Tehnici de compilare

Mircea Drăgan

November 7, 2019

Cuprins

Capitolul 1

Introducere

Un **limbaj de programare** este un limbaj care are drept scop descrierea unor procese de prelucrare a anumitor date şi a structurii acestora prelucrare care se realizează în general cu ajutorul unui sistem de calcul.

Există în prezent un număr mare de limbaje de programare de *nivel înalt* sau evoluate care se caracterizează printr-o anumită naturaleţe, în sensul că descrierea procesului de prelucrare cu ajutorul limbajului este apropiată de descrierea naturală a procesului respectiv. Dintre limbajele de acest tip cu o anumită răspândire în momentul de faţă menţionăm limbajele PASCAL, FORTRAN, C, JAVA, etc. Un program redactat într-un limbaj de programare poartă denumirea de **program sursă**.

Fiecare sistem de calcul, în funcţie de particularităţile sale, posedă un anumit limbaj propriu, numit *cod maşină*, acesta fiind singurul limbaj înţeles de procesorul sistemului. Un astfel de limbaj conţine un număr mic de instrucţiuni elementare de manipulare a datelor (dar care se execută foarte rapid). Un program redactat în limbajul cod maşină al sistemului de calcul îl numim **program obiect**.

Procesul de transformare al unui program sursă în program obiect se numeşte **compilare** sau *translatare*, uneori chiar *traducere*.

Tehnicile de compilare sunt tehnici de programare specializate utilizate in primul rând pentru la scrierea programelor de translatare, aplicabile la realizarea unei game de programe similare translatoarelor.

Primul compilator a fost realizat in 1957-1958 pentru limbajul FOR-

TRAN, de către o echipă de cercetatori de la I.B.M., condusă de John Backus. Acum, scrierea unui compilator se face mult mai uşor şi simplu datorită elaborării unei teorii a compilării fundamentate matematic de Teoria limbajelor formale şi apariţia unor unelte software specializate pentru generarea automată a unor componente ale translatorului.

1.1 De ce invăţăm tehnici de compilare?

Desigur că foarte puţini informaticieni vor avea nevoie să scrie un compilator pentru un limbaj general de nivel înalt, ca de exemplu C, Pascal, Java. Totuşi, iată câateva motive pentru care acest curs apare în planul de invăţământ al majorităţii instituţiilor de învăţământ superior din domeniul Computer Science:

- Se consideră că tehnicile de compilare impreună cu teoria limbajelor formale face parte din cunoştintele de bază (*cultura generală obligatorie*) ale unui informatician.

- Un bun meseriaş (*profesionist*) trebuie să îşi cunoască sculele, iar pentru un informatician limbajele şi compilatoarele sunt printre principalele instrumente de lucru.

- Tehnicile de compilare folosite pentru verificarea corectitudinii programelor sunt utile şi în alte ramuri ale informaticii (*Sisteme de operare, Baze de date, Prelucrarea textelor, etc.*).

- Este foarte posibil ca un informatician să scrie un minicompilator sau interpretor pentru un limbaj specializat.

Primul motiv pare un pic *exagerat* deoarece este legat de istoricul dezvoltării informaticii ca ştiinţă şi aceasta este un domeniu în continuă schimbare, deci este greu de definit ce înseamnă cultură generală în domeniu.

Al doilea motiv este mult mai întemeiat deoarece înţelegerea modului de construcţie a compilatoarelor va ajuta programatorul în depanarea programelor scrise în limbaje de nivel înalt şi optimizarea programului ce se execută pe maşina de calcul.

Cunoaşterea mecanismelor de verificare a programelor ajută la înţelegerea şi manipularea facilă a oricărui tip de text structurat, ca de exemplu XML. Ultimul motiv devine tot mai actual datorită folosirii limbajelor specializate (DSL–*Domain Specific Languages*) destinate unei clase de probleme, de exemplu: interogarea bazelor de date, formatarea textelor şi a imaginilor, simularea economică, etc.

1.2 Limbaje de programare si traducerea programelor

Aşa cum am precizat în introducere, un **limbaj de programare** este un limbaj care are drept scop descrierea unor procese de prelucrare a anumitor date şi a structurii acestora (în unele cazuri descrierea structurii datelor este preponderentă), prelucrare care se realizează în general cu ajutorul unui sistem de calcul.

Limbajele de programare de *nivel înalt*, gen PASCAL, FORTRAN, C, JAVA, sunt oarecum independente faţă de sistemul de calcul.

O altă clasă importantă de limbaje, sunt *limbajele de asamblare*, sau de nivel inferior, ale căror caracteristici depind de sistemul de calcul considerat. In general, fiecare sistem de calcul (sau tip de sistem de calcul), are propriul său limbaj de asamblare; de exemplu, limbajul de asamblare ale sistemelor de calcul echipate cu procesoare de tip Intel Z-80 este denumit în mod curent ASSEMBLER. Instrucţiile unui limbaj de asamblare corespund cu operaţiile simple ale sistemului de calcul iar stocarea datelor în memorie este realizată direct de utilizator la nivelul locaţiilor elementare ale memoriei. Există de asemenea anumite *pseudo-instrucţii* sau *directive* referitoare la alocarea memoriei, generarea datelor, segmentarea programelor, etc., precum şi macroinstrucţii care permit generarea unor secvenţe tipice de program sau accesul la bibliotecile de subprograme.

In afară de limbajele evoluate şi de limbajele de asamblare, există numeroase *limbaje specializate*, numite uneori şi de *comandă*, care se referă la o anumită clasă de aplicaţii. Menţionăm de exemplu limbajul pentru prelucrarea listelor LISP, limbajele utilizate în cadrul softwarelui matematic (Mathematica, Maple, MATCAD, etc.) şi multe altele. În

general însă astfel de limbaje nu sunt considerate limbaje de programare propriuzise.

Translatarea unui **program sursă** în **program obiect** se numeşte **compilare** în cazul limbajelor evoluate, iar în cazul limbajelor de asamblare se utilizează termenul de *asamblare*.

Compilarea (asamblarea) este efectuată de un program al sistemului numit **compilator** (*asamblor*). De multe ori compilatoarele nu produc direct program obiect, ci un text intermediar apropiat de programul obiect, care în urma unor prelucrări ulterioare devine program obiect. De exemplu, compilatoarele sistemelor de operare DOS produc un text numit obiect (fişiere cu extensia obj) care în urma unui proces numit editare de legături şi a încărcării în memorie devine program obiect propriuzis, numit program executabil (fişiere cu extensia exe). Există mai multe raţiuni pentru o astfel de tratare, între care posibilitatea cuplării mai multor module de program realizate separat sau provenite din limbaje sursă diferite, posibilitatea creării unor biblioteci de programe în formatul intermediar şi utilizarea lor în alte programe, etc.

Un program sursă poate de asemenea să fie executat de către sistemul de calcul direct, fără transformarea lui prealabilă în program obiect. În acest caz, programul sursă este prelucrat de un program al sistemului numit *interpretor*; acesta încarcă succesiv instrucţiile programului sursă, le analizează din punct de vedere sintactic şi semantic şi după caz, le execută sau efectuează anumite operaţii auxiliare.

Procesul de compilare este un proces relativ complex şi comportă operaţii care au un anumit caracter de autonomie. Din aceste motive procesul de compilare este de obicei descompus în mai multe subprocese sau faze, fiecare fază fiind o operaţie coerentă, cu caracteristici bine definite. În principiu aceste faze sunt parcurse secvenţial (pot să existe şi anumite reveniri) iar programul sursă este transformat succesiv în formate intermediare. Se poate considera că fiecare fază primeşte de la faza precedentă un fişier cu programul prelucrat într-un mod corespunzător fazei respective, îl prelucrează şi furnizează un fişier de ieşire, iarăşi într-un format bine precizat, fişier utilizat în faza următoare. Există cinci faze de compilare principale: analiza lexicală, analiza sintactică, generarea formatului intermediar, generarea codului, optimizarea codului şi două faze auxiliare, tratarea erorilor şi tratarea tabelelor (vezi figura 1.1).

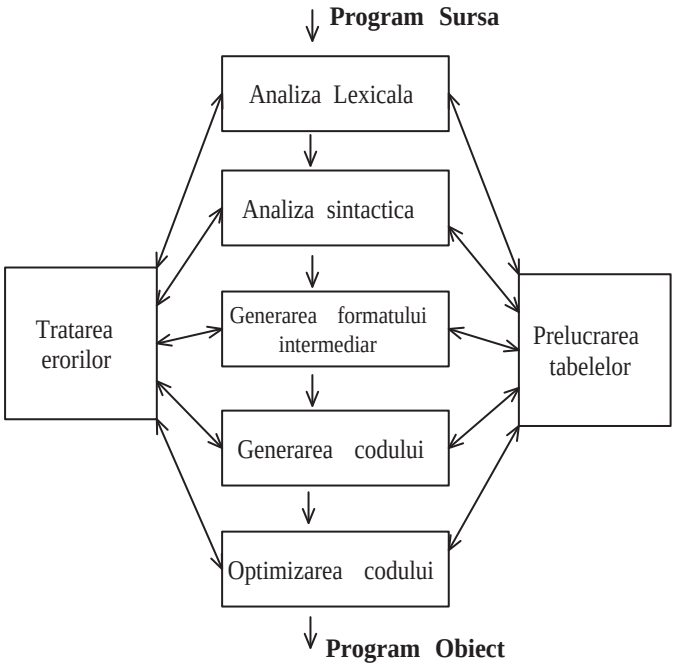

Figura 1.1: Fazele compilării

1.3 Structura unui compilator

Analiza lexicală are ca obiectiv principal determinarea unităţilor lexicale ale unui program (*secvenţe de caractere cu înţeles logic*), furnizarea codurilor acestora şi detectarea eventualelor erori lexicale. De exemplu, în secvenţa de program $ion = a + 3 * c$ sunt găsite următoarele unităţi lexicale, sau **atomi** (*tokens* în limba engleză):

- identificatorii ion, a şi c;
- operatorii =, +, *;
- numărul întreg 3;

Pe lângă aceste operaţii de bază, la analiza lexicală se mai pot efectua anumite operaţii auxiliare precum: eliminarea spaţiilor albe (dacă limbajul permite utilizarea fără restricţii a acestora), ignorarea comentariilor, diferite conversiuni ale unor date (care se pot efectua la această fază), completarea tabelelor compilatorului.

