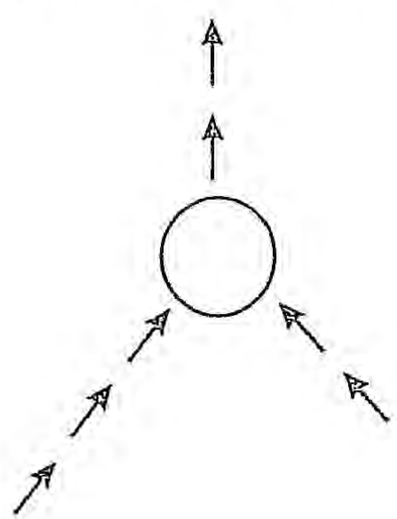

Figura 4.7: Calculul atributelor într-un nod pentru o gramatică S-atributată

Exemplul 4.4. Putem asocia o gramatică S-atributată listei de declaraţii din limbajul C, în care definim un atribut *dim* pentru a determina spaţiul necesar la alocarea variabilelor respective.

$ListDecl_1 \rightarrow$	$ListDecl_2; Decl$	$\{ListDecl_1.dim = ListDecl_2.dim + Decl.dim\}$
$ListDecl \rightarrow$	$Decl$	$\{ListDecl.dim = Decl.dim\}$
$Decl \rightarrow$	$Tip\ ListId$	$\{Decl.dim = Tip.dim \times ListId.Nr\}$
$Tip \rightarrow$	int	$\{Tip.dim = 4\}$
$Tip \rightarrow$	$long$	$\{Tip.dim = 8\}$
$ListId \rightarrow$	Id	$\{ListId.Nr = 1\}$
$ListId_1 \rightarrow$	$ListId_2, Id$	$\{ListId_1.Nr = ListId_2.Nr + 1\}$

Considerând declaraţia *int i; long a;* obţinem arborele din figura 4.8, în care săgeţile întrerupte indică ordinea de evaluare a atributelor.

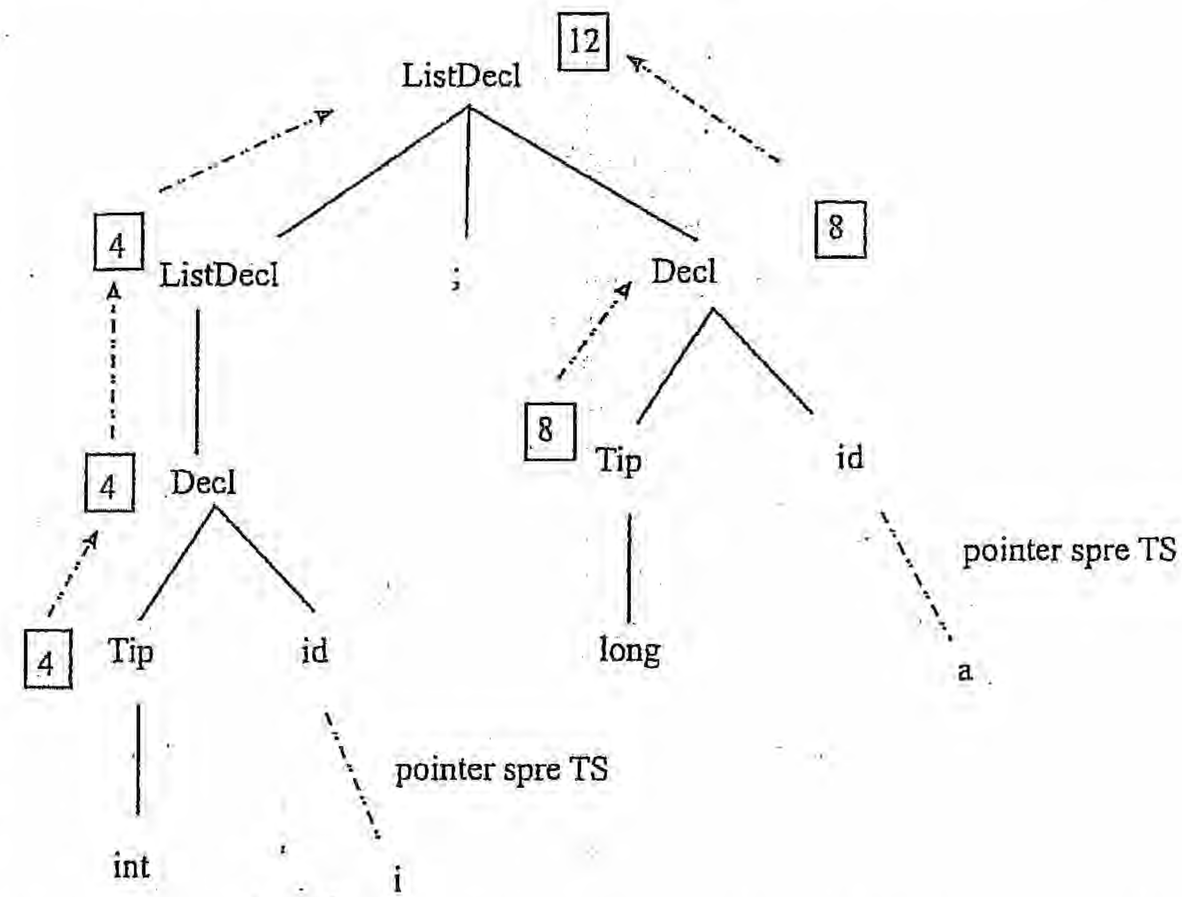

Figura 4.8: Gramatică S-atributată pentru calculul dimensiunii zonei de memorie la alocarea variabilelor

4.6. Metode manuale

Deşi gramaticile de atribute reprezintă un instrument puternic pentru a realiza analiza semantică, detaliile legate de implementarea lor au dus la căutarea unor tehnici mai eficiente. Caracteristicile programelor impun uneori asocierea multor atribute, ceea ce duce la o ocupare mare de spaţiu şi la încetinirea procesului de compilare. De asemenea, discuţia despre evaluatori de atribute a evidenţiat că implementarea lor este dependentă de modul de definire al atributelor. Dacă mai adăugăm faptul că definirea atributelor şi a regulilor de calcul asociate este realizată manual, putem argumenta că merită să aruncăm o privire şi asupra altor metode, care chiar dacă nu sunt atât de eficiente, beneficiază de simplitate.

Aceste metode poartă numele de ***metode manuale*** şi sunt în număr de două: *interpretare simbolică* şi *ecuaţii de flux de date*. Ambele pornesc de la arborele de sintaxă abstractă, eventual parţial atributat, dar necesită informaţii relative la trasferul controlului prin arbore, mai exact pentru un anumit nod care este succesorul sau succesorii în fluxul de control. Pentru aceasta vom avea nevoie de *graful fluxului de control*.

4.6.1. Graful fluxului de control

Noţiunea de graf al fluxului de control este întâlnită în majoritatea compilatoarelor moderne. Graful este de obicei asociat codului intermediar astfel: nodurile reprezintă blocuri de bază ale programului, iar arcele indică modul în care se transferă controlul de la un bloc la altul. Proprietăţile pe care graful obţinut le are pot fi apoi folosite cu succes în optimizarea codului şi generarea de cod obiect. Există însă şi posibilitatea de a construi "mai repede" acest graf, şi anume plecând de la arborele de sintaxă abstractă. Cele două abordări au totuşi un scop comun: extragerea de informaţii la momentul compilării despre ce se va întâmpla la execuţie. În continuare vom considera graful fluxului de control asociat arborelui de sintaxă abstractă, urmând să revenim asupra celeilalte abordări în capitolul următor.

Definiţia 4.3. *[ASU86] **Graful fluxului de control** asociat arborelui de sintaxă abstractă este un graf orientat în care nodurile sunt reprezentate de nodurile arborelui, iar arcele reprezintă succesiunea nodurilor în fluxul de control.*

Graful fluxului de control se construieşte prin fragmentarea arborelui de sintaxă abstractă folosind o rutină de fragmentare pentru fiecare nod, specializată în funcţie de tipul nodului respectiv. Rutina corespunzătoare

unui nod N primeşte ca parametru un pointer spre nodul ce urmează a fi procesat, determină ce producţie a lui N descrie nodul şi apelează rutinele de fragmentare a descendenţilor săi într-o traversare recursivă a arborelui. Toate rutinele actualizează o variabilă globală *ultim_nod*, un pointer spre ultimul nod procesat pe drumul fluxului de control, prin intermediul unei variabile *succesor* ataşată fiecărui nod, reprezentând un pointer spre următorul nod ce urmează a fi procesat. Astfel se obţine o parcurgere a grafului în ordinea în care se va deplasa şi fluxul de control în execuţia programului.

De exemplu, pentru o expresie binară fragmentarea ei înseamnă fragmentarea subexpresiei stângi şi a subexpresiei drepte şi apoi actualizarea variabilei *ultim_nod*, după cum se observă şi în figura 4.9.

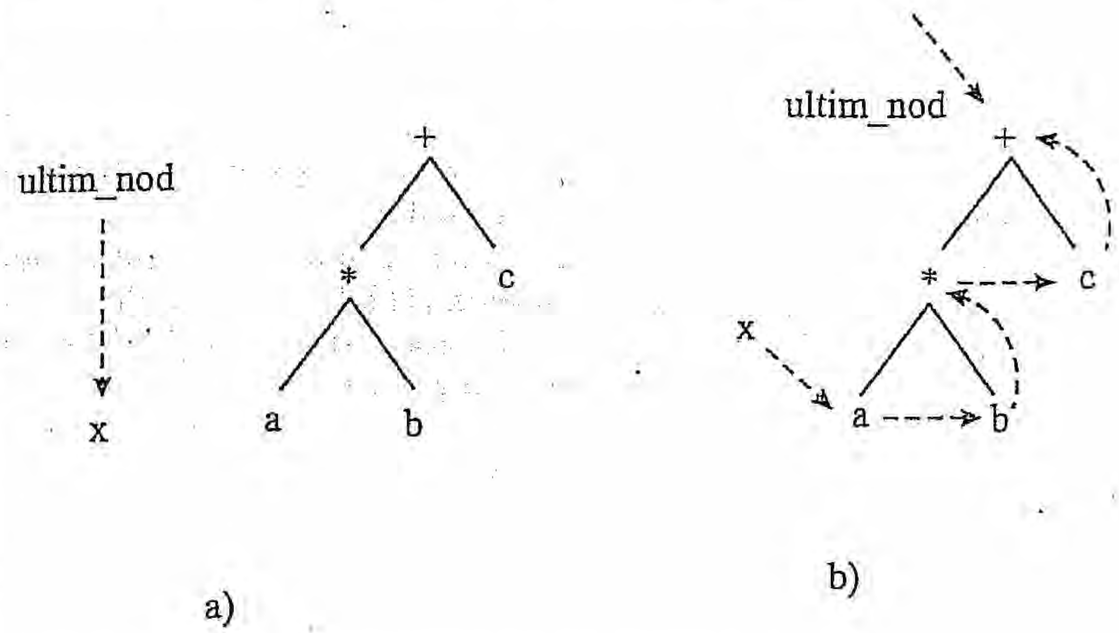

Figura 4.9: a) Arborele corespunzător expresiei $a * b + c$
b) Graful fluxului de control ataşat arborelui

Totuşi, nu pentru toate construcţiile din arborele de sintaxă abstractă lucrurile sunt aşa simple. Pentru o instrucţiune condiţională, subarborii corespunzători celor două ramuri au puncte diferite de terminare, care vor genera un conflict la actualizarea variabilei *ultim_nod*. În plus, nodul corespunzător deciziei va avea doi descendenţi şi nu unul ca până acum. Această situaţie poate fi uşor rezolvată prin stocarea unei liste de succesori în locul unei singure variabile *succesor*. Vom construi un nod special de unificare a celor două ramuri, nod care va fi inserat în graful fluxului de control fără a face parte însă din arbore, după cum se observă în figura 4.10.

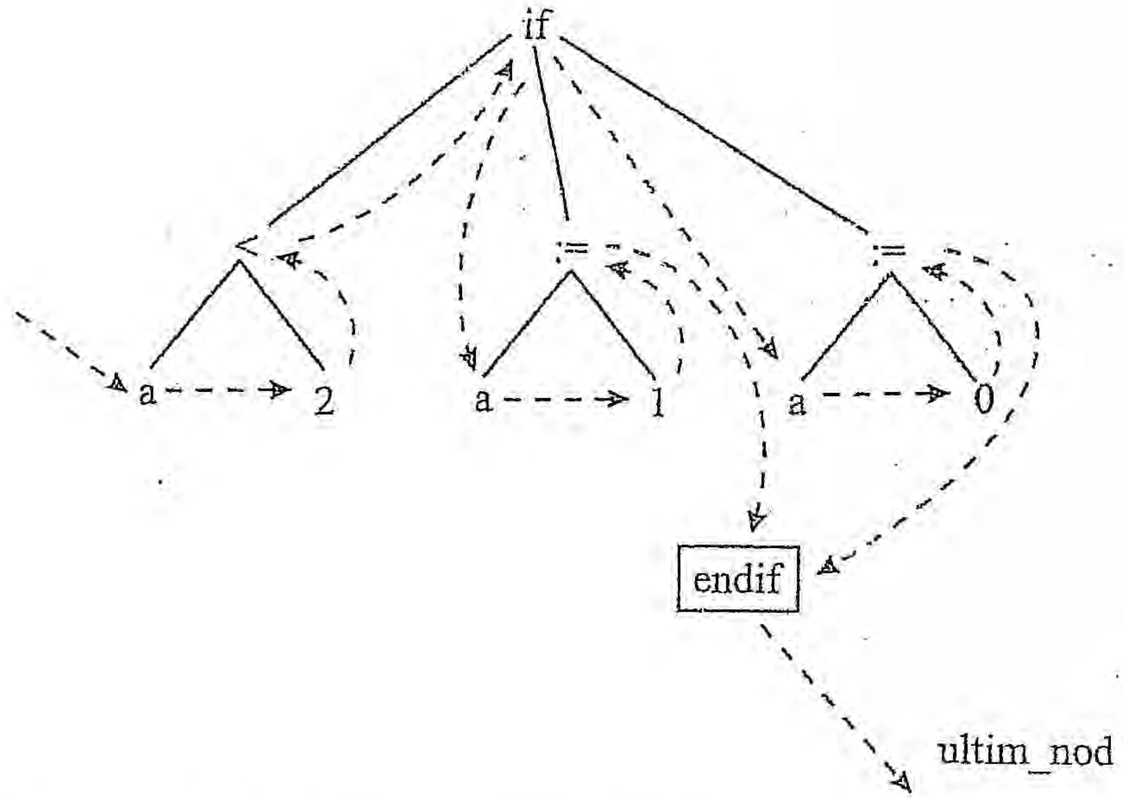

Figura 4.10: Graful fluxului de control pentru
if $a < 2$ *then* $a := 1$ *else* $a := 0$

4.6.2. Interpretare simbolică

În momentul execuţiei unui program fluxul de control parcurge un drum posibil prin graf. Codul care se execută în fiecare nod este determinat de valorile variabilelor la intrarea în fragmentul respectiv şi determină la rândul său valorile variabilelor la ieşire. Unele informaţii contextuale relativ la variabile pot fi deduse static prin interpretare simbolică.

Interpretarea simbolică [Sam76] sau simularea pe stivă reprezintă simularea execuţiei la momentul compilării, ataşând o stivă fiecărui arc din graful fluxului de control. Această reprezentare statică a stivei execuţiei conţine câte o intrare pentru fiecare identificator (în principal, variabile şi constante) vizibil în acel punct de program, intrare care stochează toate informaţiile disponibile la momentul compilării despre variabila sau constanta respectivă în momentul în care fluxul de control trece prin arcul considerat.

Exemplul 4.5. Vom considera graful deja construit din figura 4.10. Presupunem că în momentul în care controlul este transferat acestei instrucţiuni condiţionale configuraţia stivei indică faptul că variabila a are va-

loarea 5. Interpretarea simbolică asociată se poate urmări în figura 4.11.

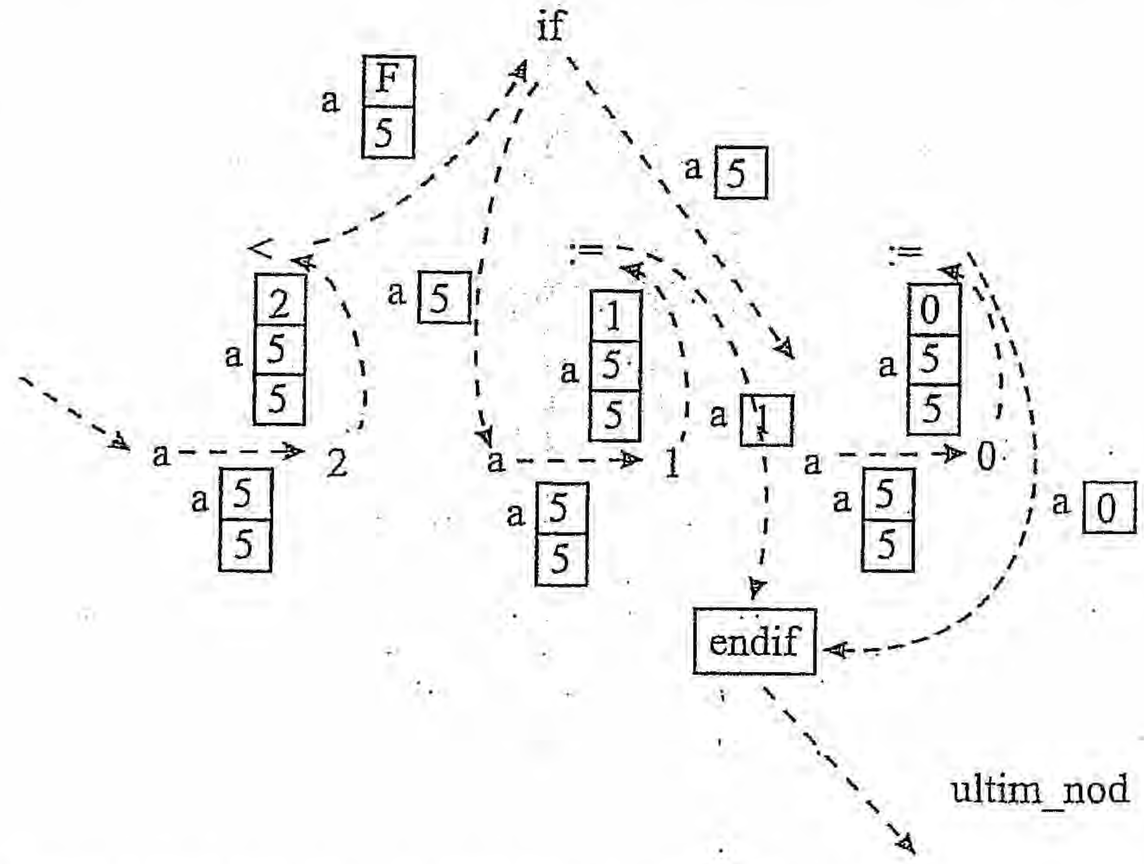

Figura 4.11: Interpretare simbolică pentru *if* $a < 2$ *then* $a := 1$ *else* $a := 0$

Fluxul intră prima dată în nodul a şi după ieşirea din acest nod valoarea lui a se află în vârful stivei. Apoi intră în nodul 2, pune şi această valoare în stivă şi ajunge în nodul $<$. Ajungând la un nod care conţine un operator, scoate din stivă ultimele două valori (care se compară) şi va pune în stivă rezultatul (F - fals). Fluxul intră în nodul *if* şi valoarea din vârful stivei este folosită pentru a decide pe care ramură se va merge în continuare.

Partea *then* ar putea fi marcată ca fiind "cod mort". Totuşi este mai practic să parcurgem ambele ramuri începând cu aceeaşi reprezentare a stivei şi să le verificăm. Un motiv în acest sens îl constituie faptul că un compilator emite erori de compilare pentru tot codul, chiar şi pentru cel care nu este atins în execuţie.

Dacă considerăm o altă instrucţiune, similară cu prima, şi anume *if* $a < 2$ *then* $b := 1$ *else* $b := 0$ şi la intrare ştim doar că variabila a a fost iniţializată, neavând nici o informaţie despre b, parcurgerea drumului făcut de flux în arbore, pe ambele ramuri, va avea ca rezultat o valoare pentru b (este iniţializat).

Interpretarea simbolică a fost folosită încă din anii '60 (în 1965, Naur [Nau65] a folosit-o pentru verificarea tipurilor în ALGOL60) şi poate fi aplicată cu succes atât în analiza semantică, cât şi în optimizarea codului, pentru diferite scopuri, precum: detectarea folosirii variabilelor neiniţializate, propagarea constantelor, detectarea apariţiilor de folosire pentru variabile, verificarea tipurilor etc.

4.6.3. Ecuaţii de flux de date

Pe lângă fluxul de control, în orice program există şi un flux de date, care descrie modul în care sunt propagate valorile variabilelor şi al constantelor în program. O analiză a fluxului de date poate oferi informaţii importante în faza de analiză semnatică, dar se poate folosi şi în fazele următoare. Principalele aplicaţii ale acestei metode sunt: determinarea variabilelor active[4] şi propagarea copiilor (variabilelor). În analiza semantică metoda este folosită în principal pentru a detecta starea variabilelor (declarată sau nu, iniţializată sau nu, valoare modificată sau nu).

Analiza fluxului de date se bazează pe scrierea şi rezolvarea unor sisteme de ecuaţii, numite ***ecuaţii de flux de date***, care reprezintă informaţia în diferite puncte ale unui program. Analiza fluxului de date se poate face în două moduri:

1. "înainte": propagarea valorilor în graf se face în direcţia fluxului de control; forma generală a ecuaţiilor de flux de date este:

   $out[I] = gen[I] \cup (in[I] - kill[I])$

   cu semnificaţia: "informaţia la terminarea unei instruţiuni I este fie generată în interiorul instrucţiunii, fie transmisă la început şi nu se pierde (nu e distrusă) în timp ce fluxul traversează instrucţiunea";

2. "înapoi": propagarea valorilor în graf se face în sens opus fluxului de control; forma generală a ecuaţiilor de flux de date este:

   $in[I] = gen[I] \cup (out[I] - kill[I])$

   cu semnificaţia: "informaţia la începutul unei instruţiuni I este fie generată în interiorul instrucţiunii, fie transmisă de la terminare şi nu se pierde (nu e distrusă) în timp ce fluxul traversează instrucţiunea".

Mulţimile care se definesc au următoarele semnificaţii (pentru un nod N) [ASU86]:

[4] în engleză *live variable*, adică o variabilă ce posedă o valoare care va fi folosită în continuare în program

- $IN(N)$ informaţia care intră în nodul N;
- $OUT(N)$ informaţia care iese din nodul N;
- $GEN(N)$ informaţia care se generează în nodul N;
- $KILL(N)$ informaţia care se distruge în nodul N.

Se poate observa că cele două mulţimi $IN(N)$ şi $OUT(N)$ înlocuiesc stiva din interpretarea simbolică. Deoarece aceste ecuaţii se asociază grafului de flux de control, trebuie să precizăm anumite relaţii între nodurile succesoare:

$$IN(N) = \cup\, OUT(M), \forall\, M \textit{ predecesor al lui } N$$

Modul de obţinere şi rezolvare a ecuaţiilor de flux de date depinde de:

- mulţimile *gen* şi *kill*, de generare şi respectiv distrugere, depind de informaţia dorită - scopul aplicării metodei;
- deoarece fluxul de date urmează fluxul de control, analiza lui este influenţată de structurile de control ale programului. Pentru a calcula $out[I]$ trebuie să presupunem că există un unic punct de terminare prin care fluxul de control iese din instrucţiune. De aceea, ecuaţiile de flux de date se asociază de obicei blocurilor de bază, în defavoarea instrucţiunilor simple.

Rezolvarea ecuaţiilor se face prin traversări repetate ale grafului de control, calculând mulţimile IN şi OUT până când acestea nu se mai modifică.

Exemplul 4.6. Vom considera graful fluxului de control (figura 4.6) pentru secvenţa *if a>2 then b:=1 else a:=1*. Folosind ecuaţii de flux de date dorim să stabilim starea variabilelor a şi b la terminarea secvenţei, pornind de la presupunerea că a este iniţializat şi b este neiniţializat.

$$\begin{array}{lll}
y > 0: & IN = & \{a = init; b = neinit\}; KILL = \emptyset; GEN = \emptyset; \\
 & & OUT = \{a = init; b = neinit\} \\
b := 1: & IN = & \{a = init; b = neinit\}; KILL = \{b\}; GEN = \{b\}; \\
 & & OUT = \{a = init; b = init\} \\
a := 1: & IN = & \{a = init; b = neinit\}; KILL = \{a\}; GEN = \{a\}; \\
 & & OUT = \{a = init; b = neinit\}
\end{array}$$

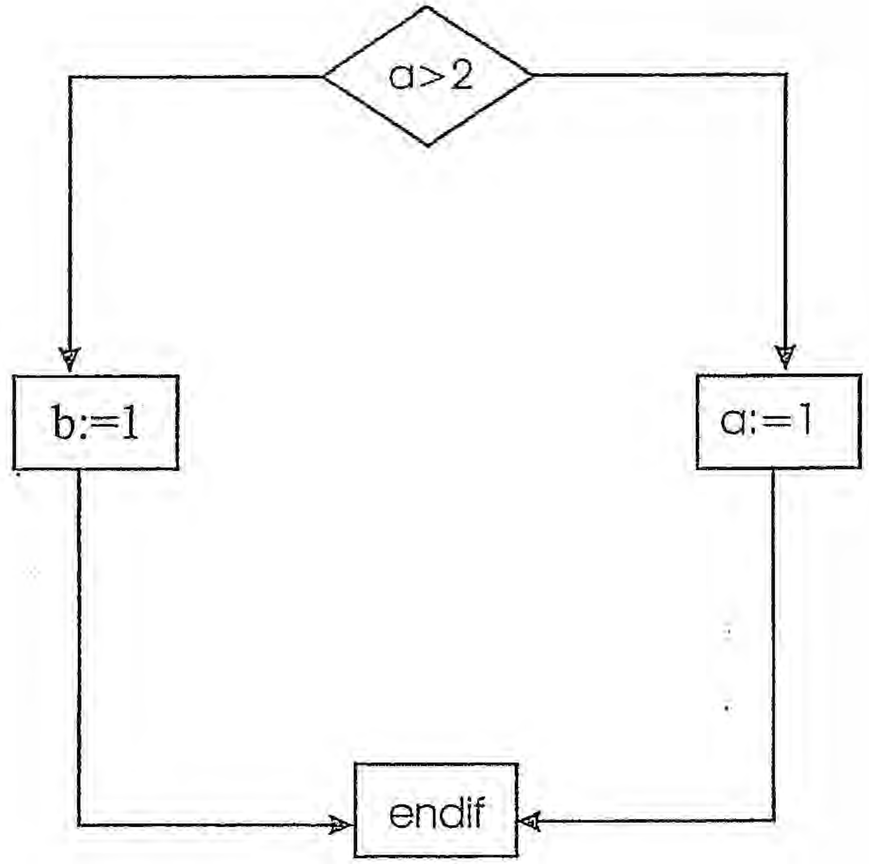

Figura 4.12: Ecuaţii de flux de date pentru $if\ a < 2\ then\ b := 1\ else\ a := 1$

$$endif: \quad OUT = OUT(b := 1) \cup OUT(a := 1) = \{a = init; b = init \vee neinit\}$$

Se observă faptul că în nodurile de unificare din graful fluxului de control se aplică o unificare şi asupra mulţimilor OUT calculate pe cele două ramuri. Rezolvarea ecuaţiilor a evidenţiat faptul că e posibil ca variabila b să nu posede valoare după această instrucţiune if, deoarece numai una din ramuri conţine o atribuire către variabila b.

5. GENERAREA CODULUI INTERMEDIAR

5.1. Cod intermediar

Majoritatea compilatoare folosesc treceri multiple peste programul sursă sau o reprezentare a lui, chiar dacă din punct de vedere teoretic compilarea poate fi realizată printr-o singură trecere. Motivul acestei abordări este legat de eficienţă: cod mai bun, care să ocupe mai puţin spaţiu şi al cărui timp de execuţie să fie cât mai mic. Programul sursă este reprezentat în interiorul compilatorului, de la o fază la alta, folosind un *limbaj intermediar*, numit uneori şi reprezentare internă. Fiecare fază a compilării prelucrează această reprezentare extinzând-o cu informaţii adiţionale furnizate de faza respectivă.

După cum am menţionat în primul capitol, compilatorul este împărţit în două componente mari: analiza şi sinteza (figura 1.2). Criteriul după care o anumită fază a compilării este de analiză sau de sinteză îl constituie forma programului cu care lucrează faza respectivă: program sursă sau program obiect. Codul intermediar poate fi considerat ca o interfaţă între analiză şi sinteză, trebuind să îndeplinească două cerinţe: să reflecte funcţionalitatea programului sursă şi să posede o formă adecvată optimizării şi generării eficiente a codului obiect.

O problemă importantă a codului intermediar este cea a reutilizării: proiectarea de compilatoare este dificilă dacă luăm în considerarea multitudinea de limbaje, sisteme de operare şi platforme. Ar fi mult mai profitabil să proiectăm un limbaj intermediar cât mai general pentru întreaga familie de compilatoare. Astfel, problema construirii a $m \times n$ compilatoare (pentru m limbaje sursă şi n platforme rezultat) poate fi redusă la a analiza m limbaje sursă, a construi o reprezentare internă comună şi a genera cod pentru n platforme rezultat ($m + n$), fapt ilustrat şi de figura 5.1.

Acestă idee a fost implementată în diverse compilatoare (WAM[Ait91], p-cod[MPL84], ALPHA [DQR95]), dar a primit o nouă dimensiune o dată cu apariţia ideii de compilare dinamică. Merită menţionate aici următoarele limbaje intermediare:

- *Java bytecode* (cod de octeţi) - implementează cazul în care un singur limbaj sursă, Java, este compilat pe mai multe maşini. Generarea codului maşină este sarcina unei maşini virtuale (Java Virtual Ma-

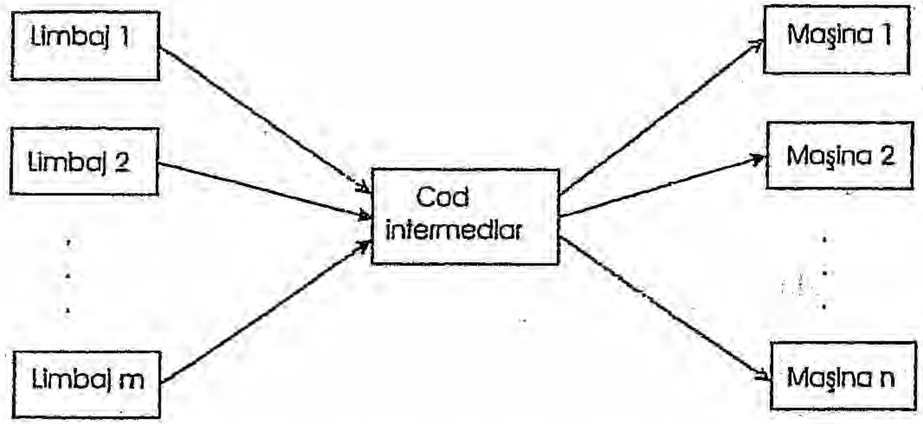

Figura 5.1: Crearea de compilatoare pentru m limbaje şi n maşini folosind cod intermediar

chine) specifică fiecărei platforme în parte. Performanţele acestui limbaj intermediar au avut ca rezultat implementarea unor compilatoare pentru alte limbaje care generează java bytecode.