Analiza sintactică grupează ierarhic atomii în colecţii mai mari, numite unităţi sintactice ale programului (secvenţe de text pentru care se poate genera format intermediar: expresiile, instrucţiunile, declaraţiile, etc.) şi verifică programul din punct de vedere sintactic. Este faza centrală a unui compilator, deseori toate celelalte faze sunt rutine apelate de analizorul sintactic pe măsură ce este posibilă efectuara unei părţi din faza respectivă. Deseori se foloseşte o variantă a arborelui de derivare pentru gramatici independente de context, numit **arbore sintactic**, în care fiecare nod reprezintă o operaţie iar fii sunt argumentele operaţiei (fig. 1.2). Tot la analiza sintactică se definitivează tabelele de informaţii (verificarea tipurilor, analiza domeniului de vizibilitate, etc) şi se realizează prelucrarea erorilor sintactice.

Faza de *generare a formatului intermediar* se referă la transformarea programului într-o formă numită intermediară pornind de la care se poate, printr-o procedură relativ simplă, să se obţină programul obiect. Structura acestei faze depinde de formatul intermediar ales de către proiectant şi de modalitatea de implementare; uzual se folosesc *cvadruple, triplete* sau *şirurile poloneze*.

Generarea codului este o fază în care se realizează codul obiect corespunzător programului. Practic în această fază se obţine programul

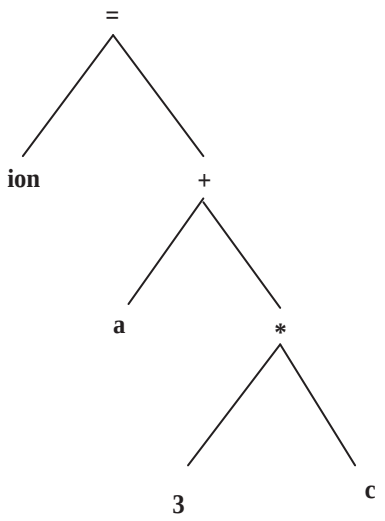

Figura 1.2: Arborele sintactic pentru $ion = a + 3 * c$

în limbaj de asamblare corespunzător programului sursă redactat într-un limbaj evoluat. În mod obişnuit generatorul de cod este constituit din rutinele generatoare de cod corespunzătoare diverselor unităţi ale formatului intermediar.

În faza de *optimizare* a codului se realizează o anumită îmbunătăţire a codului obiect astfel încât programul rezultat să fie cât mai performant (în privinţa consumului de timp şi memorie). Cele mai tipice exemple sunt eliminarea încărcărilor şi a memorărilor redundante sau optimizarea ciclurilor.

Fazele de *Prelucrare a tabelelor* şi de *Tratare a erorilor* vor fi atinse numai parţial în aceast curs. Facem însă menţiunea că prelucrarea tabelelor poate să aibă o influenţă importantă asupra performanţelor unui compilator, în mod special din punctul de vedere al timpului de execuţie (compilare) şi că tratarea erorilor are o anumită implicaţie în eliminarea erorilor sintactice sau semantice.

Delimitarea fazelor compilării ca în structura precedentă urmăreşte organizarea logică a unui compilator. În implementări concrete fazele

se grupează de obicei în **front–end** şi **back–end**. Front end cuprinde de obicei fazele dependente de limbajul sursă şi include analiza lexicală, analiza sintactică, crearea tabelei de simboluri şi generarea formatului intermediar. Back end include acele porţiuni ce depind de calculatorul destinaţie şi include generarea şi optimizarea codului.

Capitolul 2

Analiza lexicală

2.1 Analiza lexicală

Procesul de analiză lexicală este o fază a procesului de compilare în care se determină unităţile lexicale (cuvintele, atomii) ale unui program sursă, se furnizează codurile interne ale acestora şi se detectează eventualele erori lexicale. Analizorul lexical mai poate efectua o serie de operaţii auxiliare precum: eliminarea blank-urilor, ignorarea comentariilor, diferite conversiuni ale unor date, completarea tabelelor compilatorului, gestiunea liniilor textului sursă.
Unităţi lexicale O unitate lexicală (Lexic = vocabular; totalitatea cuvintelor unei limbi) este o secvenţă din textul sursă care are o anumită unitate logică. Definiţia riguroasă a unităţilor lexicale ale unui limbaj particular se dă la definirea limbajului de programare respectiv. De obicei, în majoritatea limbajelor de programare, unităţile lexicale sunt: *cuvinte cheie, identificatori, constante, operatori, delimitatori.* Din punctul de vedere al analizei lexicale şi al modului de prelucrare, unităţile lexicale pot fi de două categorii:

- Unităţi lexicale simple, sunt unităţi lexicale care nu comportă atribute suplimentare, de exemplu, cuvintele cheie, operatorii;
- Unităţi lexicale compuse (atributive), sunt unităţi lexicale care comportă anumite atribute suplimentare, de exemplu, identificatorii şi constantele. Atributele sunt informaţii specifice, de exemplu, tipul identificatorului sau al constantei (întreg, real, etc.).

> Este necesară specificarea tipului unui identificator sau al unei constante din startul programului deoarece structura programului obiect sau reprezentarea internă a constantelor depinde de acest tip.

Reprezentarea internă a unităţilor lexicale se face în funcţie de categoria lor. Cele simple se reprezintă printr-un cod specific(număr întreg). Unităţile lexicale compuse se reprezintă prin cod şi informaţii (de natură semantică) asupra sa. De obicei compilatoarele utilizează tabele pentru stocarea atributelor (tabel de constante, tabel de variabile, tabel de etichete, etc.). In acest caz unitatea lexicală se reprezintă intern printr-un cod urmat de o referinţă într-un tabel. Informaţia conţinută de tabel ar putea fi pentru constante: cod, tip, valoare, iar pentru identificatori: cod, tip, indicator de iniţializare.

Este posibil ca o clasă de unităţi lexicale simple să se reprezinte printr-un cod unic şi un atribut pentru distingerea în cadrul clasei. De exemplu, operatorii aritmetici cu aceiaşi prioritate au o tratare similară din punct de vedere al analizei sintactice. Lista unităţilor lexicale şi definiţia riguroasă a acestora se dă la proiectarea compilatorului.

Un exemplu de unităţi lexicale şi coduri asociate ar putea fi cele din figura 2.1.

Analizorul primeşte textul sursă şi produce şirul de unităţi lexicale în codificarea internă. De exemplu secvenţa de text sursă următoare:

```
{if (unu < 2) return 0;
  a=33;
  }
```

va produce sirul de unităţi lexicale
LBRACE If LPAR [ID,22] [opr,1] [NUM, 40], RPAR RETURN [NUM, 42] SEMI [ID,24] opAssign [NUM,44] SEMI RBRACE.

In lista codurilor interne găsite atributul identificatorului este adresa relativa din tabela de identificatori, analog atributele constantelor numerice sunt adrese relative în tabelele de constante.

Majoritatea limbajelor evoluate conţin şi secvenţe de text care nu sunt unităţi lexicale, dar au acţiuni specifice asociate . De exemplu:

Unitate lexicală	COD	ATRIBUT	Exemplu
if	if = 1	-	if, If, IF
else	else = 2	-	else ElSe
identificator	ID = 3	referinţă	Nelu v tabel
constantă întreagă	NUM = 4	referinţă	4 -3 233
constantă reală	FLOAT = 5	referinţă	4.32 -3.233
+	op = 6	1	
-	op = 6	2	
×	op = 6	3	
/	op = 6	4	
<	opr = 7	1	
>	opr = 7	2	
<=	opr = 7	3	
>=	opr = 7	4	
(	LPAR = 8	-	
)	RPAR = 9	-	
{	LBRACE = 10	-	
}	RBRACE = 11	-	

Figura 2.1: Coduri asociate unităților lexicale

```
comentarii                    /* text */

directive de preprocesare     #include<stdio.h>
                              #define MAX 5.6
```

Înaintea analizei lexicale (sau ca subrutină) se preprocesează textul şi abia apoi se introduce rezultatul în analizorul lexical.

Specificarea unităţilor lexicale Definirea riguroasă a unităţilor lexicale se face de către proiectantul limbajului. O posibilitate de descriere este limbajul natural. De exemplu, pentru **C** şi **JAVA**:

"*Un identificator este o secvenţa de litere şi cifre: primul caracter trebuie să fie literă. Liniuţa de subliniere contează ca literă. Literele mari şi mici sunt diferite. Dacă şirul de intrare a fost împărţit în unităţi lexicale până la un caracter dat, noua unitate lexicală se consideră astfel încât să includă cel mai lung şir de caractere ce poate constitui o unitate lexicală. Spaţiile, taburile, newline şi comentariile sunt ignorate cu excepţia cazului când servesc la separarea unităţilor lexicale. Sunt necesare spaţii albe pentru separarea identificatorilor, cuvintelor cheie şi a constantelor.*"

Orice limbaj rezonabil poate fi folosit pentru implementarea unui analizor lexical.