- *MSIL (Microsoft Intermediate Language)* - permite mai multe limbaje sursă şi mai multe maşini rezultat, reprezentând cea mai generală abordare a acestui concept de limbaj intermediar. Generarea codului maşină se face la momentul execuţiei prin intermediul sistemului de execuţie .NET.

- cea mai autentică implementare a ideii de limbaj intermediar comun o reprezintă familia de compilatoare GNU, care se bazează pe *GNU RTL (GNU Register Transfer Language)*, care poate fi generat din mai multe limbaje (C, C++, Fortran, Ada) şi poate fi folosit pentru a genera cod maşină pentru diferite platforme.

Calităţile principale pe care trebuie să le posede un bun limbaj intermediar sunt: uşor de transcris din arborele de sintaxă abstractă şi uşor de translatat în cod maşină. În principiu, compilatoarele aleg una din următoarele forme de reprezentare pentru limbajul intermediar:

- arbore atributat;
- formă poloneză postfixată;
- cod cu trei adrese.

Arborii atributaţi constituie, probabil, cea mai folosită reprezentare internă în compilatoare. Fiecare construcţie din programul sursă este reprezentată printr-un nod în arbore, cu referinţe spre toate elementele

construcţiei respective. În plus, fiecare nod conţine nişte atribute, informaţii adiţionale, necesare în fazele de sinteză. Avantajul acestei metode constă în faptul că arborii sunt deja construiţi în faza de analiză sintactică, structura de arbore poate fi parcursă uşor şi este avantajoasă ca reprezentare pentru unele faze. În acestă alegere a reprezentării faza de analiză semantică se suprapune cu cea de generare de cod intermediar.

Forma poloneză postfixată: Notaţia "poloneză" a fost introdusă de logicianul polonez Lukasiewicz. Pentru o expresie, de exemplu $a + b * c$, există trei tipuri de notaţii:

- infixată: forma uzuală de scriere cu care suntem obişnuiţi, în care operatorul este încadrat de cei doi operanzi; expresia infixată este: $a + b * c$;
- prefixată: în care operatorul precede operanzii; expresia prefixată este: $* + abc$;
- postfixată: în care operanzii preced operatorul; expresia postfixată este: $abc * +$.

Forma poloneză postfixată este o alternativă avantajoasă ca limbaj intermediar, deoarece se potriveşte modelului stivei: operanzii sunt introduşi în stivă, iar operaţiile se efectuează în vârful stivei - se scot operanzii din stivă şi se pune rezultatul. MSIL foloseşte acest model pentru majoritatea operaţiilor. În plus, forma poloneză postfixată posedă două proprietăţi foarte importante: nu conţine paranteze şi operatorii apar în ordinea în care se execută (nu e necesară prioritatea operatorilor).

Orice expresie în formă infixată poate fi trecută la forma postfixată şi reciproc. Algoritmul 5.1 prezintă o astfel de metodă care se bazează pe folosirea unei stive auxiliare.

Pentru operatori unari se pot folosi operatori binari echivalenţi: $-b$ poate fi scris ca $0 - b$ sau se poate introduce o notaţie specială, pentru a diferenţia operatorul unar '-' de operatorul binar '-'.

5.2. Cod cu trei adrese

Definiţia 5.1. *[Şer87] **Codul cu trei adrese** este o secvenţă de instrucţiuni cu un format simplu, foarte apropiat de codul obiect, cu următoarea formă generală:*

$< rezultat >=< arg1 >< op >< arg2 >$

unde $< rezultat >, < arg1 >, < arg2 >$ reprezintă variabile sau constante din program sau nume temporare create de compilator, iar $< op >$ este un operator binar.

Algoritmul 5.1 Forma poloneză postfixată

```
INPUT: x[i], i=1,n (expresia în forma poloneză infixată), prior[k],
k=1,nr_operatorilor (prioritățile operatorilor)
OUTPUT: fpp[i], i=1,m (expresia în forma poloneză postfixată)
j:=0;
for i:=1,n do
  if x[i] operand then
    j := j + 1
    fpp[j]:=x[i]
  end if
  if x[i] operator then
    while prior[varf(stiva)] >= prior[x[i]] do
      pop(stiva,o);
      j:=j+1;
      fpp[j]:=o
    end while
    push(stiva,x[i]);
  end if
  if x[i]='(' then
    push(stiva,x[i])
  end if
  if x[i]=')' then
    repeat
      pop(stiva,o);
      if o<>'(' then
        j:=j+1;
        fpp[j]:=o
      end if
    until o='('
  end if
end for
while not(vida(stiva)) do
  pop(stiva,o);
  j:=j+1;
  fpp[j]:=o;
end while
```


Denumirea provine de la faptul că fiecare instrucțiune conține trei adrese: două pentru operanzi și una pentru rezultat. În această prezentare, vom folosi numele efectiv al variabilelor sau constantelor, dar în implementare se folosesc pointeri spre adresele din tabela de simboluri pentru variabilele și constantele din program. Pentru numele temporare noi introduse în scopul de a memora rezultate parțiale ale calculelor se va crea spațiu în tabela de simboluri și pointerii vor referi locațiile respective.

Principalul avantaj al acestei reprezentări îl constituie simplitatea instrucțiunilor. O instrucțiune în codul cu trei adrese conține un singur operator, ceea ce va permite generarea codului obiect aproape direct. Codul cu trei adrese constituie o reprezentare liniară a arborelui de sintaxă abstractă, în care ordinea de execuție a instrucțiunilor este mult mai clară. În plus, codul poate fi ușor manipulat și rearanjat în faza de optimizare.

Exemplul 5.1. Pentru expresia $b*b-4*a*c$, arborele de sintaxă abstractă și codul cu trei adrese corespunzător sunt ilustrate în figura 5.2.

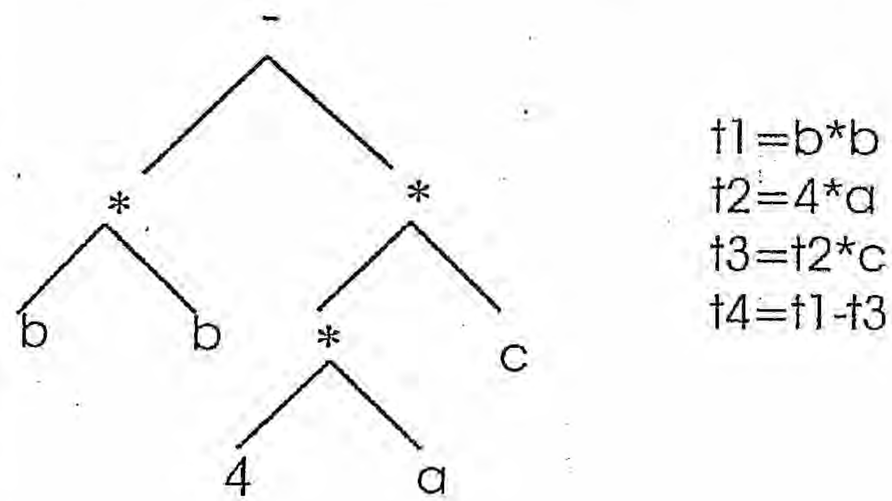

Figura 5.2: Arborele de sintaxă abstractă și codul cu trei adrese pentru $b*b-4*a*c$

Forma generală a instrucțiunilor din codul cu trei adrese nu pare, la prima vedere, că s-ar potrivi peste șablonul construcțiilor sintactice ale unui program. Totuși, acestea pot fi aduse la niște forme particulare de instrucțiuni. Aceste cazuri speciale pot fi clasificate în:

1. Expresii care conțin operatori unari: vor fi tratate ca și când primul operand lipsește:

   $< rezultat >=< op >< arg2 >$

2. Instrucţiunea de atribuire: '=' joacă rolul atribuirii. Dacă vom avea o atribuire de forma $a := b$ atunci instrucţiunea cu trei adrese va fi $a = b$ (lipsesc operatorul şi al doilea operand).

3. Salt necondiţionat: instrucţiunea va avea forma *goto* L, unde L este eticheta unei instrucţiuni din codul cu trei adrese.

4. Salt condiţionat: instrucţiunea va avea forma $if\ c\ goto\ L$, cu semnificaţia: dacă c este evaluat la adevărat atunci se face un salt necondiţionat la instrucţiunea cu eticheta L, altfel (dacă c este evaluat la fals) se execută instrucţiunea imediat următoare din codul cu trei adrese.

5. Apelul de procedură $p(x_1, x_2, ..., x_n)$ va fi reprezentat prin următoarea secvenţă de instrucţiuni:

   param x_1
   param x_2
   ...
   param x_n
   call p, n

   cu semnificaţia: se evaluează parametrii prin instrucţiunea *param*, în ordine, şi apoi se apelează procedura p cu n parametri. Precizarea numărului de parametri este necesară apelurilor imbricate.

6. Variabile indexate: $< arg1 >, < arg2 >, < rezultat >$ pot fi elemente de tablou, de forma $a[i]$. Pentru simplificare, se permite o singură dimensiune şi deci tablourile cu mai multe dimensiuni vor trebui liniarizate: $a[i, j]$ va avea ca şi corespondent în codul cu trei adrese $a[i*(dim-1)+j]$, unde *dim* este prima dimensiune a tabloului.

7. Modul de adresare al unei variabile (prin adresă sau referinţă) poate fi şi el o operaţie în codul cu trei adrese: $\&x, *x$.

5.2.1. Reprezentarea codului cu trei adrese

Codul cu trei adrese este o reprezentare abstractă, a cărui implementare se realizează ca un tablou de înregistrări, fiecare înregistrare corespunzând unei instrucţiuni. Formele posibile ale acestor tipuri de înregistrări sunt: cvadruple, triplete şi triplete indirecte.

Cvadruple

Cvadruplul este o structură de înregistrare cu patru câmpuri:
$< op > < arg1 > < arg2 > < rezultat >$.

Exemplul 5.2. Pentru codul cu trei adrese considerat în exemplul 5.1, secvenţa de cvadruple este:

op	arg1	arg2	rez
*	b	b	t1
*	4	a	t2
*	t2	c	t3
-	t1	t3	t4

Pentru salturile condiţionate sau necondiţionate, eticheta de salt se trece la rezultat.

Exemplul 5.3. Fie instrucţiunea $if\ (a < b)\ AND\ c\ then\ a := -1\ else\ b := a + c;$. Secvenţa de cvadruple corespunzătoare este:

Nr.	op	arg1	arg2	rez
(1)	<	a	b	t1
(2)	AND	t1	c	t2
(3)	if	t2		(7)
(4)	+	a	c	t3
(5)	:=	t3		b
(6)	goto			(9)
(7)	@	1		t4
(8)	:=	t4		a
(9)				

Observaţii:

- În cazul atribuirii $var := exp$, var va apărea ca şi rezultat, atribuirea ca şi operator, iar expresia ca şi argument, iar cel de-al doilea argument lipseşte.
- Numărul de ordine al cvadruplului joacă rol de etichetă. Când rezultatul este un salt, pentru a nu face confuzie între constanta 7 şi eticheta 7, acesta din urmă apare între paranteze.
- '@' denotă operatorul minus unar.
- Instrucţiunile (2)-(5) reprezintă ramura *else*, iar cele de la (7) la (8) corespund părţii *then*. Rolul saltului necondiţionat din instrucţiunea (6) este de a evita execuţia ramurii *then* după ce ramura *else* a fost executată. Saltul se face la următorul cvadruplu după cele corespunzătoare instrucţiunii *if*.

Triplete

Tripletele sunt foarte asemănătoare cvadruplelor, cu excepţia reprezentării rezultatului. În cazul cvadruplelor rezultatul era reprezentat explicit, iar la triplete se consideră că înregistrarea care calculează un anumit rezultat îl va şi memora. Se evită astfel crearea numelor temporare şi încărcarea tabelei de simboluri cu acestea.

Deci, tripletul este o structură de înregistrare cu trei câmpuri:

$< op > < arg1 > < arg2 >$.

Exemplul 5.4. Pentru codul cu trei adrese considerat în exemplul 5.1 şi secvenţa de cvadruple din exemplul 5.2, secvenţa de triplete este:

Nr.	op	arg1	arg2
(1)	*	b	b
(2)	*	4	a
(3)	*	(2)	c
(4)	-	(1)	(3)

Dezavantajul principal al acestei reprezentări îl constituie faptul că instrucţiunile pot suferi modificări în faza de optimizare şi aceste modificări trebuie efectuate şi asupra referirilor la instrucţiunile respective. Situaţia poate fi rezolvată, prin crearea unei liste suplimentare care să conţină numărul de ordine al tripletelor în ordinea în care acestea se execută. Această reprezentare poartă numele de *triplete indirecte*.

Compararea reprezentărilor

Alegerea uneia sau alteia dintre reprezentări trebuie să ţină cont de avantajele, respectiv dezvantajele fiecăreia, care se pot sintetiza în tabelul următor:

	Cvadruple	Triplete
Avantaje	-referire explicită la rezultate şi accesare imediată în TS -algoritmi mai simpli în optimizarea codului	-dimensiune mai mică (3 câmpuri, în loc de 4) - evită supraîncărcare TS
Dezavantaje	-dimensiune mai mare -spaţiu suplimentar în TS	-la optimizarea codului mutarea instrucţiunii implică modificarea referinţelor la ea -la generare de cod e necesară scanarea tuturor tripletelor pentru a şti nr. de locaţii temporare active

5.2.2. Trecerea la cod cu trei adrese a arborelui sintactic

Construcţia codului cu trei adrese se poate face printr-o traversare a arborelui atribuit. Această operaţie se mai numeşte şi traducere orientată de sintaxă şi se bazează pe aplicarea unor proceduri în funcţie de nodul curent în arbore, de fapt, de producţia care stă la baza construcţiei subarborelui nodului respectiv. Vom prezenta doar câteva astfel de proceduri, celelalte bazându-se pe aceeaşi idee.

Pornind de la regula de producţie aplicată, vom asocia anumite acţiuni semantice, şi anume:

- acţiune de calcul al atributelor, care sunt folosite în generarea codului pentru tot subarborele;
- parte de generare de instrucţiuni în cod cu trei adrese.

Considerăm gramatica corespunzătoare unei instrucţiuni de atribuire, în care expresia din partea dreaptă poate conţine operatori aritmetici, iar operanzii sunt identificatori sau constante.
$G = (\{S, E, T, F\}, \{:=, +, *, -, (,), id, const\}, P, S)$, cu următoarele producţii:
$S \rightarrow id := E$
$E \rightarrow E + E | E * E | - E | (E) | id | const$

Generarea codului cu trei adrese, (numită *gen*), alocă nume temporare pentru calculele intermediare prin apelul funcţiei *nou_temp*, care va returna de fiecare dată un nume nou. Atributele de care avem nevoie sunt:

- $E.loc$ - locaţia unde se află valoare lui E;
- $E.cod$ - secvenţa de instrucţiuni cu trei adrese care evaluează E.

Astfel, atributul $S.cod$ va genera secvenţa de instrucţiuni cu trei adrese pentru atribuire.