Unităţile lexicale se pot specifica cu ajutorul limbajelor regulate, deci folosind gramatici de tipul 3 sau expresii regulate ce notează limbajele. Ambele specificaţii conduc la construirea de automate finite echivalente, care se pot uşor programa. În cele ce urmează, vom folosi ambele variante de specificare pentru un set uzual de unităţi lexicale întâlnit la majoritatea limbajelor evoluate.

Considerăm gramatica regulată ce generează identificatori, constante întregi, cuvinte cheie, operatori relaţionali şi aritmetici.

$$G: \begin{cases} < ul > \rightarrow < id > | < num > | < cc > | < op > | < opr > \\ < id > \rightarrow l < id1 > | l, < id1 > \rightarrow l < id1 > | c < id1 > | l | c \\ < num > \rightarrow c < num > | c \\ < cc > \rightarrow if | do | else | for \\ < op > \rightarrow + | - | * | / \\ < opr > \rightarrow < | <= | > | >= \end{cases},$$

unde *l*-litera, *c*-cifra.

Pornind de la această gramatică se poate construi una echivalentă în formă normală, apoi se extrage funcţia de evoluţie a automatului finit determinist echivalent ce recunoaşte unităţile lexicale.

Descrieri echivalente ale unităţilor lexicale cu ajutorul expresiilor regulate sunt

```
cuvinte cheie = if | do | else | for

identificatori = ( a|b|c|...z )( a|b|c|...z|0|1|...|9 )*

Numar = ( 0|1|...|9 )( 0|1|...|9 )*

Operatori aritmetici = + | - | * | /

Operatori relationali = < | <= | > | >=
```

Limbajul generat de gramatica precedentă se obţine prin suma expresiilor regulate. Menţionăm că există programe specializate (Lex, Flex, JavaCC) ce generează un analizor lexical (în C sau Java) pornind de la expresiile regulate. Sintaxa folosită în scrierea expresiilor regulate este dependentă de programul folosit.

Programarea unui analizor lexical Realizarea efectivă a unui analizor revine la simularea funcţionării unui automat finit. O variantă de programare este ataşarea unei secvenţe de program la fiecare stare a automatului. Dacă starea nu este stare finală atunci secvenţa citeşte următorul caracter din textul sursă şi găseşte următorul arc din diagrama de stări. Depinzând de rezultatul căutării se transferă controlul altei stări sau se returnează eşec (posibilă eroare lexicală). Dacă starea este finală atunci se apelează secvenţa de returnare a codului unităţii lexicale şi eventuala instalare a unităţii lexicale în tabelele compilatorului.

Pentru simplificarea implementării se caută următoarea unitate lexicală prin încercarea succesivă a diagramelor corespunzătoare fiecărei unităţi lexicale (într-o ordine prestabilită). Eroarea lexicală se semnalează doar atunci când toate încercările se încheie cu eşec.

De obicei textul sursă conţine şi secvenţe ce se pot descrie cu ajutorul expresiilor regulate, dar care nu sunt unităţi lexicale (de exemplu

comentariile). Aceste secvenţe nu vor genera cod lexical, dar au asociate diverse acţiuni specifice. Pentru a evita apariţia unor caractere necunoscute in textul sursă se consideră şi limbajul ce constă din toate simbolurile ASCII. Astfel, indiferent de unde începe analiza textului, programul de analiză lexicală găseşte o potrivire cu o descriere. Spunem că specificaţia este **completă**.

Există posibilitatea ca mai multe secvenţe cu aceeaşi origine să corespundă la diferite descrieri ale unităţilor lexicale. Se consideră unitate lexicală cel mai lung şir ce se potriveşte unei descrieri (**longest match rule**). Dacă sunt două reguli care se potrivesc la acelaşi şir de lungime maximă atunci se consideră o prioritate asupra descrierilor (**rule priority**).

De exemplu, in textul următor,

```
i  if  if8
```

unităţile lexicale delimitate vor fi **i**–identificator, **if**–cuvânt cheie, **if8**–identificator. Regula de prioritate se aplică pentru potrivirea lui if cu identificator şi cuvânt cheie, iar criteriul de lungime maximă pentru if8.

Pentru depistarea celei mai lungi potriviri, din punt de vedere al programării analizorului, este suficient să prevedem un pointer suplimentar pe caracterul ce corespunde ultimei stări finale atinse pe parcursul citirii.

Studiu de caz. Se consideră problema realizării unui analizor lexical ce delimitează într-un text sursă cuvinte din limbajul ce conţine identificatori, cuvinte cheie (pentru simplificare folosim doar cuvântul cheie if), constante numerice (întregi fără semn). De asemenea se face salt peste spaţiile albe şi se ignoră comentariile. Presupunem că un comentariu începe cu două caractere slash şi se termină cu newline. Orice caracter ce nu se potriveşte descrierii este semnalat ca şi caracter ilegal in text.

Etapa I. Descriem secvenţele cu ajutorul expresiilor regulate

```
IF = "if"

ID = (a|b|c|...|z)(a|b|c|...|z|0|1|...|9)*
```

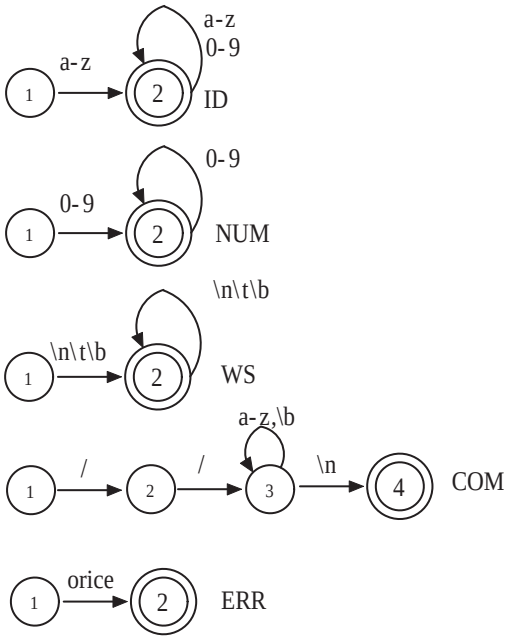

Figura 2.2: Automatele finite corespunzătoare expresiilor regulate

```
NUM = (0|1|...|9)(0|1|...|9)* = (0|1|...|9)+

WS = (\n|\t|" ")+

COMMENT = "//"(a|b|c|...|z|0|1|...|9|" ")*\n

ALL = a|b|...|z|0|...|9|\t| ...  toate caracterele ASCII
```

Etapa II. Corespunzător expresiilor avem următoarele automate finite deterministe echivalente, prezentate în figura 2.2 (stările au fost notate prin numere întregi):

Etapa III. Se construieşte sistemul tranziţional (vezi figura 2.3) ce recunoaşte limbajul reuniune, adăugând o nouă stare iniţială (notată cu 1), pe care o conectăm prin arce punctate (ce corespund λ-tranziţiilor). Construcţia provine din *legarea* sistemelor tranziţionale *în paralel*. S-

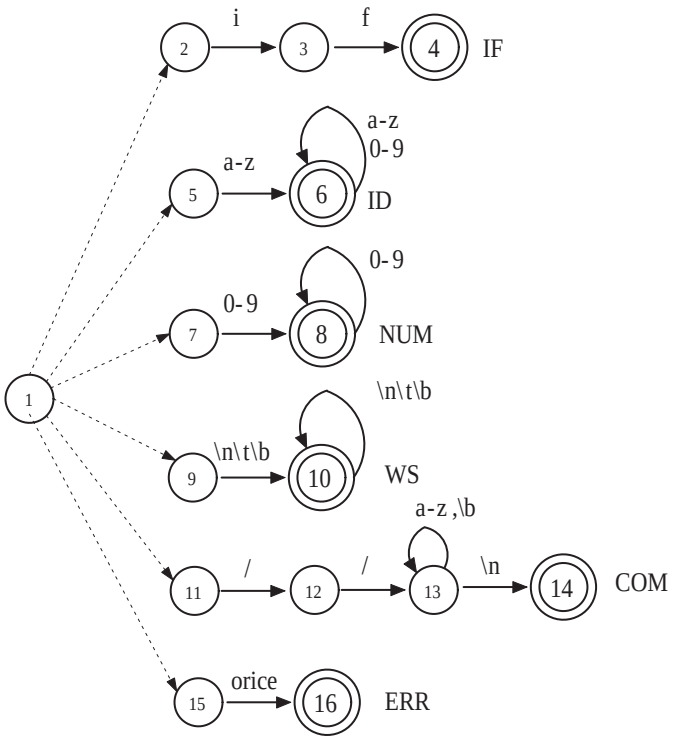

Figura 2.3: Sistemul tranziţional ce recunoaşte reuniunea limbajelor

au renumerotat stările sistemului tranziţional, asignând nume simbolice stărilor finale.

Etapa III. Construcţia automatului finit determinist general ce recunoaşte reuniunea limbajelor. Pentru aceasta sistemul tranziţional se transformă cu teorema de echivalenţă în automat finit determinist (practic se trece de la stări ale sistemului tranziţional la submulţimi de stări, ce devin stările automatului finit).

În cazul particular al automatului nostru, diagrama de stări este dată în figura 2.4.