Producţie	Reguli semantice
$S \rightarrow id := E$	S.cod = E.cod ||gen(id.loc ':=' E.loc)
$E \rightarrow E_1 + E_2$	E.loc = nou_temp E.cod = E_1.cod ||E_2.cod|| gen(E.loc '=' E_1.loc '+' E_2.loc)
$E \rightarrow E_1 * E_2$	E.loc = nou_temp E.cod = E_1.cod ||E_2.cod|| gen(E.loc '=' E_1.loc '*' E_2.loc)
$E \rightarrow -E_1$	E.loc = nou_temp E.cod = E_1.cod || gen(E.loc '=' '@' E_1.loc)
$E \rightarrow (E_1)$	E.loc = E_1.loc E.cod = E_1.cod
$E \rightarrow id$	E.loc = id.loc E.cod = ' '
$E \rightarrow const$	E.loc = const.loc E.cod = ' '

Exemplul 5.5. Considerând instrucţiunea $i := a + b \times c$, aplicarea pas cu pas a producţiilor şi regulilor semantice conduce la următorul cod cu trei adrese. Tabelul prezintă producţia aplicată codul generat până în acel moment şi accesarea locaţiilor. Pentru simplificare, am considerat că locaţia unui identificator este dată de numele lui.

Producţie	Locaţie	Cod
$S \rightarrow id := E$		E.cod i := E.loc
$E \rightarrow E_1 + E_2$	$E.loc = T_1$	$E_1.cod$ $E_2.cod$ $T_1 = E_1.loc + E_2.loc$ $i := T_1$
$E_1 \rightarrow id$	$E_1.loc = id.loc$	$E_2.cod$ $T_1 = a + E_2.loc$ $i := T_1$
$E_2 \rightarrow E_{21} * E_{22}$	$E_2.loc = T_2$	$E_{21}.cod$ $E_{22}.cod$ $T_2 = E_{21}.loc \times E_{22}.loc$ $T_1 = a + T_2$ $i := T_1$
$E_{21} \rightarrow id$	$E_{21}.loc = id.loc$	$E_{22}.cod$ $T_2 = b \times E_{22}.loc$ $T_1 = a + T_2$ $i := T_1$
$E_{22} \rightarrow id$	$E_{22}.loc = id.loc$	$T_2 = b \times c$ $T_1 = a + T_2$ $i := T_1$

Dacă dorim să generăm cod cu trei adrese pentru o instrucţiune condiţională, vom avea nevoie de o funcţie *nou_eticheta*, care va genera o etichetă nouă. Secvenţa de instrucţiuni cu trei adrese trebuie să cuprindă: instrucţiunile pentru evaluarea expresiei din condiţie ($E.cod$) care va genera un salt la ramura *then* sau *else*, apoi secvenţa corespunzătoare ramurii *else* ($S_1.cod$) care se încheie cu un salt necondiţionat la instrucţiunea următoare de program şi în final secvenţa corespunzătoare ramurii *then* ($S_2.cod$). Atributele *true* şi *false* asociate expresiei E sunt necesare pentru calculul etichetei de salt în funcţie de valoarea de adevăr a expresiei.

Producţie	Reguli semantice
$S \rightarrow if\ E\ then\ S_1\ else\ S_2$	E.false = *nou_eticheta* E.true = *nou_eticheta* S_1.urm = S.urm S_2.urm = S.urm S.cod = E.cod ||*gen*($E.false$' :')||S_2.cod || *gen*('*goto*' S.urm)|| *gen*($E.true$' :')||S_1.cod

Exemplul 5.6. Să considerăm următoarea instrucţiune condiţională:

$if(a < b)\ then\ a := -1\ else\ a := 1$

Aplicarea regulii semantice corespunzătoare va duce la următoare configuraţie a codului cu trei adrese:

$E.cod$
$(1) : S_2.cod$
$goto\ S.urm$
$(2) : S_1.cod$
$S.urm : \ldots$

Înlocuind acum codul pentru evaluarea expresiei ($E.cod$) şi a instrucţiunilor de pe cele două ramuri $S_1.cod$ şi $S_2.cod$) obţinem:

$if\ a < b\ goto\ (2)$
$(1) : a := 1$
$goto\ (3)$
$(2) : T_1 = @1$
$a := T_1$
$(3) : \ldots$

Pentru o instrucţiune de ciclare, generare codului cu trei adrese este asemănătoare instrucţiunii condiţionale. Dacă luăm în considerare o instrucţiune de ciclare cu precondiţie, atunci saltul pentru evaluarea condiţiei la fals se face la instrucţiunea imediat următoare (ca şi când nu ar exista ramura *else*), iar secvenţa de instrucţiuni corespunzătoare evaluării la adevăr a condiţiei se termină cu un salt necondiţionat la începutul instrucţiunii *while*.

Producţie	Reguli semantice
$S \rightarrow while\ E\ do\ S_1$	S.begin :=*nou_eticheta* E.true = *nou_eticheta* E.false = S.urm S_1.urm = S.begin S.cod = gen(S.begin ':') || E.cod || gen(E.false ':')||gen(goto S.urm)|| gen(E.true ':')||S_1.cod|| gen(goto S.begin)

Exemplul 5.7. Instrucţiunii $while\ a < b\ do\ a := 1 - a$ îi corespunde, conform regulilor specificate, următoarea secvenţă în codul cu trei adrese:

$(1) if\ a < b\ goto\ (2)$
$goto\ (3)$
$(2) : T_1 = a - 1$
$a := T_1$
$goto\ (1)$
$(3) : \ldots$

6. OPTIMIZAREA CODULUI INTERMEDIAR

Algoritmii folosiţi în fazele anterioare ale compilării generează diferitele forme intermediare ale programului pe baza unor şabloane sau a unor reguli. De aceea este posibil, în special pentru programele mari, ca programul obiect obţinut să aibă dimensiune mare. *Optimizarea codului constă dintr-o colecţie de transformări care se aplică în scopul obţinerii unui program mai eficient.* Eficienţa ia în considerare două aspecte: *spaţiul* - programul obiect să fie de dimensiuni cât mai mici şi *timpul* - execuţia lui să dureze cât mai puţin.

După cum menţionam şi în capitolul 1, optimizarea nu este o componentă obligatorie a unui compilator, programul obiect putând fi generat şi fără faza de optimizare. În acest capitol ne vom referi doar la tehnicile de optimizare ale codului intermediar, optimizarea codului obiect fiind prezentată în capitolul 7.

Deoarece operaţiile de optimizare în sine necesită şi ele timp de execuţie, efectuarea lor trebuie să îşi justifice resursele consumate – cât de mult se îmbunătăţeşte codul generat. În primul rând va trebui să identificăm părţile de program care merită supuse optimizării, urmând ca apoi să decidem ce transformări să aplicăm. Ciclurile reprezintă un astfel de caz, deoarece corpul unei instrucţiuni repetitive se execută de mai multe ori.

Principiul de bază care trebuie luat în considerare este că orice transformare asupra codului intermediar trebuie să îi conserve efectul, deci codul intermediar şi cel optimizat trebuie să fie semantic echivalente.

În linii mari, faza de optimizare constă din următoarele componente:

- analiza fluxului de control;
- analiza fluxului de date;
- transformări.

6.1. Graful de flux al unui program

În faza de optimizare se analizează *graful de flux* asociat unui program, în care nodurile sunt blocurile de bază, iar arcele indică transferul controlului (execuţiei) de la un bloc la altul. Această reprezentare diferă

de graful considerat în faza de analiză semantică prin structura nodurilor: graful de flux pentru optimizare are ca noduri blocuri de bază (secvenţe de instrucţiuni în cod cu trei adrese), comparativ cu graful de flux al controlului care avea nodurile din arborele de sintaxă abstractă.

Un ***bloc de bază*** [ASU86] este o secvenţă de instrucţiuni consecutive în care fluxul de control intră în prima instrucţiune, iese la sfârşitul ultimei instrucţiuni, fără posibilitate de salt sau ramificare. Partiţionarea[1] unui program în blocuri de bază se face după următoarele reguli:

1. Se determină prima instrucţiune a unui bloc, astfel:
   - prima instrucţiune din program;
   - orice instrucţiune care este rezultatul unui salt condiţionat sau necondiţionat;
   - orice instrucţiune imediat următoare unui salt condiţionat sau necondiţionat.
2. Pentru fiecare primă instrucţiune a unui bloc se construieşte blocul de bază corespunzător care cuprinde: prima instrucţiune şi toate instrucţiunile care urmează până la următoarea primă instrucţiune (exclusiv aceasta).

Arcele grafului se construiesc astfel: există un arc între blocurile B_i şi B_j dacă B_j urmează după B_i într-o secvenţă de execuţie, mai exact într-una din următoarele situaţii:

- există un salt condiţionat sau necondiţionat de la ultima instrucţiune a lui B_i la prima instrucţiune a lui B_j;
- B_j urmează imediat după B_i în ordinea secvenţială din program şi ultima instrucţiune a lui B_i nu este salt necondiţionat.

Exemplul 6.1. Fie următoarea secvenţă de cod cu trei adrese:

$$
\begin{array}{lll}
(1) & i & = 1 \\
(2) & s & = 0 \\
(3) & t1 & = s + i \\
(4) & s & = t1 \\
(5) & t2 & = i + 1
\end{array}
$$

[1] Prin partiţionare se înţelege împărţirea programului astfel încât fiecare instrucţiune aparţine unui singur bloc

(6) $i \quad = t2$

(7) $if \quad i < 10\ goto(3)$

Determinarea primelor instrucţiuni ale blocurilor va avea ca rezultat instrucţiunile (1), deoarece e prima instrucţiune şi (3) ca rezultat al unui salt din instrucţiunea (7). Astfel vom avea două blocuri de bază. Graful de flux asociat acestei secvenţe este prezentat în figura 6.1.

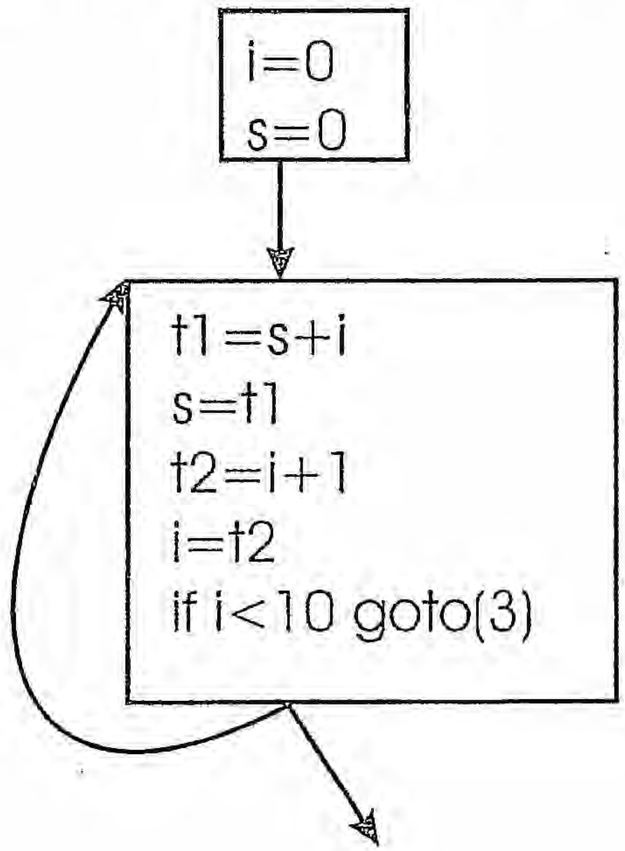

Figura 6.1: Graful de flux asociat unei secvenţe de program

Transformările care se aplică în interiorul unui bloc de bază se numesc *locale*, iar celelalte *globale*.

Principalele acţiuni posibile de optimizare sunt:

- realizarea unor calcule la compilare;
- eliminarea calculelor redundante;
- eliminarea codului inaccesibil;
- optimizarea ciclurilor.

6.2. Optimizări locale

Realizarea unor calcule la compilare

Presupunând că avem o instrucţiune de tipul $a = x + y$ şi x şi y sunt constante, atunci valoarea lui a poate fi calculată la momentul compilării. Astfel de situaţii sunt:

x:= 2+2;

x:= 2 * pi; { *unde pi este constantă*}

Tot în acestă categorie se încadrează conversiile implicite realizate de calculator. Pentru expresia $a = a + 6$, unde a este un număr real, codul intermediar conţine un calcul intermediar pentru conversia întregului 6 la numărul real 6. Aceasta poate fi realizată în momentul compilării.

Eliminarea calculelor redundante

Eliminarea calculelor redundante sau a subexpresiilor comune se bazează pe folosirea repetată a unor calcule în program.

Exemplul 6.2. Următoarea secvenţă de program prezintă 2 subexpresii comune $C * B$ şi $D + C * B$.

D:=D+C*B

A:=D+C*B

C:=D+C*B

Secvenţa corespunzătoare de cod cu trei adrese, reprezentat ca triplete, este:

(1)	*	C	B
(2)	+	D	(1)
(3)	:=	(2)	D
(4)	*	C	B
(5)	+	D	(4)
(6)	:=	(5)	A
(7)	*	C	B
(8)	+	D	(7)
(9)	:=	(8)	C

Aceste subexpresii comune se regăsesc în triplete identice, ca şi (1), (4), (7), dar şi unele mai greu de observat ca (2), (5), (8). Ideea este că un astfel de calcul se poate efectua o singură dată şi valoarea respectivă folosită de câte ori e nevoie, fără a mai fi necesar calculul. Acest lucru nu este întotdeauna posibil. De exemplu, tripletele (4) şi (7) pot fi eliminate şi se va folosi tripletul (1) în locul lor, dar tripletele (5) şi (8) nu pot fi

înlocuite de tripletul (2). O parcurgere atentă a codului va evidenţia faptul că unul dintre operanzi, şi anume D, şi-a modificat valoarea între cele două triplete identice, în tripletul (3).

A i-a operaţie dintr-un bloc se numeşte ***redundantă*** dacă există o operaţie anterioară identică (fie j) şi dacă niciunul dintre operanzii de care depinde operaţia i nu se modifică de către o a treia operaţie cuprinsă între işi j. Tripletele (4) şi (7) sunt redundante (cu tripletul (1)), iar tripletul (8) este redundant cu tripletul (5).

Pentru a putea aplica această optimizare va trebui să determinăm care sunt operaţiile redundante şi doar pe acelea le vom elimina.

Algoritmul pentru eliminarea redundanţelor scanează tripletele în ordinea operaţiilor şi foloseşte o relaţie de dependenţă: $dep(A)$ este numărul de dependenţe pentru variabila $A, \forall A$ variabilă în program:

1. iniţial, $dep(A) = 0$ (pentru că nu depinde de nici un triplet);
2. dacă tripletul i atribuie o valoare variabilei A atunci $dep(A) = i$;
3. la execuţia tripletului i avem: $dep(i) = 1 + (maxdep(A))$ unde A e operand în tripletul i.

Dacă tripletul i este identic cu tripletul $j, j < i$ atunci tripletul i e redundant dacă şi numai dacă $dep(i) = dep(j)$.

Există anumite situaţii în care subexpresia comună nu este uşor observabilă. Câteva cazuri speciale sunt:

1. $A := 1 + B + C + 1$ şi $B := B + C + 6$. Dacă operaţiile cu aceeaşi prioritate se execută în ordine de la stânga la dreapta, subexpresia comună $B + C$ nu va apărea în codul cu trei adrese.
2. $B + C$ şi $C + B$ reprezintă aceeaşi subexpresie numai dacă se ia în considerare proprietatea de comutativitate a adunării.
3. $x[i]$ şi $x[i+1]$. Folosirea variabilelor indexate se traduce în cod intermediar printr-o succesiune de instrucţiuni care calculează valoarea elementului pornind de la adresa primului element şi deplasamentul dat de indice. Pentru cele două elemente din şir calculul deplasamentului poate fi făcut relativ: În loc de:

   t1=adresa(x)

   t2=t1+i

   t3=x[t2]

   t4=i+1

   t5=t1+t4

   t6=x[t5]

   putem genera

   t1=adresa(x)

   t2=t1+i

   t3=x[t2]

   t4=t2+1

   t6=x[t5]

Eliminarea codului inaccesibil

Un cod inaccesibil este o secvenţă de program care nu se execută pe nici un drum prin graful de control. Este puţin probabil ca astfel de cod să existe intenţionat în programul sursă, una dintre posibilităţi fiind aceea a unei condiţii care se evaluează la o constantă şi prin urmare una dintre ramuri nu se va executa niciodată. Transformările efectuate în optimizare pot genera la rândul lor astfel de fragmente de cod şi existenţa secvenţei respective nu mai are sens, codul optimizat fiind mai scurt.