Etapa IV. *Programarea analizorului lexical.*

Automatul finit obţinut are stările finale asociate cu clasele de cuvinte recunoscute. Se asociază acţiuni stărilor finale, corespunzător

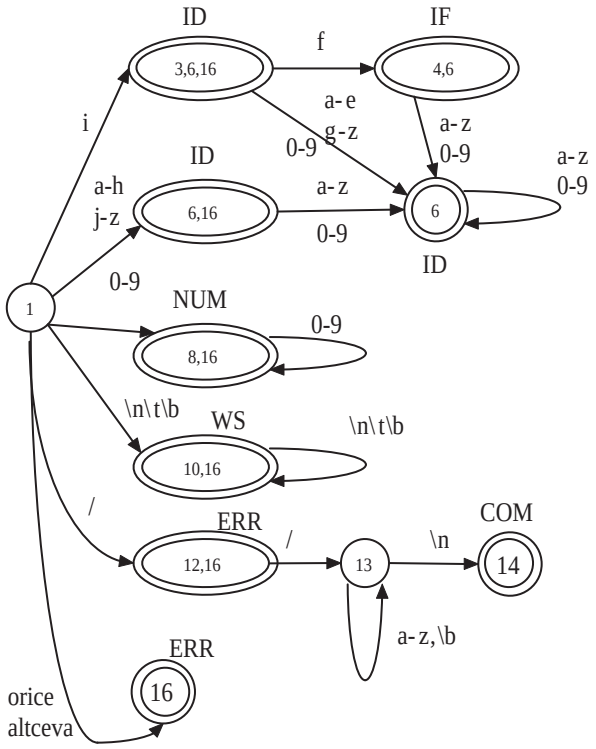

Figura 2.4: Automatul finit determinist ce recunoaşte reuniunea limbajelor

definiţiilor unităţilor lexicale (de exemplu pentru constante numerice se generează reprezentarea interna, se memorează în tabelul de constante şi se returnează codul unităţii lexicale NUM). Pentru programarea analizorului se folosesc trei variabile de tip pointer in textul sursă: **F**irst**S**ymbol, **C**urrent**S**ymbol, **L**ast**F**inal**S**ymbol, ce reţin indicele caracterului de început al secvenţei, indicele caracterului ce urmează la citire, indicele ultimului caracter ce corespunde atingerii unei stări finale. Cei trei pointeri au fost reprezentaţi prin semnele grafice $|$, $\perp$ respectiv $\top$. De asemenea considerăm o variabilă *State*, ce reţine starea curentă a automatului.

In tabelul 2.5 este indicată evoluţia analizorului lexical (inclusiv acţiunile asociate) pentru cazul analizei următorului text sursă.

```
if if8%// ha\n
```

Pentru simplificarea codificării, stările automatului finit determinist au fost redenumite prin numere întregi începând cu starea iniţială 1, starea $\{3, 6, 16\} = 2$, $\{4, 6\} = 3$, $\{6, 16\} = 4$, ş.a.m.d. de la stânga la dreapta şi de sus în jos. De obicei funcţia de evoluţie asociată automatului finit determinist se memorează sub forma unui tablou bidimensional de întregi, ca în figura 2.6. Starea 0 este asociată cu blocarea automatului. Ajungerea în această stare echivalează cu găsirea ultimei unităţi lexicale, între pointerii $|$ şi $\top$. Se execută acţiunea asociată stării finale şi se reia căutarea (*resume*) următoarei unităţi lexicale începând cu caracterul imediat următor pointerului $\top$.

Observaţie: Cele mai costisitoare operaţiuni (ca timp) din analiza lexicală sunt ignorarea comentariilor şi tratarea erorilor lexicale. Primele generatoare automate de analizoare lexicale şi sintactice au apărut în anii ′70 şi au fost incluse în sistemul de operare Unix.

2.2 Minimizarea automatelor finite deterministe

In procesul de generare automată a unui analizor lexical apare uneori o fază de optimizare a automatului finit determinist în sensul determinării unui automat finit determinist echivalent care să aibă un număr minim

Last Final	Current State	Current Input	Accept Action
0	1	$\|^{\top}_{\perp} i\ f\quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
2	2	$\|\ i^{\top}_{\perp} f\quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
3	3	$\|\ i\ f^{\top}_{\perp}\quad i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	return $cc =< if >$
	0	$\|\ i\ f^{\top}\ {}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\|^{\top}_{\perp}\ i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
7	7	$i\ f\|\ {}^{\top}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
	0	$i\ f\|\ {}^{\top} i_{\perp} f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\ \|^{\top}_{\perp} i\ f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
2	2	$i\ f\ \|\ i^{\top}_{\perp} f\ 8\ \%\ /\ /\quad h\ a\ \backslash n$	
3	3	$i\ f\ \|\ i\ f^{\top}_{\perp} 8\ \%\ /\ /\quad h\ a\ \backslash n$	
5	5	$i\ f\ \|\ i\ f\ 8^{\top}_{\perp}\ \%\ /\ /\quad h\ a\ \backslash n$	return $id =< if8 >$
	0	$i\ f\ \|\ i\ f\ 8^{\top}\ \%_{\perp}/\ /\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\quad i\ f\ 8\|^{\top}_{\perp}\ \%\ /\ /\quad h\ a\ \backslash n$	
11	11	$i\ f\quad i\ f\ 8\|\ \%^{\top}_{\perp}/\ /\quad h\ a\ \backslash n$	print ("illegal character: %");
	0	$i\ f\quad i\ f\ 8\|\ \%^{\top}/_{\perp}/\quad h\ a\ \backslash n$	*resume*
0	1	$i\ f\quad i\ f\ 8\ \%\|^{\top}_{\perp}/\ /\quad h\ a\ \backslash n$	
0	8	$i\ f\quad i\ f\ 8\ \%\|^{\top}/_{\perp}\ /\quad h\ a\ \backslash n$	
0	9	$i\ f\quad i\ f\ 8\ \%\|^{\top}/\ /_{\perp}\quad h\ a\ \backslash n$	
...	...	...	

Figura 2.5: Evoluţia analizorului lexical pentru textul *if if8*%// ha\n

```
int edges[][] = {  /* ...  0 1 2 ...  e f g h i j ...   */

/* state 0 */    {0,0, ...,0,0,0, ...,0,0,0,0,0,0, ... },

/* state 1 */    {0,0, ...,6,6,6, ...,4,4,4,4,2,4, ... },

/* state 2 */    {0,0, ...,5,5,5, ...,5,3,5,5,5,5, ... },

etc

}
```

Figura 2.6: Reprezentarea funcţiei de evoluţie a automatului finit

de stări, deci o reprezentare internă de dimensiune mai mică. Procedeul folosit pentru optimizare se bazează pe teorema de caracterizare algebrică a limbajelor regulate ce oferă o variantă de automat finit minimal.

Teorema 2.1 **Caracterizarea limbajelor regulate**

Fie $E \subset I^*$ un limbaj. Următoarele afirmaţii sunt echivalente.

(a) $E \in \mathcal{R}$;

(b) E este o reuniune de clase de echivalenţe a unei congruenţe de rang finit;

(c) Următoarea congruenţă

$$\mu = \{(p, q) | \chi_E(r_1 p r_2) = \chi_E(r_1 q r_2), \forall r_1, r_2 \in I^*\},$$

unde χ_E este funcţia caracteristică a lui E, este de rang finit.

Pentru demonstraţia completă se poate revedea cursul de Limbaje formale şi teoria automatelor din anul I. Folosind clasele de echivalenţă determinate de congruenţa μ se poate construi un automat finit ce recunoaşte limbajul E astfel":

$AF_\mu = (I^*/\mu, I, f, C_\lambda, \Sigma_f)$,
unde funcţia de evoluţie f şi mulţimea de stări finale sunt

$$f(C_p, i) = C_{pi},\ \Sigma_f = \{C_p | p \in E\}.$$

Automatul astfel definit are proprietatea de minimalitate din punctul de vedere al numărului de stări. Deoarece construcţia automatului presupune identificarea claselor de echivalenţă a unei relaţii definite pe o mulţime infinită, procedeul nu oferă un algoritm acceptabil de minimizare a automatului finit. Din acest motiv, minimizarea se face prin gruparea stărilor unui automat finit dat folosind o relaţie de echivalenţă pe multimea finită a stărilor şi definirea unui automat finit echivalent ce are proprietatea că este $izomorf$ cu AF_μ.

Întreaga construcţie se bazează pe proprietatea de *separabilitate* a două stări, care provine de la următoarea observaţie. Pentru oricare două cuvinte dintr–o clasă de echivalenţă definită de μ, $p, q \in \mu$ avem $\chi_E(pr) = \chi_E(qr), \forall r \in I^*$, adică $f(s_0, pr) \in \Sigma_f$ dacă şi numai dacă $f(s_0, qr) \in \Sigma_f$ pentru orice cuvânt r. Deci, stările $s = f(s_0, p)$, $s' = f(s_0, q)$ au proprietatea că $f(s, r) \in \Sigma_f$ dacă şi numai dacă $f(s', r) \in \Sigma_f$ pentru orice cuvânt arbitrar r. Se spune că stările s şi s' sunt *inseparabile* (prin cuvinte de orice lungime).Cum orice automat defineşte o partiţionare a lui I^* în clase de echivalenţă incluse în totalitate în clasele determinate de μ, adică o clasă a lui μ este formată prin reuniunea unui număr finit de clase determinate de automat, se incearcă o partiţionare a multimii stărilor în submulţimi pentru care are loc proprietatea inseparabilitate caracteristică tuturor claselor din reuniune.

Vom da o descriere a procedeului după modelul prezentat de T. Jucan în [4]. Fie $E \subset I^*$ un limbaj regulat peste alfabetul I şi $AFD = (\Sigma, I, f, s_0, \Sigma_f)$ un automat finit determinist ce recunoaşte E.

Definiţie 2.1 *Spunem că un cuvânt $p \in I^*$ separă stările $s_1, s_2 \in \Sigma$ în raport cu Σ_f) dacă una şi numai una din stările $f(s_1, p)$ şi $f(s_2, p)$ se află în Σ_f), adică $\chi_{\Sigma_f}(f(s_1, p)) \neq \chi_{\Sigma_f}(f(s_2, p))$.*

Definiţie 2.2 *Stările s_1 şi s_2 se zic* k–inseparabile *în raport cu Σ_f, notat $s_1 \overset{k}{\equiv} s_2$, dacă pentru orice cuvânt $p \in I^*$ cu lungimea $|p| \leq k$ avem $\chi_{\Sigma_f}(f(s_1, p)) = \chi_{\Sigma_f}(f(s_2, p))$.*

Definiţie 2.3 *Stările s_1 şi s_2 se zic* inseparabile *în raport cu Σ_f, , notat simplu $s_1 \equiv s_2$ dacă sunt k–inseparabile în raport cu Σ_f pentru orice număr natural k.*

Este evident că relaţiile , $\overset{k}{\equiv}$ şi $\equiv$ sunt relaţii de echivalenţă pe mulţimea stărilor. Pentru determinarea relaţiilor între stări sunt utile următoarele două leme, uşor de demonstrat.

Lema 2.1 *Pentru orice* $k \geq 1$, $s_1, s_2 \in \Sigma$ *avem* $s_1 \overset{k}{\equiv} s_2$ *dacă şi numai dacă* $s_1 \overset{k-1}{\equiv} s_2$ *şi* $f(s_1, i) \overset{k-1}{\equiv} f(s_2, i), \forall i \in I$.

Lema 2.2 *Dacă* $|\Sigma| = n$ *atunci stările* s_1 *şi* s_2 *sunt inseparabile dacă şi numai dacă sunt* $(n-2)$*–inseparabile.*

Pentru calculul relaţiilor se porneşte cu $\overset{0}{\equiv}$, inseparabilitate prin cuvinte de lungime zero, care împarte mulţimea stărilor în două clase de echivalenţă Σ_f şi $\Sigma - \Sigma_f$, apoi se aplică de cel mult $n-2$ ori lema 2.1.

Următorul rezultat pe care îl enunţăm fără demonstraţie, dă un mod de construcţie al automatului finit minimal.

Teorema 2.2 *Fie* E *un limbaj regulat acceptat de un automat finit determinist* $AFD = (\Sigma, I, f, s_0, \Sigma_f)$ *neblocabil (adică* $f(s,i) \neq \emptyset, \forall s \in \Sigma, i \in I$*) şi fără stări inaccesibile (adică fiecare stare este extremitatea unei traiectorii ce porneşte din starea iniţială). Automatul* $AFD' = (\Sigma/\equiv, I, f', C_{s_0}, \Sigma'_f)$*, unde* $f'(C_s, i) = C_{f(s,i)}, \forall i \in I$ *şi* $\Sigma'_f = \{C_s | s \in \Sigma_f\}$ *recunoaşte limbajul* E *şi este izomorf cu automatul minimal* AF_μ *definit de relaţia* μ *din teorema de structură algebrică a limbajelor regulate.*

2.2.1 Studiu de caz

Considerăm un automatul finit determinist descris prin diagrama de stari din figura 2.7. Să observam că automatul este **complet** (din fiecare nod avem exact un arc pentru fiecare sinmbol de intrare) dar conţine stări inaccesibile. Starea D nu poate să apară pe nici o traictorie ce incepe cu starea iniţială A, deci starea D se elimină din graf împreună cu arcele ce pornesc din ea.

La construcţia automatului finit minimal echivalent se folosesc lemele 2.1 şi 2.2 pentru identificarea claselor de echivalenţa determinate de relaţia $\equiv$ pe mulţimea stărilor. O posibilitate de organizare a calculelor

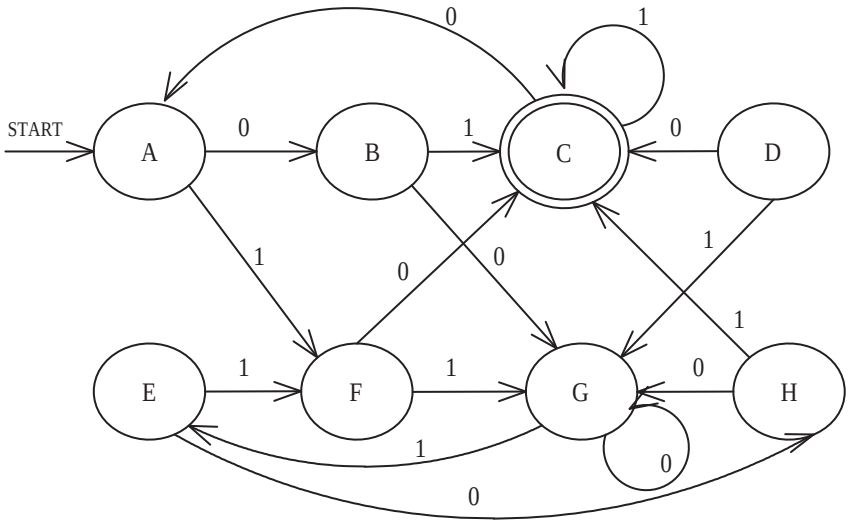

Figura 2.7: Automatul finit determinist

este cea prezentată în figura 2.8 unde se încearca marcarea perechilor de stări separabile prin cuvinte de lungime $0, 1, 2, \ldots, n-2$.

Iniţial se marcheaza în tabel perechile de stări s_i, s_j separabile prin cuvinte de lungime 0, adică perechile de stări ce conţin o stare finală şi una nefinală, exact ca in figura 2.8.

Pentru calculul perechilor de stări separabile prin cuvinte de lungime 1 se aplică lema 2.1, adică se parcurge tabelul şi pentru fiecare pereche de stări nemarcată se marchează perechea dacă există un simbol de intrare a astfel înc at perechea $f(s_i, a), f(s_j, a)$ este deja marcată. În exemplul nostru vom marca perechea A, H deoarece $f(A, 1) = F$ şi $f(H, 1) = C$ iar în tabel perechea de stări F, C este deja marcată, apoi marcăm perechea A, F ş.a.m.d. Prin parcurgerea tabelului de la stânga la dreapta şi de sus în jos vom obţine tabelul descris în figura 2.9

Se repetă procedeul descris la pasul precedent (adica marcarea perechilor nemarcate încă dacă există simboluri de intrare ce ne conduc prin aplicarea funcţiei de evoluţie la o pereche deja marcată) până când la o parcurgere nu se mai pot adăuga marcaje. Numărul de iteraţii nu

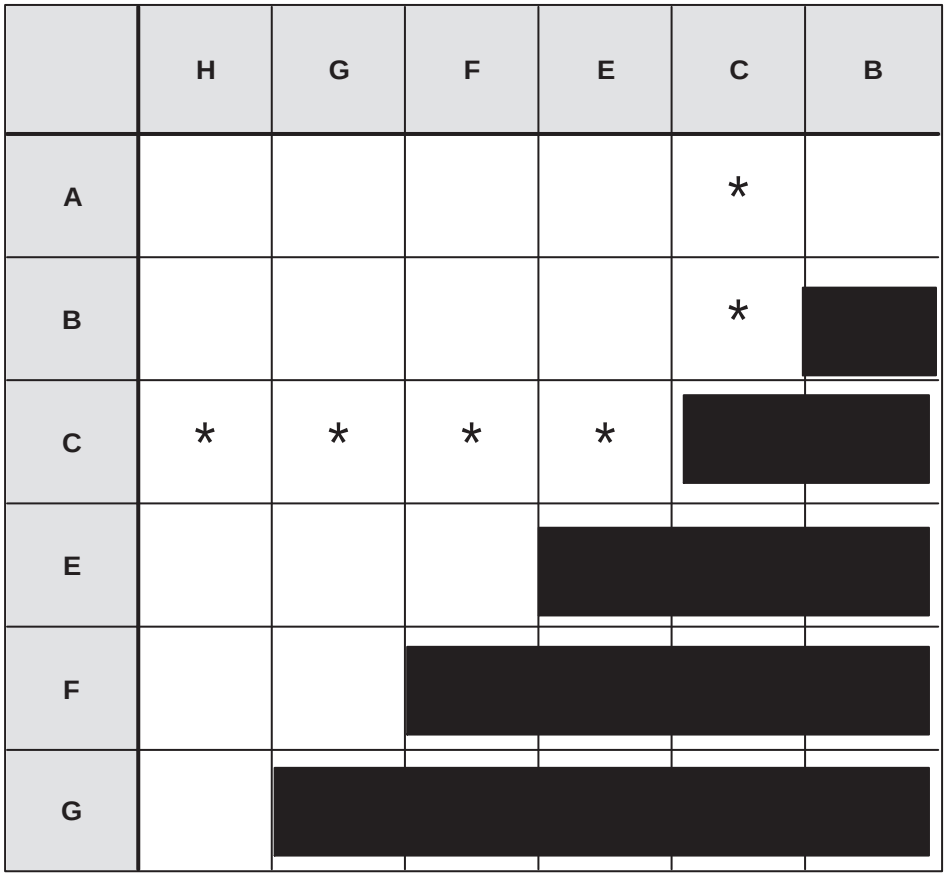

Figura 2.8: Tabelul iniţial

	H	G	F	E	C	B
A	*		*		*	*
B		*	*	*	*	
C	*	*	*	*		
E	*		*			
F	*	*				
G	*					

Figura 2.9: Tabel ce conţine marcajele după prima parcurgere

depăşeşte $n-2$ conform lemei 2.2.