6.3. Optimizarea ciclurilor

Detectarea ciclurilor este relativ uşoară în limbaje de nivel înalt, datorită construcţiilor sintactice specifice (*while, repeat, for*) şi mai dificilă în limbajele de nivel jos, necesitând analiza grafului de flux.

Optimizarea ciclurilor este considerată importantă datorită unei cunoscute remarci de programare, numită "regula 90 - 10": 90% din timpul total de execuţie al unui program are loc în 10% din codul sursă. Explicaţia constă în existenţa ciclurilor. Acţiunile principale de optimizare a ciclurilor sunt:

- factorizarea invarianţilor de cicluri şi
- reducerea puterii operaţiilor.

Factorizarea invarianţilor de cicluri

Definiţia 6.1. *O instrucţiune sau o expresie este un invariant de ciclu dacă fiecare operand este:*

- *constantă sau*

- *are toate definirile în afara ciclului sau*
- *are exact o definire şi aceea este un invariant de ciclu.*

Existenţa unui astfel de invariant în interiorul ciclului implică efectuarea instrucţiunii respective de câte ori se execută ciclul. Optimizarea constă în a scoate această instrucţiune înaintea ciclului, ea executându-se astfel o singură dată.

Exemplul 6.3. O secvenţă de program ce conţine un invariant de ciclu, înainte şi după optimizare:

```
for(i=0, i<=n,i++)
    { x=y+z;
    a[i]=i*x}
```

```
x=y+z;
for(i=0, i<=n,i++)
    { a[i]=i*x}
```

Reducerea puterii operaţiilor

Această optimizare are ca scop înlocuirea operaţiilor costisitoare (ca de exemplu înmulţirea) cu operaţii mai ieftine (adunarea) în definirea variabilelor de ciclare şi a altor variabile calculate în cicluri. Reducerea puterii operaţiilor pentru variabilele de ciclare se referă la cazul în care ele sunt indexate şi transformarea se referă la calcularea relativă a elementului în şir, după modelul descris în paragraful precedent.

Pentru celelalte calcule, reducerea puterii operaţiilor se bazează pe faptul că, de multe ori, calculele din interiorul unui ciclu reprezintă iteraţii, astfel că valoarea calculată la iteraţia i se poate exprima în funcţie de valoarea calculată la iteraţia $i-1$.

Exemplul 6.4. Considerând ciclul următor, în care v este un invariant de ciclu, el poate fi optimizat astfel:

```
for(i=k, i<=n,i++)
    { t=i*v;
    . . .}
```

```
t1=k*v;
for(i=k, i<=n,i++)
    { t=t1;
    t1=t1+v;...}
```


7. GENERAREA CODULUI OBIECT

Generarea codului obiect constă, după cum se observă şi în figura 1.3, în translatarea instrucţiunilor codului intermediar în instrucţiuni ale codului obiect. În timp ce codul intermediar se caracterizează prin: variabile (din program şi temporare), un set de instrucţiuni abstracte şi de faptul că nu există o stivă de execuţie, caracteristicile codului maşină sunt:

- număr finit de regiştri;
- variabilele sunt stocate în memorie;
- se foloseşte o stivă de execuţie şi un set de instrucţiuni de accesare a stivei;
- setul de instrucţiuni corespunde limbajului obiect.

De aceea, generarea de cod obiect din cod intermediar trebuie să ţină cont de două aspecte importante:

1. *alocarea regiştrilor* - care se ocupă de modul în care sunt stocate şi manipulate variabilele;
2. *selectarea instrucţiunilor* - modul şi ordinea în care se mapează instrucţiunile codului intermediar în instrucţiuni maşină.

Generarea codului obiect depinde în mod esenţial de tipul calculatorului şi al sistemului de operare existent. Există mai multe tehnici de structurare a codului executabil care ţin cont de aceste considerente tehnice, pe lângă cele legate de corectitudine şi viteză. Tehnicile de generarea a codului obiect se pot clasifica în funcţie de tipul calculatorului în:

- calculator cu un acumulator;
- calculator cu regiştri generali.

7.1. Generare codului pentru calculatoare cu acumulator

Generarea se bazează pe modelul maşinii de stivă, care reprezintă un model simplu de evaluare, folosind o stivă de valori pentru rezultate intermediare.

O ***maşină de stivă*** [GBJL02] constă dintr-o stivă pentru stocarea şi manipularea valorilor şi două tipuri de instrucţiuni:

- instrucţiuni pentru mutarea sau copierea valorilor în şi din capul stivei în memorie;
- instrucţiuni pentru operaţii asupra elementelor din capului stivei, care funcţionează astfel: operanzii se scot din stivă, se execută operaţia respectivă şi se depune în stivă rezultatul.

Acumulatorul va fi folosit pentru a efectua operaţia, stiva fiind dedicată stocării subexpresilor şi a rezultatului.

Exemplul 7.1. Pentru expresia 4*(3+1) codul intermediar, acumulatorul şi stiva arată ca în tabelul următor.

cod	acc	stiva
acc ← 4	4	<>
push acc	4	< 4 >
acc ← 3	3	< 4 >
push acc	3	< 3, 4 >
acc ← 1	1	< 3, 4 >
acc ← acc + cap	4	< 3, 4 >
pop	4	< 4 >
acc ← acc * cap	16	< 4 >
pop	16	<>

O altă caracteristică importantă a maşinii de stivă este că nu foloseşte regiştri. Stiva este manipulată prin două referinţe, una spre capul ei şi alta care referă începutul zonei alocate stivei. Avantajele folosirii maşinii de stivă pot fi sintetizate în: locaţia operanzilor şi a rezultatului este implicită (capul stivei) şi format simplu al instrucţiunilor (operanzii nu trebuie specificaţi). Datorită simplităţii sale, modelul a fost folosit cu succes pe calculatoare cu acumulator, ca VAX şi DEC PDP-11 şi este implementat în Java bytecode. Arhitectura calculatoarelor modificându-se în timp, majoritatea calculatoarelor de azi folosesc alt model mai eficient,

şi anume la cel al maşinii de regiştri, cu excepţia celor bazate pe arhitectură RISC.

7.2. Generare codului pentru calculatoare cu regiştri

Modelul pe care se bazează generarea de cod în această situaţie se numeşte ***maşina de regiştri*** [GBJL02] şi constă într-o memorie pentru stocarea variabilelor, un set de regiştri pentru execuţia operaţiilor şi două tipuri de instrucţiuni: pentru încărcarea şi salvarea valorilor între memorie şi regiştri şi pentru efectuarea operaţiilor din codul intermediar. Formatul instrucţiunilor este mai complex decât în cazul maşinii de stivă, deoarece trebuie specificate locaţia operanzilor, a rezultatului şi registrul în care se încarcă o anumită valoare. Pentru încărcarea valorii v în registrul R, instrucţiunea are formatul **LOAD v,R**, iar pentru salvarea în memorie a valorii v din registrul R instrucţiunea este **STORE R,v**. Pentru efectuarea operaţiilor unul din regiştri care conţine un operand este folosit pentru a reţine şi rezultatul: **ADD R1,R2** are semnificaţia valoarea din $R1$ este valoarea iniţială din $R1$ adunată cu valoarea din registrul $R2$.

Exemplul 7.2. Pentru instrucţiunea $c := a+b*3$ maşina de stivă necesită următoarea secvenţă de instrucţiuni:

```
Load b,R1
Load 3, R2
Mul R1,R2
Load a,R2
Add R1,R2
Store R1,c
```

Folosirea acestui model pentru calculatoarele cu regiştri generali necesită stocarea anumitor informaţii legate de regiştri şi variabile:

- Un registru poate fi disponibil (să conţină o valoare nesemnificativă) sau ocupat, dacă conţine valoarea uneia sau mai multor variabile. Pentru aceasta, vom defini o mulţime $VAR(R)$ asociată fiecărui registru, care reprezintă mulţimea variabilelor a căror valoare se află în registrul R.
- Este necesar să se precizeze pentru fiecare variabilă locul (registru, stivă sau memorie) în care se află valoarea curentă a variabilei. Vom defini mulţimea $MEM(x)$ pentru fiecare variabilă,reprezentând

mulţimea locaţiilor în care se află la un moment dat valoarea variabilei x. MEM va fi un câmp în tabela de simboluri.

Exemplul 7.3. Pentru instrucţiunea $F := A * B - (C + B) * (A + B)$ codul intermediar şi obiect corespunzător, împreună cu starea mulţimilor VAR şi MEM asociate sunt:

1. T1:=A*B
2. T2:+C+B
3. T3:=T2*T1
4. F:=T1-T3

Cod intermediar	Cod obiect	VAR	MEM
		VAR(R0)={} VAR(R1)={}	
(1)T1:=A*B	LOAD A, RO MUL B, R0	VAR(R0)={T1}	MEM(T1)={R0}
(2)T2:=C+B	LOAD C, R1 ADD B,R1	VAR(R1)=T2	MEM(T2)={R1}
(3)T3:=T2*T1	MUL R0 ,R1	VAR(R1)={T3}	MEM(T2)={} MEM(T3)={R1}
(4)F:=T1-T3	SUB R1, R0 STORE R0,F	VAR(RO)={F} VAR(R1)={}	MEM(T1)={} MEM(F)={R0} MEM(F)={RO,F}

Înaintea trecerii acestui fragment de cod intermediar la cod obiect se ştie că există doi regiştri disponibili, $R0$ şi $R1$. Pentru prima instrucţiune se încarcă valoarea lui A în registrul $R0$ şi valoarea din registru se înmulţeşte cu B, obţinându-se valoarea $T1$. Se actualizează mulţimea $VAR(R0)$ cu numele $T1$ şi mulţimea $MEM(T1)$ cu registrul $R0$. Codul obiect corespunzător instrucţiunii (3) conţine o optimizare, pe baza observaţiei că rezultatul instrucţiunii anterioare aflate în registru devine operand în această instrucţiune şi astfel nu mai e nevoie de încărcarea valorii. După descărcarea registrului în memorie, mulţimea MEM trebuie actualizată şi registrul respectiv poate deveni disponibil.

7.3. Alocarea regiştrilor

Alocarea regiştrilor are ca scop selectarea variabilelor care trebuie încărcate în regiştri. În timp ce numărul variabilelor dintr-un program este

relativ mare, numărul regiştrilor disponibili într-un calculator este limitat. De aceea un registru va fi obligat să stocheze mai multe valori pe parcursul execuţiei, în funcţie de anumite condiţii. O abordare optimală a gestiunii variabilelor în regiştri ia în calcul apariţiile variabilelor respective în program. Dacă o variabilă nu este folosită la un moment dat atunci ocuparea unui registru de către valoarea ei nu se justifică. Alocarea şi gestionarea regiştrilor se bazează pe variabilele folosite la un moment dat. O variabilă este activă dacă este posibil ca valoarea ei să fie necesară în viitor.

Definiţia 7.1. *[MJ81] O variabilă v este* ***activă*** *într-un punct de program p dacă:*

- *v a fost definită într-o instrucţiune anterioară lui p pe orice drum;*
- *v poate fi folosită de o instrucţiune s şi există un drum de la p la s;*
- *v nu e distrusă între p şi s.*

O variabilă v este activă între punctul p_i imediat următor definirii sale şi punctul p_j imediat după ultima ei folosire. Intervalul $[p_i, p_j]$ este ***intervalul activ*** al variabilei v.

Exemplul 7.4. Să considerăm următoarea secvenţă de cod intermediar:

$$
\begin{array}{lll}
(1) & T1 & = x + 4 \\
(2) & T2 & = T1 * 3 \\
(3) & T3 & = T1 - y \\
(4) & T4 & = T1 * T2 \\
(5) & T5 & = T2 - T3 \\
(6) & T6 & = T4 + T5
\end{array}
$$

Intervalul activ pentru $T1$ este $[1, 4]$, pentru $T2$ este $[2, 5]$, iar pentru $T4$ este $[4, 6]$.

Determinarea numărului minim de regiştri necesari acestui bloc de bază va eficientiza alocarea, astfel încât să se evite descărcarea valorilor în memorie, urmată de încărcarea lor. Pentru aceasta se determină intervalele active ale tuturor variabilelor şi se găseşte instrucţiunea cea mai "încărcată" (care apare în cele mai multe intervale active). Pentru exemplul considerat, după cum se observă în figura 7.1, obţinem: pentru instrucţiunea (4) avem patru variabile active, $T1, T2, T3, T4$, la fel

ca pentru instrucţiunea (5) - $T2, T3, T4, T5$. Deci va fi nevoie de patru regiştri pentru secvenţa de cod intermediar considerată. Pentru execuţia instrucţiunii (4) se vor aloca patru regiştri. După terminarea ei, registrul care conţine variabila $T1$ va fi descărcat, devenind disponibil şi putând fi folosit pentru încărcarea valorii $T5$ din instrucţiunea (5).

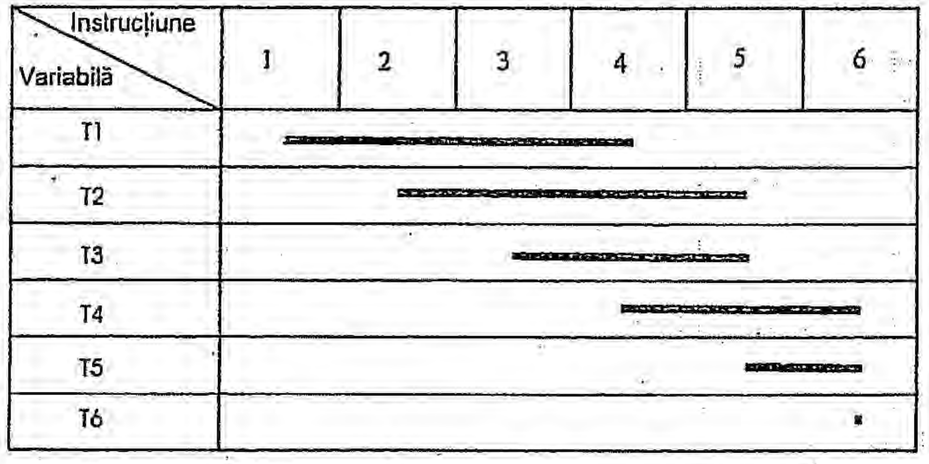

Figura 7.1: Intervalele active ale variabilelor

Algoritmii de alocare a regiştrilor cei mai performanţi folosiţi în implementarea compilatoarelor se bazează pe versiuni ale algoritmului de colorare a grafurilor.

7.3.1. Alocarea regiştrilor bazată pe colorarea grafurilor

Algoritmul porneşte de la graful de interferenţă al regiştrilor, care conţine informaţii relative la variabilele active dintr-un program.

Definiţia 7.2. *[Cha82] Un graf de interferenţă al regiştrilor este un graf neorientat în care:*

- *nodurile reprezintă variabile (identificatori sau nume temporare) care necesită alocare;*
- *muchiile reprezintă interferenţele: există o muchie între v_1 şi v_2 dacă există o instrucţiune în program în care variabilele v_1 şi v_2 să fie active simultan. În acest caz spunem că variabilele v_1 şi v_2 interferează.*

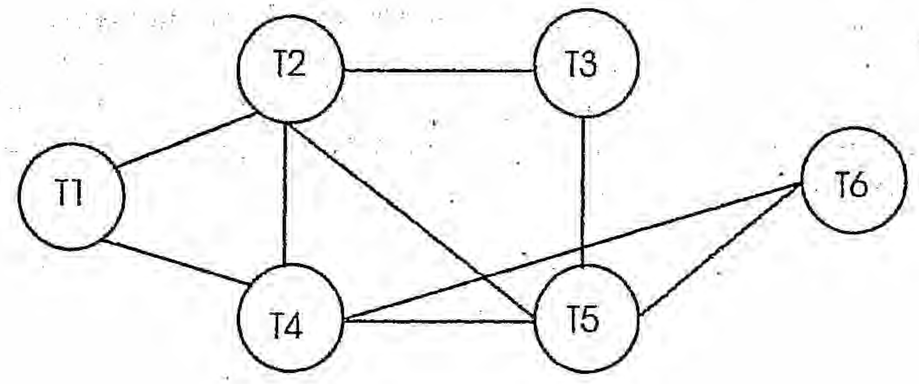

Figura 7.2: Graful de interferenţă asociat variabilelor active

Graful de interferenţă corespunzător codului intermediar considerat în exemplul 7.4 este prezentat în figura 7.2.

Definiţia 7.3. *[Tom81] Un graf este **k-colorabil** dacă fiecărui nod din graf i se poate asocia una din cele k culori astfel încât orice pereche de noduri adiacente (legate printr-o muchie) are culori distincte.*

Problema k-colorării grafului de interferenţă poate fi transpusă direct în alocarea regiştrilor, mapând fiecare culoare la câte un registru. Proprietatea de colorare asigură că nu există două variabile care să interfereze între ele şi care să fie alocate unui aceluiaşi registru.

Din păcate, problema determinării k-colorării unui graf neorientat este NP-completă pentru $k \geq 3$ [Tom81]. Un algoritm euristic reuşeşte să rezolve problema colorării într-un timp aproape liniar, pornind de la următoarea proprietate:

Teorema 7.1. *[CLR00] Dacă G este un graf neorientat şi x un nod în graf cu grad(x)[1] < k atunci G este k-colorabil dacă G' este k-colorabil, unde G' este graful obţinut din graful G în care se elimină nodul x şi toate muchiile asociate lui x.*

Se obţine astfel un graf mai mic, care fie este k-colorabil, fie poate fi din nou transformat pe baza proprietăţii de mai sus.

Algoritmul 7.1, după Chaitin [Cha82], foloseşte o stivă (notată s) pentru a memora muchiile din graf împreună cu o etichetă *et* astfel:

et=0 dacă nodul nu produce un impas în colorare (gradul nodului < k);

[1] grad(x)= numărul de muchii care trec prin x

et=1 dacă numărul de culori este insuficient pentru colorarea nodului şi a vecinilor săi.

Vom nota prin $vecini(v)$ mulţimea vecinilor nodului v, adică acele noduri cu care v este unit printr-o muchie.

Liniile [3-13] ale algoritmului procesează nodurile din graf, depunând în stivă muchiile lui v împreună cu eticheta lor. Următoarele linii [14-27] realizează procesarea nodurilor: dacă nodul are mai puţin de k vecini atunci $atrib(R, v)$ atribuie un registru R lui v cu condiţia ca R să nu fie asignat nici unui vecin al lui v, apoi se inserează nodul v în graf împreună cu toate muchiile sale ($insert(G, v)$). Dacă nodul are k sau mai mulţi vecini atunci ar trebui executată o transformare a grafului, conform teoremei 7.1. Liniile [20-22] propun o optimizare, verificând în primul rând dacă există registru disponibil şi abia în caz contrar marchează nodul v pentru a fi scos din graf. Procedura $alege(v, G)$ selectează un nod din cele rămase într-o anumită ordine, în funcţie de anumiţi parametri estimativi. Marcarea nodurilor are scopul de a decide care variabile să fie descărcate din regiştri. Criteriul propus de Chaitin [Cha82] recomandă alegerea nodului cu cel mai mic raport cost al descărcării/grad.

Principalele avantaje ale acestui algoritm sunt: viteza de execuţie (timp liniar relativ la dimensiunea grafului) şi faptul că oferă o soluţie pentru cazul în care graful nu este colorabil. Datorită faptului că se bazează pe o abordare euristică, algoritmul nu garantează o colorare minimală. Modul de implementare a euristicii pentru problema k-colorării şi pentru a decide descărcarea (care registru şi când) au dus la îmbunătăţiri ale algoritmului iniţial al lui Chaitin:

- îmbunătăţire a algoritmului lui Chaitin bazată pe observaţia că algoritmul execută descărcări în unele cazuri în care mai există regiştri disponibili. Îmbunătăţirea constă în luarea deciziei de descărcare doar atunci când ea apare (nu mai există regiştri disponibili) şi nu la construirea stivei [BCKT89];

- colorare bazată pe priorităţi [CH84]: se asociază priorităţi în colorarea nodurilor, în funcţie de unele estimări asupra alocării cantitative a regiştrilor. Procesul de colorare devine liniar, faţa de cazul algoritmului general de colorare care era exponenţial;

- colorare ierarhică a grafului, în care ierarhia este dată de prioritatea regiştrilor [CK91].

Algoritmul 7.1 Alocarea regiştrilor prin colorarea grafului de interferenţă

```
 1: INPUT: G - graful de interferenţă; k = nr. regiştri
 2: OUTPUT: alocare registru sau marcare nod în graf
 3: repeat
 4:    while ∃v : grad(v) < k do
 5:       push(v,vecini(v),0)   {pune în stivă muchiile care trec prin v}
 6:       sterge(G,v)   {şterge v şi toate muchiile sale din graful G}
 7:    end while
 8:    if G ≠ vid then
 9:       alege(G,v)
10:       push(v,vecini(v),1)
11:       sterge(G,v)
12:    end if
13: until G = vid
14: while S ≠ vidă do
15:    pop(v,vecini(v),et);
16:    if et=0 then
17:       atrib(R,v);
18:       insert(G,v)
19:    else
20:       if ∃ R neasignat vecini(v) then
21:          atrib(R,v);
22:          insert(G,v)
23:       else
24:          marcare(v);
25:       end if
26:    end if
27: end while
```

7.4. Selectarea instrucţiunilor

Optimizările care se aplică în generarea codului urmăresc îmbunătăţirea performanţei compilatorului din două puncte de vedere: lungimea programului şi numărul de regiştri folosiţi. Dacă alocarea regiştrilor prezentată în paragraful anterior se concentra doar asupra regiştrilor, această metodă de selectare a instrucţiunilor se adresează în primul rând asupra ordinii de execuţie a instrucţiunilor maşină, dar reuşeşte, în unele cazuri, să îmbunătăţească şi alocarea regiştrilor, şi anume numărul de încărcări şi descărcări ale lor.

Tehnica propusă analizează codul intermediar, reprezentat ca arbore abstract şi se bazează pe modificarea ordinii în care se execută instrucţiunile, bineînţeles asigurând echivalenţa semantică a acestor modificări (chiar dacă ordinea de evaluare se schimbă, rezultatul rămâne acelaşi). Pentru a înţelege intuitiv această metodă considerăm următorul exemplu.

Exemplul 7.5. Fie codul cu trei adrese corespunzător expresiei $b \times b - (4 - a \times c)$:

(1) $T1 = b \times b$
(2) $T2 = a \times c$.
(3) $T3 = 4 - T2$
(4) $T4 = T1 - T3$

şi arborele abstract corespunzător din figura 7.3. O selectare a instrucţiunilor corespunzătoare unei parcurgeri secvenţiale a codului cu trei adrese generează următoarea secvenţă de cod obiect, presupunând că avem 2 regiştri disponibili pentru efectuarea operaţiilor:

(1) *load* $b, R0$
(2) *mul* $b, R0$
(3) *load* $a, R1$
(4) *mul* $c, R1$ {nu există registru disponibil}
(5) *store* $R0, T1$ {se descarcă registrul R0}
(6) *load* $4, R0$
(7) *sub* $R1, R0$ {registrul R1 e disponibil}
(8) *load* $T1, R1$
(9) *sub* $R0, R1$

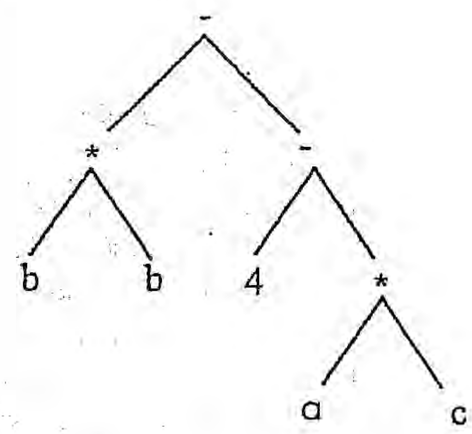

Figura 7.3: Arborele de sintaxă abstractă pentru expresia $b \times b - (4 - a \times c)$

(10) *store* $R1, T4$

Avem în total 10 instrucţiuni, ordinea de calcul a variabilelor temporare fiind $T1, T2, T3, T4$ şi e necesară o operaţie de descărcare a regiştrilor, din lipsă de spaţiu.

O altă parcurgere a arborelui corespunzător acestei expresii evidenţiază o altă ordine de evaluare, $T2, T3, T1, T4$, codul obiect fiind:

(1) *load* $a, R0$
(2) *mul* $c, R0$
(3) *load* $4, R1$
(4) *sub* $R0, R1$
(5) *load* $b, R0$
(6) *mul* $b, R0$
(7) *sub* $R1, R0$
(8) *store* $R0, T4$

După cum se observă, codul obiect conţine doar 8 instrucţiuni şi nu e nevoie de descărcări ale regiştrilor.

Problema selectării optimale a instrucţiunilor se poate formula în a găsi o ordine de vizitare a tuturor nodurilor arborelui, care să minimizeze

numărul de operaţii din codul obiect. Există două modalităţi de stabilire a acestei ordini:

- un algoritm greedy: se pleacă de la rădăcina arborelui şi se selectează subarborele cu număr maxim de noduri, se vizitează apoi descendenţii recursiv după acelaşi criteriu până se ajunge la frunze;
- un algoritm de programare dinamică ascendent, care lucrează dinspre frunze înspre rădăcină, atribuind o etichetă sau un cost nodurilor din arbore. Algoritmul lui Sethi - Ullman [SU70] se bazează pe această abordare.

7.4.1. Algoritmul lui Sethi – Ullman

Algoritmul 7.2 [SU70] se bazează pe a atribui fiecărui nod o etichetă, iar selectarea instrucţiunilor se bazează pe acestă etichetare. Etichetarea se face printr-o parcurgere de la frunze spre rădăcina arborelui astfel: frunzele sunt etichetate cu 0 (pentru cel mai din stânga descendent al unui părinte comun) şi cu 1 ceilalţi descendenţi, iar pentru nodurile interioare se aplică următoarea formulă: fie $c_1, ..., c_k$ descendenţii nodului a, atunci

$$eticheta(A) = max_{1 \leq i \leq k}(eticheta(c_i) + i - 1)$$

Algoritmul 7.2 Algoritmul Sethi-Ullman de etichetare a arborelui abstract

```
INPUT: arborele abstract
OUTPUT: arborele abstract etichetat
if n este frunză then
   if n e cel mai din stâga descendent al unui nod then
      eticheta(n) = 1
   else
      eticheta(n) = 0
   end if
else
   FIE c_1, ..., c_k descendenţii nodului n, în ordinea etichetei
   eticheta(n) = max_{1≤i≤k}(eticheta(c_i) + i - 1)
end if
```

Pentru arborele abstract din exemplul 7.5, algoritmul de etichetare generează valorile din figura 7.4.

Sethi şi Ullman [SU70] folosesc un algoritm *Gen_Cod* pentru generarea codului, având ca intrare arborele etichetat, interpretând eticheta unui nod ca numărul de regiştri necesari subarborelui cu rădăcina în acel nod.

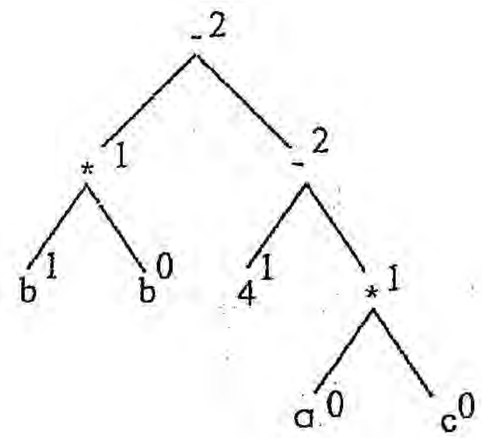

Figura 7.4: Etichetarea arborelui conform algoritmului 7.2

Sunt folosite două stive cu următoarea semnificaţie:

- *rstiva* - o stivă a regiştrilor, care la început conţine toţi regiştrii notaţi $R_0, R_1, ..., R_{r-1}$ (r fiind numărul total de regiştri disponibili); procedura de generare de cod nu modifică ordinea regiştrilor în stivă;
- *tstiva* - o stivă pentru alocarea locaţiilor temporare de memorie.

Algoritmul apelează proceduri de lucru asupra stivelor: *push(stiva,a)* care introduce elementul a în *stiva*, *pop(stiva)* care scoate un element din stivă şi *swap(stiva)* care interschimbă primele două elemente din stivă. Această ultimă procedură este necesară pentru a garanta că fiul stâng şi nodul părinte sunt evaluaţi în acelaşi registru.

Dacă nodul curent este o frunză atunci se va genera doar o instrucţiune de încărcare a operandului notat de nod în registru (liniile 3-4). Dacă este nod interior în arbore atunci avem una din următoarele situaţii în algoritm:

1. în liniile 6-10, un subarbore în care fiul drept este un operand, deci se va genera cod numai pentru fiul stâng (evaluarea lui n_1) în registrul R, urmată de instrucţiunea $n\ n_2, R$ (unde n este o operaţie);
2. în liniile 11-19, când nodul este rădăcina unui subarbore în care n_1 poate fi evaluat direct (fără memorarea evaluărilor intermediare), dar n_2 e mai dificil de evaluat. Pentru acest caz se interschimbă regiştrii din vârful stivei, se execută evaluarea lui n_2 în capul stivei R, se scoate R din stivă, apoi se evaluează n_1 şi se generează instrucţiunea $n\ R, S$, aducând apoi stiva la configuraţia dinaintea acestui apel;

Algoritmul 7.3 Algoritmul Gen_cod – algoritmul Sethi Ullman de generare de cod

```
1: INPUT: arborele etichetat
2: OUTPUT: cod obiect
3: if (n e frunză) AND (e cel mai din stânga descendent) then
4:    print 'LOAD', n, ',', cap(rstiva)
5: else
6:    if n e nod interior, fiul stânga e n1, fiul dreapta n2 then
7:       if eticheta(n2)=0 then
8:          Gen_Cod(n1);
9:          print n, n2,',',cap(rstiva)
10:      end if
11:   else
12:      if (1 ≤ eticheta(n1) < eticheta(n2)) AND (eticheta(n1) ≤ r)
         then
13:         swap(rstiva);
14:         Gen_Cod(n2);
15:         R = pop(rstiva);
16:         Gen_Cod(n1);
17:         print n, R, ',' , cap(rstiva);
18:         push(rstiva,R);
19:         swap(rstiva);
20:      else
21:         if (1 ≤ eticheta(n2) ≤ eticheta(n1)) AND (eticheta(n2) ≤ r)
            then
22:            Gen_Cod(n1);
23:            R = pop(rstiva);
24:            Gen_Cod(n2);
25:            print n, cap(rstiva), ',' , R;
26:            push(rstiva,R);
27:         else
28:            Gen_Cod(n2);
29:            T:= pop(tstiva);
30:            print 'STORE', cap(rstiva),',',T;
31:            Gen_Cod(n1);
32:            push(tstiva,T);
33:            print n, T, ',', cap(rstiva);
34:         end if
35:      end if
36:   end if
37: end if
```


3. în liniile 20-26, un caz similar celui precedent: n_2 se poate evalua direct, iar n_1 este mai dificil de evaluat; din punctul de vedere al algoritmului, situaţia e mai simplă deoarece nu necesită interschimbarea regiştrilor în stivă;

4. în liniile 27-33, situaţia în care ambii subarbori necesită mai mult de r regiştri, caz în care trebuie folosite variabile temporare, iar evaluarea se face în preordine: subarbore stâng, subarbore drept, rădăcină.