Tabelul final este prezentat în figura 2.10 unde se observă că avem doar două perechi de stări echivalente A, E şi B, H. Diagrama de stări pentru automatul finit minimal echivalent se obţine imediat pornind de la diagrama 2.7 alegând pentru construcţia arcelor orice reprezentant al clasei de echivalenţa. Rezultatul este descris de figura 2.11.

2.3 Construcţia expresiei regulate pentru un limbaj regulat

Expresiile regulate sunt notaţii pentru limbaje regulate (se poate construi un *sistem tranziţional* ce recunoaşte limbajul notat de expresie folosind legarea sistemelor tranziţionale în serie, paralel sau scurtcircuit *vezi [2]*). Problema care rămâne de rezolvat este cum se poate construi o expresie regulată pentru un limbaj recunoscut de un automat finit.

Vom considera un automat finit determinist $AFD = (S, I, f, s_0, S_f)$ cu n stări $S = \{s_1, s_2, \ldots, s_n\}$ unde starea iniţială este $s_0 = s_1$. Putem presupune că automatul este complet şi fără stări inaccesibile, ca în algoritmul de minimizare a unui AFD.

Ideea construcţiei provine din asocierea unui cuvânt recunoscut cu o traiectorie ce porneşte cu starea iniţială şi se termină cu o stare finală din S_f. Se consideră mulţimile de cuvinte ce definesc traiectorii de la starea s_i la starea s_j, iar stările intermediare au indici cel mult k. Formal,

$$\mathbf{R}_{i,j}^{k} = \{p \in I^* | f(s_i, p) = s_j\}, i, j \in \{1, \ldots, n\}, k \geq 0$$

şi pe traiectorie toate stările intermediare au indici cel mult k .

Pentru $k = 0$ se consideră mulţimile de cuvinte ce definesc traiectorii fără stări intermediare, adică

$$\mathbf{R}_{i,j}^{0} = \begin{cases} \{a \in I | f(s_i, a) = s_j\} & , i \neq j \\ \{a \in I | f(s_i, a) = s_j\} \cup \{\lambda\} & , i = j \end{cases}$$

Este evident că pentru mulţimile $\mathbf{R}_{i,j}^{0}$ avem notaţii cu expresii regulate $\mathbf{r}_{i,j}^{0}$ deoarece sunt mulţimi finite. De exemplu pentru mulţimea $\{a, b, c\}$

	H	G	F	E	C	B
A	*	*	*		*	*
B		*	*	*	*	
C	*	*	*	*		
E	*	*	*			
F	*	*				
G	*					

Figura 2.10: Tabel cu stările separabile marcate

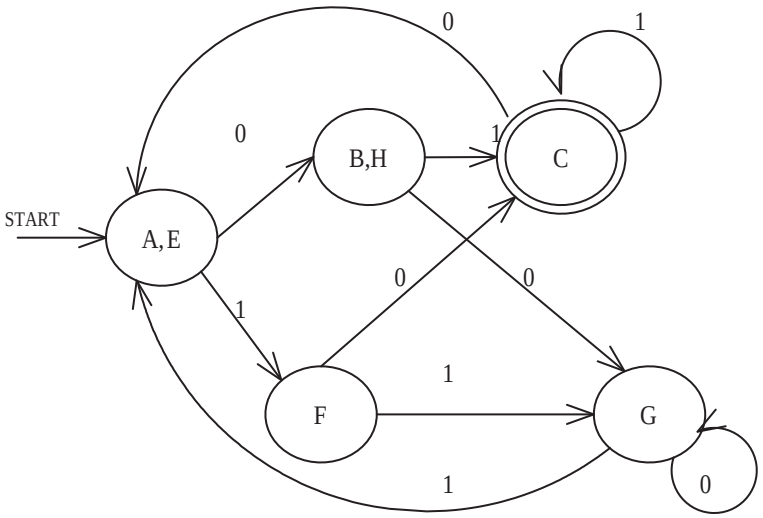

Figura 2.11: Automatul finit determinist minimal

notaţia este $a|b|c$. Pentru calculul mulţimilor $\mathbf{R}_{i,j}^{k}, k > 0$ se foloseşte formula

$$\mathbf{R}_{i,j}^{k} = \mathbf{R}_{i,j}^{k-1} \cup \mathbf{R}_{i,k}^{k-1}(\mathbf{R}_{k,k}^{k-1})^{*}\mathbf{R}_{k,j}^{k-1}$$

Justificarea formulei este următoarea: Pentru un cuvânt din $\mathbf{R}_{i,j}^{k}$ traiectoria porneşte de la starea s_i şi apoi parcurge stări intermediare cu indici maxim k până ajunge la starea s_j. Dacă toţi indicii stărilor intermediare sunt mai mici decât k atunci cuvântul face parte din $\mathbf{R}_{i,j}^{k-1}$. Dacă există pe traiectorie şi indicele k atunci putem partiţiona cuvântul în subcuvinte folosind stările s_k într-un şir de subcuvinte astfel: de la s_i la prima apariţie a stării s_k (adică un cuvânt din $\mathbf{R}_{i,k}^{k-1}$), apoi un subcuvânt cuprins între prima şi a doua apariţie pe traiectorie a stării k (adică un cuvânt din $\mathbf{R}_{k,k}^{k-1}$), apoi un subcuvânt cuprins între a doua şi a treia apariţie pe traiectorie a stării k (adică un cuvânt din $\mathbf{R}_{k,k}^{k-1}$), ş. a. m. d. , iar după ultima apariţie a stării s_k avem un cuvânt ce ne duce în starea s_j (adică un cuvânt din $\mathbf{R}_{k,j}^{k-1}$).

Formula de calcul poate fi transformată într-o formulă cu expresii regulate, adică

$$\mathbf{r}_{i,j}^{k} = \mathbf{r}_{i,j}^{k-1} | \mathbf{r}_{i,k}^{k-1} (\mathbf{r}_{k,k}^{k-1})^{*} \mathbf{r}_{k,j}^{k-1}$$

Este evident că limbajul recunoscut de automatul finit este

$$L(AFD) = \bigcup_{s_i \in S_f} \mathbf{R}_{1,i}^{n}$$

care poate fi notată de expresia regulată $|_{s_i \in S_f} \mathbf{r}_{1,i}^{n}$.

Capitolul 3

Analiza sintactică

Analiza sintactică este o fază a procesului de compilare care are următoarele două obiective principale:

- Stabileşte dacă un cuvânt dat aparţine sau nu limbajului, deci dacă cuvântul este *corect* din punct de vedere sintactic. În particular limbajul poate fi definit de o gramatică generativă de tip Chomsky şi deci termenul de analiză sintactică trebuie înţeles în sensul teoriei limbajelor formale. Mai menţionăm că prin *cuvânt* înţelegem orice structură constituită cu simbolurile acceptate de limbaj, în particular un întreg program, dar de obicei ne vom mărgini la anumite entităţi, de exemplu, o linie sau un rând.

- Determină derivarea (arborele de derivare) corespunzător cuvântului. Odată cu această operaţie sunt degajate anumite structuri pentru care se poate genera cod intermediar, structuri pe care le vom numi *unităţi sintactice*.

Pe lângă aceste obiective principale se mai efectuază şi alte operaţii, de exemplu, analiza şi tratarea erorilor, prelucrarea tabelelor, etc. Rezultatul analizei sintactice va fi un fişier care conţine derivările (arborii de derivare) corespunzătoare unităţilor sintactice în care este descompus programul: expresii aritmetice, declaraţii, etc. Acest fişier este utilizat în faza de generare a formatului intermediar. În mod curent însă, generatoarele de format intermediar sunt nişte rutine, apelate de analizorul sintactic, astfel încât formatul intermediar se obţine succesiv.

Teoretic, problema analizei sintactice este rezolvată de automatele corespunzătoare diverselor tipuri de limbaje; această cale conduce însă la algoritmi cu complexitate mare (număr mare de stări, funcţie de tranziţie conmplexă, etc.). Există algoritmi speciali de analiză sintactică cu eficienţă superioară. În continuare ne vom ocupa cu două clase de astfel de algoritmi:

- Algoritmi top-down (de sus în jos);
- Algoritmi bottom-up (de jos în sus).

3.1 Algoritmi TOP-DOWN

3.1.1 Algoritmul general de analiză top-down

Algoritmul general de analiză top-down are un principiu foarte simplu: *se aplică în mod sistematic regulile de generare, începând cu simbolul de start; în cazul unui eşec se revine în sus şi se încearcă o altă regulă. Regulile se aplică în ordinea în care sunt scrise în gramatică, fără să existe o anumită ordine preferenţială de scriere, întrucât natura algoritmului nu permite nici un fel de ierarhizare a regulilor din punctul de vedere al frecvenţei de utilizare.*

Pentru descrierea riguroasă a acestui algoritm am urmat modelul din cursul *Limbaje Formale şi Tehnici de Compilare* predat la UVT în anii 80 de către Dl. Prof. Ştefan Măruşter, unul dintre pionierii informaticii în Romania.

Fie $G = (V_N, V_T, x_0, \mathcal{P})$ o gramatică de tipul 2 şi $p \in V_T^*$. Ne interesează următoarele două probleme:

(a) $p \in L(G)$?,

(b) Dacă $p \in L(G)$ atunci să se determine derivarea $x_0 \Rightarrow p$.

Considerăm toate regulile care au X_0 în stânga:

$$x_0 \to x_{11} \dots x_{1n_1} | x_{21} \dots x_{2n_2} | \dots | x_{k1} \dots x_{kn_k}$$

Iniţial x_0 devine *activ* şi alege prima regulă $x_0 \to x_{11} \dots x_{1n_1}$. Dacă această alegere este corectă trebuie să avem $x_0 \Rightarrow x_{11} \dots x_{1n_1} \overset{*}{\Rightarrow} p$ şi în conformitate cu *lema de localizare* a gramaticilor de tipul 2, cuvântul

p se poate descompune în forma $p = p_1 p_2 \dots p_{n_1}$, unde $x_{1j} \stackrel{*}{\Rightarrow} p_j$, $j = 1, \dots n_1$.

Simbolul x_0 îl activează pe x_{11} şi îi cere să găsească derivarea $x_{11} \stackrel{*}{\Rightarrow} p_1$; dacă x_{11} reuşeşte, transmite lui x_0 *succes*. Simbolul x_0 îl dezactivează pe x_{11}, îl activează pe x_{12} şi îi cere să găsească derivarea $x_{12} \stackrel{*}{\Rightarrow} p_2$, etc. Dacă toate simbolurile activate de x_0 transmit succes, construcţia este terminată. Să presupunem că x_{1j} transmite *eşec*; atunci x_0 îl dezactivează pe x_{1j}, îl reactivează pe x_{1j-1} căruia îi transmite: *Mi-ai dat o derivare, dar aceasta nu este bună, încearcă alta.* Dacă x_{1j-1} reuşeşte, procesul se continuă spre dreapta; dacă nu, atunci x_0 îl dezactivează pe x_{1j-1}, îl reactivează pe x_{1j-2} căruia îi cere o altă derivare. Procesul se continuă în acest mod fie spre dreapta, fie spre stânga. Dacă se ajunge la x_{11} şi acesta nu reuşeşte să găsască o altă derivare, x_0 decide că prima regulă aleasă nu este bună şi încearcă cu următoarea regulă, adică $x_0 \to x_{21} \dots x_{2n_2}$, ş. a. m. d.

Observaţii

- Fiecare simbol devenit activ, procedează exact ca şi părintele său, alege prima regulă, activează primul simbol, etc.

- Nu se cunoaşte anticipat descompunerea $p = p_1 p_2 \dots p_{n_1}$. Deci x_{1j} transmite succes dacă reuşeşte să găsească o derivare $x_{1j} \stackrel{*}{\Rightarrow} p_j$, unde p_j este un subcuvânt oarecare al lui p, cu singura condiţie ca p_{1j} să înceapă din punctul unde s-a terminat p_{1j-1}. De exemplu, dacă $p = i_1 i_2 \dots i_8 \dots$ şi $p_1 = i_1 i_2 i_3 i_4$, $p_2 = i_5 i_6$, atunci x_{13} trebuie să găsească o derivare de forma $x_{13} \stackrel{*}{\Rightarrow} i_7 i_8 \dots$. În particular, dacă $x_{1j} \in V_T$ decizia de *succes* sau *eşec* depinde de faptul dacă x_{1j} coincide sau nu cu simbolul din p care urmează (În exemplul de mai sus, dacă x_{13} coincide sau nu cu i_7).

- Când se reactivează un simbol şi i se cere o nouă derivare, acesta reactivează ultimul simbol fiu şi îi cere acelaşi lucru.

Exemplu Considerăm următoarea gramatică G_E care generează expresii aritmetice simple. Operanzii sunt notaţi simbolic cu a, operatorii sunt + şi * iar ordinea naturală a operaţiilor este completată de paranteze.

$$\begin{cases} E \to T + E | T \\ T \to F * T | F \\ F \to (E) | a \end{cases}$$

Procesul de analiza sintactică top-down pentru cuvântul $p = a * a$ este prezentat în figura 3.1.

În această figură, revenirile în sus au fost marcate prin încadrarea într-un dreptunghi a subarborelui care a condus la un eşec. Dreptunghiurile interioare au semnificaţia unor eşecuri realizate mai devreme (este vorba de timpul de desfăşurare al procesului). Arborele corespunzător cuvântului este cel neîncadrat în vreun dreptunghi (în figură acesta este situat în partea dreaptă).

3.1.2 Programarea algoritmului top-down general

Descrierea formală a algoritmului este făcută după Aho şi Ullman, *The theory of Parsing, Translation, and Compiling*. Vom considera că regulile gramaticii sunt numerotate, iar pentru regulile cu neterminalul A în partea stangă, $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_k$ vom nota A_i indexul regulii i. Algoritmul foloseşte două stive $L1$ şi $L2$, precum şi un index ce reprezintă caracterul curent în şirul de intrare $w = a_1 a_2 \ldots a_n, n >= 0$. Pentru descrierea formală vom defini o configuraţie printr-un cvadruplu (s, i, α, β), unde s este *starea algoritmului*, i reprezintă indexul locaţiei curente (pointer pe următorul caracter din şirul de intrare), α este conţinutul stivei $L1$ iar β este conţinutul stivei $L2$. Stiva $L2$ va conţine la fiecare pas forma propoziţională din derivarea extrem stângă, iar simbolul aflat în vârful stivei va fi simbolul activ din arborele de derivare. Stiva $L1$ va conţine un istoric al alegerii regulilor şi simbolurile de intrare care au fost deja generate (*prefixele viabile* ale şirului de intrare). Pentru uşurinţa prezentării vom considera că se foloseşte un marcaj de sfârşit pentru cuvântul examinat, notat cu \$, acesta fiind pe poziţia $n + 1$.

Algoritmul foloseşte trei stări q, b şi t cu semnificaţiile urmatoare: q operaţie normală, b revenire (*backtracking*) şi t terminare. Configuraţia iniţială a algoritmului va fi $(q, 1, \lambda, S\$)$. Funcţionarea algoritmului va fi

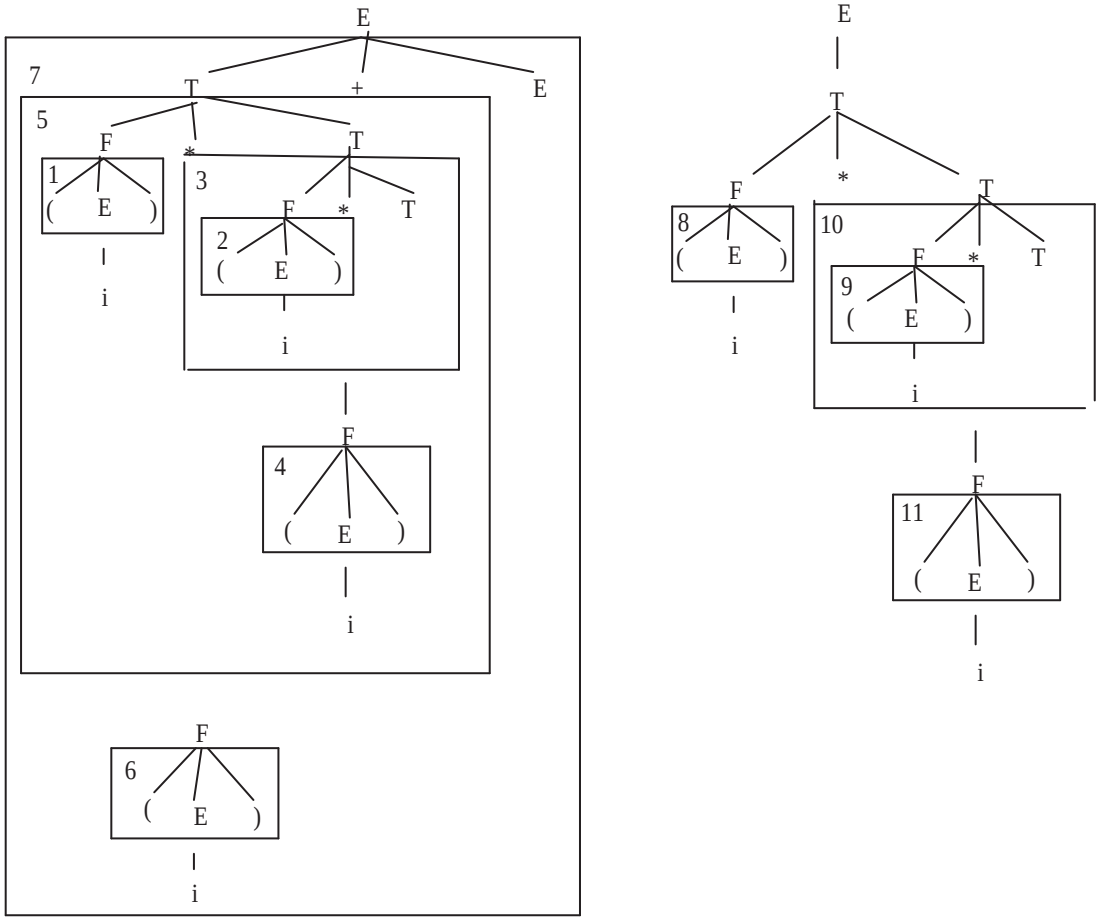

Figura 3.1: Arborele sintactic pentru $p = i * i$

o secvenţă de paşi discreţi ce pot fi descrişi printr–o relaţie pe mulţimea configuraţiilor, notată cu $\longmapsto$ca şi în cazul evoluţiei automatului push down.

Conform cu descrierea informală a algoritmului, avem următorele tipuri de paşi:

- **Expandarea arborelui**

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

  unde, $A \longrightarrow \gamma_1$ este prima regula din listă, aplicată celui mai din stânga neterminal în forma propoziţională curentă.