Exemplul 7.6. Pentru arborele etichetat din figura 7.4, aplicarea algoritmului generează următoarea secvenţă de apeluri şi tipăriri. Faţă de codul din exemplul 7.5 ordinea diferă, dar se generează aceleaşi instrucţiuni şi acelaşi efect.

```
Gen_Cod(T4)
   Gen_Cod(T3)
      Gen_Cod(4)        LOAD 4,R1
      Gen_Cod(T2)
         Gen_Cod(c)     LOAD c,R0
         MUL a,R0
      SUB R0, R1
   Gen_Cod(T1)
      Gen_Cod(b)        LOAD b,R0
      MUL b,R0
   SUB R1,R0
```

7.5. Forma codului obiect

Rezultatul fazei de generare de cod obiect constă în acest moment într-o secvenţă de instrucţiuni maşină într-un anumit format decis de autorul compilatorului, dar care nu are formatul binar uzual fişierelor obiect (*.obj), ceea ce indică faptul că mai este necesară o conversie a codului maşină la formatul codului obiect. Cu toate că această conversie este uşurată de optimizările aplicate - alocarea regiştrilor, selectarea instrucţiunilor - ea reprezintă o operaţie complicată, deoarece necesită specificarea multor şabloane de biţi. De aceea, majoritatea compilatoarelor preferă să evite această situaţie, alegând una din următoarele opţiuni:

- Asamblor: majoritatea sistemelor de operare posedă un asamblor integrat, care transformă instrucţiuni din cod maşină în cod obiect. Prin generarea codului de asamblare ca şi cod rezultat se evită partea

de generare a formatului binar al codului obiect, chiar dacă procesul de compilare durează mai mult (se adaugă etapa de asamblare). Costul acestei asamblări este compensat de faptul că se evită scrierea de cod pentru generarea formatului binar, care e cunoscută ca generatoare de erori.

- Compilatorul unui alt limbaj: în acest caz are loc de fapt o traducere a programului sursă în program scris în alt limbaj de programare, ceea ce se poate realiza direct din arborele atributat, fără a mai genera cod intermediar.

Compilatoarele cele mai folosite în astfel de scopuri sunt cele ale limbajelor C şi C++, fiind considerate cele mai optimale (execută optimizări performante ale codului) şi sunt vast răspândite: există o varietate mare de compilatoare C şi C++ pentru diverse platforme şi sisteme de operare. Acesta este şi motivul pentru care limbajul C este considerat "limbaj de asamblare independent de maşină".

Această abordare este întâlnită în mod deosebit la compilatoarele pentru limbaje ne-imperative (funcţionale, logice, distribuite), dar nu numai - primele compilatoare de C++ generau cod C. Folosirea limbajului C ca limbaj rezultat încetineşte substanţial, uneori chiar dublează, timpul de compilare.

8. TRATAREA ERORILOR

Una din sarcinile de bază ale unui compilator este aceea de a valida programul în raport cu definiţia limbajului, astfel ca el să detecteze, să clasifice şi să emită mesaje corespunzătoare pentru fiecare eroare. Dacă se detectează o eroare atunci compilatorul nu va mai funcţiona normal şi nu se va genera cod obiect. Există compilatoare (de exemplu, Turbo Pascal) care se opresc la prima eroare întâlnită sau altele (Borland C) care continuă analiza lexicală şi sintactică, furnizând o listă completă a acestor erori.

```
; expected
type mismatch
unknown identifier
THEN expected
```

sunt doar câteva erori de compilare pe care un compilator Pascal le emite. Pe lângă acestea mai pot apărea erori în timpul execuţiei, unele mai uşor de detectat, altele necesitând întreruperea forţată a execuţiei.

Tratarea erorilor presupune cel puţin detectarea acestora. Este cea mai simplă soluţie din punctul de vedere al compilatorului, dar total ineficientă pentru programator: imaginaţi-vă un compilator căruia furnizându-i un program, veţi primi un mesaj "Există o eroare". Un astfel de compilator este neperformant, fiind sarcina programatorului de a căuta şi corecta eroarea, operaţie care necesită mult timp. De aceea, cerinţa minimă pentru tratarea erorii este detectarea şi localizarea ei: în ce linie, la ce construcţie (atom) s-a detectat eroarea şi preferabil, un diagnostic al ei.

O abordare mai pretenţioasă a tratării erorilor presupune o continuare a analizei în unul din următoarele moduri:

- ieşire din zona de eroare, chiar dacă se neglijează celelalte elemente din zonă, şi continuare cu restul programului sursă - *recuperare din eroare*;
- încercare de modificare a zonei de eroare pentru ca aceasta să devină corectă - *reparare a erorii*;
- efectuarea unei presupuneri asupra erorii, în funcţie de context, modificând textul sursă - *corectare a erorii*.

Toate compilatoarele realizează detecţia şi diagnosticul erorilor, multe dintre ele realizează şi recuperarea, repararea este implementată doar într-un număr foarte mic de compilatoare, iar corectarea este deocamdată

mai mult un deziderat teoretic, neexistând compilatoare care să realizeze această operaţie.

Mesajele de eroare sunt destul de multe la număr, iar cauzele care le-au generat sunt multiple: există posibilitatea ca un mesaj de eroare să fie emis pentru două tipuri diferite de erori. Erorile detectabile la compilare se pot clasifica în:

- erori lexicale;
- erori sintactice;
- erori semantice.

Erorile lexicale şi semantice sunt considerate locale - ele pot fi reparate local, acolo unde apar, fără a necesita o parcurgere înainte sau înapoi a zonei în care a apărut. Erorile sintactice sunt considerate globale, repararea unei erori putând necesita modificarea unui alt atom decât cel care a declanşat procedura de tratare a erorii.

Exemplul 8.1. Secvenţa *a = b-1 then ...* generează o eroare în momentul în care e analizat atomul *then*, deşi greşeala se află la începutul secvenţei şi repararea va consta în inserarea atomului *if* la începutul secvenţei.

Erorile semantice pot fi detectate doar parţial la compilare, unele dintre ele putând fi detectate doar la execuţie. Verificările introduse în faza de analiză semantică relative la tipuri şi declaraţii pot detecta unele dintre aceste erori, dar recuperarea se poate face doar ţinând cont de contextul erorii. Dintre cele mai des întâlnite erori semantice, care pot fi detectate în momentul compilării amintim: nedefinirea sau definirea multiplă a unui identificator, utilizarea incorectă a unei variabile (din punct de vedere al tipului), index al unei variabile indexate în afara domeniului definit.

8.1. Tratarea erorilor în analiza lexicală

Conform algoritmului general al analizei lexicale, singurul tip de eroare care poate apărea în această fază a compilării este cazul în care un atom lexical nu poate fi clasificat. În funcţie de abilitatea de a recupera eroarea de către analizorul sintactic, mesajul de eroare poate fi diferenţiat.

Dacă proiectarea compilatorului presupune că analizorul lexical este comandat de către analizorul sintactic, atunci tratarea erorilor se poate face mai uşor: se poate cere ca următorul atom să aparţină unei anumite clase, putând astfel raporta tipul erorii şi eventuale sugestii de reparare.

Rutina de tratare a erorii din cadrul analizorului lexical poate emite un diagnostic în funcţie de atomul la care s-a detectat eroarea: dacă un identificator nu respectă regula lexicală asociată - nu începe cu literă sau depăşeşte lungimea maximă admisă; dacă o constantă reală conţine mai mult de un punct sau caracter nepermis (când se detectează un caracter care nu aparţine alfabetului limbajului).

8.2. Tratarea erorilor în analiza sintactică

Datorită folosirii gramaticilor independente de context ca fundament teoretic pentru această fază şi a metodelor de analiză sintactică predictive (LL şi LR), detectarea erorilor sintactice şi diagnosticarea lor este uşor de implementat. Tabelele de analiză folosite permit ca decizia apariţiei erorii să fie pusă pe seama faptului că nu există nici o tranziţie a analizorului dintr-o anumită stare. Recuperarea din eroare constă în presupunerea de către analizor a unei configuraţii din care se poate continua analiza, cu un număr minim de modificări asupra elementelor care formează configuraţia: banda de intrare, stiva, starea analizorului şi banda de ieşire.

Strategiile de tratare a erorilor sintactice urmează una din următoarele situaţii:

- *"Mod de panică"*: analiza sintactică nu mai continuă conform algoritmului, intră într-un mod de panică şi fie ignoră şirul de intrare până când găseşte un atom important (cuvânt rezervat), fie scoate din stiva de lucru atomi până când atomul e acceptat. Această metodă e uşor de implementat şi rapidă, deoarece nu necesită spaţiu suplimentar, dar are dezavantajul că fie se poate ignora o parte considerabilă din intrare, fie se pierde informaţia semantică a elementelor scoase din stivă.

- *Codificarea erorilor în tabelele de analiză*: presupune o analiză a frecvenţelor tipurilor de erori şi în funcţie de aceasta, locaţia de tabel corespunzătoare unei erori este înlocuită cu apelul unei proceduri de tratare a erorii. Din păcate este dificil de prevăzut toate cazurile posibile de erori şi în plus, se adaugă mult cod pentru tratarea fiecărui tip de eroare.

- *Extinderea gramaticii cu producţii de eroare.*

- *Metode independente de limbaj* [Roh80].

8.2.1. Tratarea erorilor în analiza sintactică LL(1)

Cea mai cunoscută metodă de recuperare a erorii în analiza sintactică LL(1) este *metoda mulţimii acceptabile* [GBJL02], care constă în următorii paşi:

1. Determinarea mulţimii acceptabile - se poate folosi ca mulţime acceptabilă mulţimea FOLLOW [Har77, Pem80].

2. Omiterea simbolurilor inacceptabile: zero sau mai multe simboluri de pe banda de intrare sunt omise, până când se găseşte un simbol care aparţine mulţimii acceptabile (deoarece simbolul de sfârşit de secvenţă $ este întotdeauna acceptabil, acest pas se va termina).

3. Resincronizarea analizorului sintactic: continuarea analizei sintactice se face cu modificări: mai întâi se încearcă tranziţiile *push* şi *pop*. Dacă acestea reuşesc, atunci analizorul sintactic a revenit într-o stare obişnuită, poate continua normal. Dacă tranziţiile eşuează, atunci:

   - dacă în vârful stivei de lucru este un simbol neterminal, se va alege cea mai scurtă alternativă de retranscriere a lui;
   - dacă în vârful stivei de lucru este un simbol terminal, se va insera această predicţie.

Pasul 3 se va repeta până când o tranziţie normală va reuşi.

8.2.2. Tratarea erorilor în analiza sintactică LR(k)

Literatura de specialitate indică o preocupare deosebită în tratarea erorilor pentru acest tip de analiză sintactică. Caracteristicile metodei de analiză permit diferite strategii de tratare aplicabile asupra stivei de lucru şi a stivei de intrare, în funcţie de starea analizorului şi de simbolul curent. Vom încerca prezentarea succintă a câtorva astfel de contribuţii:

Algoritmul lui Modry [MM78]: lucrează în două faze numite: *condensare* şi *corectare*. Presupunând că eroarea a apărut în starea s şi simbolul curent de pe banda de intrare este x_1 (deci nu există nici o acţiune definită pentru această configuraţie), în faza de *condensare* se determină mulţimea de stări pentru care x_1 are o acţiune de deplasare definită, notată $P(x_1)$. Astfel stiva de lucru va deveni:

$...s\ ?\ p\ x_1\ p_1$

unde '?' este simbolul introdus de eroare în stivă.

Această presupunere poate conduce la două situaţii posibile:

- fie se va putea efectua la un moment dat o reducere peste punctul de eroare, ceea ce determină un "candidat la corecţie";
- fie se generează o nouă situaţie de eroare, obţinându-se un "candidat în aşteptare".

Faza de *corectare* preia candidaţii la corecţie şi se încearcă continuarea analizei cu acea presupunere, ieşind din zona de eroare. Dacă nici un candiat la corecţie nu încheie analiza cu succes, atunci se iau în considerare candidaţii în aşteptare şi întregul algoritm de tratare a erorii este apelat pentru această a doua eroare.

Metoda are două dezavantaje importante: ia în considerare prea multe posibilităţi de corectare a unei erori şi nu e suficient de clar când algoritmul ar trebui să se oprească din tentativele de corectare.

Algoritmul lui Graham [GHJ79] consideră două etape de aplicare a unor strategii de corectare:

- etapa 1: constă în a încerca inserarea, ştergerea sau modificarea unui singur atom; dacă nici o astfel de operaţie nu este posibilă atunci se aplică a doua etapă;
- etapa 2: este o strategie "mod de panică" ce determină o formă propoziţională greşită, dar asemănătoare unei părţi drepte de producţie şi continuă analiza după ce s-a ieşit din zona formată de aceasta.

Metoda ia în considerare doar anumite tipuri de erori (erorile comune de programare) şi din acest punct de vedere este eficientă, dar nu va putea rezolva toate tipurile de erori. Avantajul principal îl constituie faptul că algoritmul nu se restrânge doar la erori sintactice, ci poate trata şi erorile lexicale şi semantice.

Algoritmul lui Mckenzie [MMD95] este un algoritm eficient de recuperare a erorii care returnează recuperarea cu cel mai mic cost local. Pentru aceasta, algoritmul foloseşte o coadă de configuraţii împreună cu costul asociat.

Presupunând că eroarea a apărut când analizorul este în starea q_1 şi simbolul curent de pe banda de intrare este x_1 se iau în considerare toate simbolurile x pentru care există acţiune: $actiune(q_1, x) \neq \emptyset$. Dacă acţiunea este de deplasare atunci configuraţia $(q_n...q_1 \uparrow x\ x_1...x_m)$ este depusă în coadă. Fiecare configuraţie are un cost asociat care se calculează în funcţie de inserările şi ştergerile necesare pentru a ajunge din această configuraţie în cea de eroare. Pentru cazul considerat costul este inserarea lui x. Coada se păstrează sortată după cost.

Dacă $actiune(q_1, x) = reducere$ atunci se execută o secvenţă de reduceri până când se ajunge într-o stare q_i care are definită o acţiune de

deplasare pentru x şi noua configuraţie $(q_n...q_i \uparrow x\, x_1...x_m)$ este depusă în coadă cu costul corespunzător.

După ce toate configuraţiile bazate pe inserare sunt formate, se încearcă ştergerea simbolului x_1 şi configuraţia $(q_n...q_1 \uparrow x_2...x_m)$ este depusă în coadă, iar costul ştergerii lui x_1 este adăugat costului total al acestei configuraţii.

După construirea tuturor configuraţiilor, se consideră configuraţia din capul cozii (cu cel mai mic cost) şi dacă nu este eronată (există acţiune pentru ea) atunci se returnează ca reparare. Altfel, algoritmul se aplică din nou pentru această configuraţie eronată şi toate configuraţiile nou generate se depun în coadă.

O caracteristică importantă a acestui algoritm este dată de căutarea în lăţime în mulţimea stărilor, care garantează returnarea configuraţiei optime de reparare. De asemenea, se asigură revenirea din orice eroare, datorită faptului că se iau în considerare toate inserările şi ştergerile posibile. Dezavantajul metodei constă în complexitatea mare a căutării în mulţimea configuraţiilor. Bertsch [BN99] propune o îmbunătăţire a acestui algoritm.

Recuperarea erorilor în *yacc*

Generatorul de analiză sintactică *yacc* dispune de propriul mecanism de recuperare a erorilor, care poate fi folosit scriind reguli destinate recunoaşterii atomului special `error`, care este un simbol terminal definit şi rezervat pentru tratarea erorilor. *Yacc* generează un atom `error` ori de câte ori apare o eroare sintactică. Prin asocierea unei reguli pentru recunoaşterea acestui atom în context, analiza sintactică poate evita zona de eroare şi continua, ca în următoarea regulă de producţie:

```
stmnts:  /* instrucţiune vidă*/
   |stmnts '\n'
   |stmnts exp '\n'
   |stmnts error '\n'
```

Recuperarea se face după următoarea strategie: în primul rând se scot din stivă stări şi simboluri până când se ajunge într-o stare în care atomul `error` este acceptabil, adică poate fi deplasat. În acest moment, dacă atomul care urma iniţial nu este acceptabil pentru a fi deplasat în continuare, analizorul citeşte simboluri de pe banda de intrare şi le ignoră până când găseşte un simbol acceptabil.

Alegerea regulilor de producţie pentru erori determină strategia de recuperare a erorilor. Cea mai simplă modalitate este de a ignora linia curentă sau instrucţiunea curentă la momentul detectării unei erori:


```
stmnt:  error ';' /*ignoră până la ';' */
```

Se poate întâmpla ca recuperarea unei erori să genereze o altă eroare imediat după ea. Pentru a preveni astfel de erori "în cascadă", *yacc* nu va emite mesaje de eroare pentru o eroare apropiată de cea iniţială, ci doar după ce trei simboluri consecutive au fost deplasate cu succes.

Macroul `YYRECOVERING` reprezintă o expresie care are valoarea 1 dacă analizorul se recuperează dintr-o eroare şi 0 altfel. Valoarea 1 indică faptul că mesajele de eroare sunt momentan inactive.

BIBLIOGRAFIE

[Aab] AABY A.- *Compiler Construction using Flex and Bison*, cs.wwc.edu./ aabyan/Linux/compiler.pdf

[AGT89] AHO A.V., GANAPATHI M., TJIANG S.W.K. - *Code generation using tree patern matching and dynamic programming*, ACM Trans, Programming Languages and Systems, vol.11, nr. 4, 1989, pag. 491-516

[AJ74] AHO. A.V., JOHNSON S.C. - *LR Parsing*, ACM Computing Surveys, vol.6, nr.2, 1974, pag. 99-124

[AJ76] AHO A.V., JOHNSON S.C. - *Optimal Code Generation for Expression Trees*, Journal of the ACM (JACM), v.23 n.3, 1976, pag.488-501

[ASU86] AHO A.V., SETHI R., ULLMAN J.D. - *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986

[AU72] AHO A.V., ULLMAN J.D. - *The Theory of Parsing, Translation and Compiling*, vol. I: *Parsing*, Prentice Hall, 1972

[AU73] AHO A.V., ULLMAN J.D. - *The Theory of Parsing, Translation and Compiling*, vol. II: *Compiling*, Prentice Hall, 1973

[Ait91] AÏT-KACI H. - *Warren's Abstract Machine - A Tutorial Reconstruction*, MIT Press, 1991

[App98] APPEL. A.W. - *Modern Compiler Implementation in C*, Cambridge University Press, 1998

[AS87] APPEL A.W., SUPOWIT K.J. - *Generalizations of the Sethi-Ullman algorithm for register allocation*, Software - Practice and Experince, vol.17, nr.6, 1987, pag. 417-421

[Bac78] BACKUS J. - *The history of FORTRAN, I, II, and III*, ACM SIGPLAN Notices, vol. 13, nr. 8, 1978, pag. 165-180

[BE76] BAUER F.L., EICKEL J.(ed.) - *Compiler Construction, An Advanced Course* - second edition, LNCS 21, Berlin, 1976

[BN99] BERTSCH E., NEDERHOF M.J. - *On Failure of the Prunning Technique* in "Error Repair in Shift-Reduce Parsers", ACM Trans. on Programming Landuages and Systems, vol. 21, nr.1, 1999, pag.

[Boi83] BOIAN F.M. - *Determinarea funcţiilor FIRST1, FOLLOW1 şi EFF1 folosind metode booleene*, Info Iaşi - lucrările celui de-al IV-lea Colocviu de Informatică, Univ. Al.I. Cuza Iaşi, 1983, pag. 813-192

[BCKT89] BRIGGS P., COOPER K.D., KENNEDY K., TORCZON L. - *Coloring Heuristics for Register Allocation*, Proc. of ACM Sigplan Conf. of Programming Language Design and Implementation, 1989, pag 275-284

[BMMR63] BROOKER R.A.,MACCALLUM I.R., MORRIS D., ROHL J.S. - *The compiler compiler*, Annual Review Automatic Programming, nr. 3, 1963, pag. 229-322

[BS76] BRUNO J., SETHI R. - *Code generation for a one register machine*, J.ACM, vol.23, nr.3, 1976, pag. 502-510

[CK91] CALLAHAN D., KOBLENZ B. - *Register Allocation via Hierarchical Graph-Coloring*, Proc. of ACM Sigplan Conf. of Programming Language Design and Implementation, Toronto, 1991, pag. 192-203

[CNL79] CATTELL R.G.G., NEWCOMER J.M., LEVERETT B.W. - *Code generation in a machine-independent compiler*, Proceedings of the 1979 SIGPLAN symposium on Compiler Construction, Denver, SUA, pag.65-75

[Cha82] CHAITIN G.J. - *Register allocation and spilling via graph coloring*, Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction, Boston, SUA, pag.98-101

[Cha87] CHAPMAN N.P. - *LR Parsing: Theory and Practice*, Cambridge University Press, 1987

[CH84] CHOW F., HENESSY J. - *Register Allocation by priority-based coloring*, Proc. ACM Sigplan Symposium on Compiler Construction, 1984, pag. 222-232

[CK77] COCKE J., KENNEDY K. - *An algorithm for reduction of operator strength*, Comm. ACM, vol.20, nr. 11, 1977, pag. 850-856

[CLR00] CORMEN T., LEISERSON C.E., RIVEST R.R. - *Introducere în algoritmi*, Computer Libris Agora, 2000

[DF84] DAVIDSON J.W., FRASER C.W. - *Automatic generation of peephole optimizations*, ACM Sigplan Notices, vol.19, nr. 6, 1984, pag. 111-116

[DW89] DAVIDSON J.W., WHALEY D.B. - *Quick compilers using peephole optimizations*, Software - Practice and Experience, vol. 19, nr. 1, 1989, pag. 79-97

[DQR95] DE DINECHIN F., QUINTON P., RISSET T. - *Structuration of the ALPHA language*, Programming Models for Massively Parallel Computers (PMMP '95), 1995, pag. 18

[DeR71] DEREMER F.L. - *Simple LR(k) grammars*, Communication of ACM, vol. 14, nr. 7, 1971, pag. 453-460

[DeR74] DEREMER F.L. - *Lexical analysis*, în Compiler Construction. An Advanced Course (ed. F.L. Bauer, J. Eickel), Springer - Verlag, 1974, pag. 109-120

[DNF79] DONEGAN M.K., NOONAN R.E., FEYOCK S. - *A code generator generator language*, Proceedings of the 1979 SIGPLAN symposium on Compiler construction, Denver, SUA, pag.58-64

[Eng84] ENGELFRIET J. - *Attribute grammars: attribute evaluation methods*, în Methods and Tools for Compiler Construction (ed. B. Lorho), Cambridge University Press, 1984, pag. 102-138

[For92] FORTES GALVEZ J. - *Generating LR(1) parsers of small size*, în Compiler Construction. 4th International Conference CC'92 (ed. U. Kastens P. Pfahler), Springer - Verlag, 1992, pag. 16-29

[GFH82] GANAPATHI M., FISCHER C.N., HENNESSY J.L. - *Retargetable Compiler Code Generation*, ACM Computing Surveys (CSUR), Vol. 14, nr. 4, 1982, pag. ?

[GHJ79] GRAHAM S.L., HALEY C.B., JOY W.N. - *Practical LR Error Recovery*, Proc. of SIGPLAN Symp. on Compiler Construction, ACM Press, 1979, pag. 168-175

[GBJL02] GRUNE D., BAL H.E., JACOBS C.J.H., LANGENDOEN K.G. - *Modern Compiler Design*, John Wiley, 2002

[GJ90] GRUNE D., JACOBS C.J.H. - *Parsing Techniques: a Practical Guide*, Ellis Horwood, 1990

[Har77] HARTMANN A.C. - *A Concurrent-Pascal compiler for microcomputers*, Sprin-ger-Verlag, 1977

[Hop69] HOPGOOD F.R.A. - *Compiling Techniques*, MacDonald: Londra, American Elsevier: New York, 1969

[JN61] JENSEN J., NAUR P. - *An Implementation of ALGOL 60 Procedures*, Nordisk-Tidskr Inofrm. Behand., vol. 1, nr. 1, 1961, pag. 38-47

[Joh74] JOHNSON S.C. - *Yacc- Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ, Rep.CSTR 32, 1974

[JAR68] JOHNSON W.L., ACKELY S.I, ROSS D.T. - *Automatic generation of efficient lexical processors using finite state techniques*, Comm. ACM, vol.11, nr.12, 1968, pag. 805-813

[KU77] KAM J.B., ULLMAN J.D. - *Monotone Data Flow Analysis Frameworks*, Acta Informatica 7,3, pag. 305-318, 1977

[Ken81] KENNEDY K. - *A survey of data flow analysis techniques*, în Program Flow Analysis (ed. S.S. Muchnick, N.D. Jones), Prentice HAll, 1981, pag. 5-54

[KR88] KERNIGHAN B.W., RITCHIE D.M. - *The C Programming Language*, Second Edition, Prentice-Hall, 1988

[Knu68] KNUTH D.E. - *Semantics of context-free languages*, Math. Syst. Theory, vol.2, nr. 2, 1968, pag. 127-145

[Knu71] KNUTH D.E. - *Semantics of context-free languages - correction*, Math. Syst. Theory, vol.5, nr. 1, 1971, pag. 95-96

[LLH71] LALONDE W.R., LEE E.S., HORNING J.J. - *A LALR(k) parser generator*, în IFIP Congress 71 (ed. C.V. Freiman), North-Holland, 1971, pag. 153-157

[LS75] LESK M.E., SCHMIDT E. - *LEX - A Lexical Analyzer generator*, Bell Laboratories, Murray Hill, NJ, Rep. CSTR 39, 1975

[Lev82] LEVERETT, B.W. - *Topics in code generation and register allocation*, Tech. Rep. CMU-82-130, Computer Science Dep., Carnegie-Mellon Univ., Pittsburgh, Pa., iulie 1982.

[LMB92] LEVINE J.R., MASON T., BROWN D. - *Lex and Yacc*, Second Edition, O'Reilly, 1992

[MPL84] MILOS D., PLEBAN U.F., LOEGEL G. - *Direct Implementation of Compiler Specifications or the Pascal P-code Compiler Revisited*, Proc. of POPL, 1984, pag. 196-207

[MAEH66] MCCARTHY J., ABRAHAMS P.W., EDWARDS J., HART T.P., LEVIN M.I. - *LISP 1.5 Programmer's Manual*, Second Edition, MIT Press, Cambridge, Mass., 1966

[MHB90] MCKENZIE B.J., HARRIES R., BELL T.C. - *Selecting a Hashing Algorithm*, Software, Pract. Exper., vol. 20, nr. 2, 1990, pag. 209-224

[MYD95] MCKENZIE B.J., YEATMAN C., DE VERE L. - *Error repair in shift-reduce parsers*, ACM Trans. Programming Languages and Systems, vol.17, nr.4, 1995, pag. 672-689

[MM78] MICKUNAS M.D., MODRY J.A. - *Automatic Error recovery for LR Parsers*, Comm. ACM, vol.21, nr.6, 1978, pag. pag. 459-465

[Mor84] MOREL E. - *Data Flow analysis and global optimization*, în Methods and Tools for Compiler Construction (ed. B. Lorho), Cambridge University Press, 1984, pag. 289-315

[Muc97] MUCHNICK S.S. - *Advanced Compiler Design and Implementation*, Academic Press, 1997

[MJ81] MUCHNICK S.S., JONES N.D. - *Program Flow Analysis*, Prentice Hall, 1981

[Nau60] NAUR P. (ed.) - *Revised Report on the Algorithmic Language ALGOL 60*, Communications of the ACM, Vol. 3 No.5, 1960, pag. 299-314

[Nau65] NAUR P. - *Checking of operand types in ALGOL compilers*, BIT 5/1965, pag. 151-163

[Noo85] NOONAN R.E. - *An algorithm for generating abstract syntax tree*, Computer Languages, vol.10, nr.3/4, 1985, pag. 225-236

[Pag77] PAGER D.A. - *A Practical General Method for Constructing LR(k) Parsers*, Acta Informatica 9, 1977, pag. 249-268

[Pem80] PEMBERTON S. - *Comments on an error-recovery scheme by Hartmann*, Software - Practice and Experience, 10(3)/ 1980, pag. 231-240

[Poo77] POONEN G. - *Error recovery for LR(k) parsers*, în Information Processing 77 (ed. B. Gilchrist), North-Holland, 1977, pag. 529-533

[Ric85] RICHTER H. - *Noncorrecting syntax error recovery*, ACM Trans. Programming Languages and Systems, vol.3, nr.3, 1985, pag. 478-489

[Roh80] RÖHRICH J. - *Methods for the automatic construction of error correcting parsers*, Acta Informatica vol.13, nr.2, 1980, pag. 115-139

[Sam76] SAMET H. - *Compiler testing via symbolic interpretation*, Proceedings of the annual ACM Conference, Houston, SUA, ACM Press, 1976, pag. 492-497

[SU70] SETHI R., ULLMAN J.D. - *The generation of optimal code for arithmetic expressions*, J.ACM, vol. 17, nr. 4, 1970, pag. 715-728

[She59] SHERIDAN P.B. - *The arithmetic translator-compiler of the IBM FORTRAN automatic coding system*, Comm.ACM, vol. 2, nr. 2, 1959, pag. 9-21

[Spe88] SPECTOR D. - *Efficient Full LR(1) Parser generation*, ACM Sigplan Notices 23, 12, 1988, pag. 143-150

[Sti85] STIRLING C.P. - *Follow set error recovery*, Software - Practice and Experience, vol.15, nr. 3, 1985, pag. 239-257

[Ser87] ŞERBĂNAŢI L.D. - *Limbaje de programare şi compilatoare*, Ed. Academiei RSR, 1987

[Tom81] TOMESCU I. - *Probleme de combinatorică şi teoria grafurilor*, Ed. Didactică şi Pedagogică, Bucureşti, 1981

[Wir71a] WIRTH N. - *The programming language PASCAL*, Acta Informatica, vol.1, nr.1, 1971

[Wir74b] WIRTH N. - *The design of a PASCAL compiler*, Software Practice and Experience, vol.1, nr.4, 1971

[Wir74] WIRTH N. - *PASCAL - User Manual and Report.* (cu Kathy Jensen), Springer-Verlag, 1974

[Bor] *** Borland - www.borland.com

[EBN] *** EBNF - standardul britanic - http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html (ISO/IEC 14977:1996(E))

[UCS] *** UCSD-Pascal: http://www.jacobsschool.ucsd.edu/Pascal/

[LY] *** Lex and Yacc Pages - http://dinosaur.compilertools.net/