- **Potrivirea simbolului terminal din sirul de intrare cu terminalul activat de algoritm**

  $$(q, i, \alpha, a\beta) \longmapsto (q, i+1, \alpha a, \beta)$$

  Dacă simbolul din poziţia curentă i din şirul de intrare coincide cu următorul terminal derivat, $a_i = a$, atunci terminalul a aflat în vârful stivei $L2$ se extrage şi se scrie pe stiva $L1$. Totodată se avansează indexul i în şirul de intrare.

- **Terminare cu succes**

  $$(q, n+1, \alpha, \$) \longmapsto (t, n+1, \alpha, \lambda)$$

  S-a găsit marcajul de sfârşit al şirului examinat şi avem o formă propoziţională ce coincide cu şirul examinat. Lista regulilor aplicate pentru obţinerea derivării extrem stângi se poate obţine prin eliminarea terminalelor de pe stiva $L1$.

- **Nepotrivirea simbolului terminal activ cu următorul simbol din şirul de intrare**

  $$(q, i, \alpha, a\beta) \longmapsto (b, i, \alpha, a\beta)\ a_i \neq a$$

  Se intră în modul revenire deoarece forma propoziîonală actuală nu va genera cuvântul de examinat.

- **Revenire pe şirul examinat**

$$(b, i, \alpha a, \beta) \longmapsto (b, i-1, \alpha, a\beta), \forall a \in V_T$$

  În modul revenire se mută (**shift**) simbolurile de intrare inapoi de pe stiva $L1$ pe $L2$.

- **Încercarea unei reguli noi pentru un ultimul neterminal activat**

$$(b, i, \alpha A_j, \gamma_j \beta) \longmapsto$$

  - $(q, i, \alpha A_{j+1}, \gamma_{j+1}\beta)$, dacă γ_{j+1} este următoarea regulă din lista neterminalului A. Adică se inlocuieşte partea dreaptă a ultimei reguli încercate cu partea dreaptă a următoarei reguli.
  - blocarea algoritmului, atunci când $i = 1, A = S$ şi există doar j reguli pentru simbolul de start. În această situaţie şirul de intrare w nu aparţine limbajului generat de gramatica dată.
  - $(b, i, \alpha, A\beta)$ pentru orice altă situaţie. Este cazul când toate regulile lui A au fost încercate şi revenirea se face prin extragerea numarului regulii A_j de pe stiva $L1$ şi înlocuirea părţii drepte α_j cu neterminalul A în stiva $L2$.

EXEMPLU: Considerăm gramatica simplificată ce generează expresiile aritmetice fără paranteze, pentru care regulile sunt numerotate astfel:

$$\begin{cases} (1) & E \to T + E \\ (2) & E \to T \\ (3) & T \to F * T \\ (4) & T \to F \\ (5) & F \to a \end{cases}$$

În descriere vom utiliza E_1 pentru $T + E$, E_2 pentru T, samd. Examinarea şirului de intrare $w = a + a$ va fi descrisă de următoarea secvenţă de configuraţii.

$$
\begin{aligned}
(q, 1, \lambda, E\$) &\longmapsto (q, 1, E_1, T + E\$) \\
&\longmapsto (q, 1, E_1T_1, F * T + E\$) \\
&\longmapsto (q, 1, E_1T_1F_1, a * T + E\$) \\
&\longmapsto (q, 2, E_1T_1F_1a, *T + E\$) \\
&\longmapsto (b, 2, E_1T_1F_1a, *T + E\$) \\
&\longmapsto (b, 1, E_1T_1F_1, a * T + E\$) \\
&\longmapsto (b, 1, E_1T_1, F * T + E\$) \\
&\longmapsto (q, 1, E_1T_2, F + E\$) \\
&\longmapsto (q, 1, E_1T_2F_1, a + E\$) \\
&\longmapsto (q, 2, E_1T_2F_1a, +E\$) \\
&\longmapsto (q, 3, E_1T_1F_1a+, E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_1, T + E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_1T_1, F * T + E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_1T_1F_1, a * T + E\$) \\
&\longmapsto (q, 4, E_1T_2F_1a + E_1T_1F_1a, *T + E\$) \\
&\longmapsto (b, 4, E_1T_2F_1a + E_1T_1F_1a, *T + E\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_1T_1F_1, a * T + E\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_1T_1, F * T + E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_1T_2, F + E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_1T_2F_1, a + E\$) \\
&\longmapsto (q, 4, E_1T_2F_1a + E_1T_2F_1a, +E\$) \\
&\longmapsto (b, 4, E_1T_2F_1a + E_1T_2F_1a, +E\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_1T_2F_1, a + E\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_1T_2, F + E\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_1, T + E\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_2, T\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_2T_1, F * T\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_2T_1F_1, a * T\$) \\
&\longmapsto (q, 4, E_1T_2F_1a + E_2T_1F_1a, *T\$) \\
&\longmapsto (q, 4, E_1T_2F_1a + E_2T_1F_1a, *T\$) \\
&\longmapsto (b, 4, E_1T_2F_1a + E_2T_1F_1a, *T\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_2T_1F_1, a * T\$) \\
&\longmapsto (b, 3, E_1T_2F_1a + E_2T_1, F * T\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_2T_2, F\$) \\
&\longmapsto (q, 3, E_1T_2F_1a + E_2T_2F_1, a\$) \\
&\longmapsto (q, 4, E_1T_2F_1a + E_2T_2F_1a, \$) \\
&\longmapsto (t, 4, E_1T_2F_1a + E_2T_2F_1a, \lambda)
\end{aligned}
$$

O derivare extrem stângă se obţine prin eliminarea terminalelor şi asocierea numărului regulii cu notaţia rămasă pe stivă, adică ordinea de aplicare a regulilor este 145245.

3.1.3 Analiza top-down fără reveniri

În cazul unor gramatici cu o formă specială se poate face o analiză de tip top-down fără reveniri. Principala condiţie este ca în cazul mai multor alternative (reguli cu acelaşi simbol în stânga), să se poată decide cu precizie ramura corectă. În general o astfel de decizie se poate realiza prin analiza unor simboluri care urmează în cuvântul de analizat.

Exemplu Considerăm următoarea gramatică G care generează secvenţe de declaraţii şi instrucţiuni cuprinse intre cuvintele cheie **Program** şi **EndProgram**. Declaraţiile sunt notate simbolic cu d, iar instrucţiunile cu i. Fiecare instrucţie sau declaraţie se încheie cu ;, exceptând cazul celei ce precede **EndProgram**.

$$G \begin{cases} < program > \rightarrow Program\ D\ I\ EndProgram \\ D \rightarrow d; X \\ X \rightarrow d; X | \lambda \\ I \rightarrow iY \\ Y \rightarrow ; iY | \lambda \end{cases}$$

Să considerăm urmatorul cuvânt de analizat

```
Program
  d;
  d;
  i;
  i;
  i
EndProgram
```

O derivare extrem stângă pentru acest cuvânt este

$< program > \Rightarrow Program\ D\ I\ EndProgram \Rightarrow Program\ d; X\ I\ EndProgram$
$\Rightarrow Program\ d; d; X\ I\ EndProgram \Rightarrow Program\ d; d;\ I\ EndProgram$
$\Rightarrow Program\ d; d; iY\ EndProgram \Rightarrow Program\ d; d; i; iY\ EndProgram$
$\Rightarrow Program\ d; d; i; i; iY\ EndProgram \Rightarrow Program\ d; d; i; i; i\ EndProgram$

Pentru construcţia derivării extrem stângi se procedează astfel: iniţial se consideră simbolul de start $< program >$ pentru care se alege singura regulă disponibilă (dacă primul simbol din şirul de analizat nu coincide cu primul terminal al regulii se poate decide imediat *eroare sintactică*). Următorul simbol neterminal pentru care trebuie aleasă o regulă este D iar şirul rămas de analizat (*de generat*) începe cu d, astfel că regula aleasă va fi $D \to d; X$. Din cuvântul iniţial trebuie generată în continuare secvenţa $d; i; i; iEndProgram$ ce începe cu d iar neterminalul cel mai din stânga este X, astfel că se alege regula ce începe cu d. În continuare, din cuvântul iniţial rămâne de generat secvenţa $i; i; i; EndProgram$ ce începe cu i iar neterminalul cel mai din stânga este X, astfel că se alege regula ce $X \to \lambda$, ş.a.m.d.

Dacă se consideră gramatica ce generează expresii aritmetice simple, în forma considerată la algoritmul general top-down (3.2.9), atunci la primul pas al unei derivări extrem stângi pentru cuvântul $(a + a * a) + a$ nu se poate decide regula de ales prin citirea primului terminal (deoarece ar fi necesar să consultăm şirul rezultat până la întâlnirea operatorului + aflat după paranteza închisă.

O gramatică pentru care alegerea regulii de aplicat este **unic determinată** de următorul simbol terminal din şirul de analizat se numeşte gramatică $LL(1)$ (Left to right parsing, Leftmost derivation, 1 symbol lookahead). În multe cazuri există posibilitatea de a transforma gramatica într-una echivalentă de tip $LL(1)$. Pentru cazul particular al gramaticii pentru generarea expresiilor aritmetice, o gramatică echivalentă este următoarea:

$$\begin{cases} E \to TE' \\ E' \to +TE' | -TE' | \lambda \\ T \to FT' \\ T' \to *FT' | /FT' | \lambda \\ F \to (E) | id | num \end{cases}$$

$$\begin{cases} < bloc > \rightarrow \{< lista >\} \\ < lista > \rightarrow < instr > L \\ L \rightarrow ; < instr > L \mid \lambda \\ < instr > \rightarrow id = E | if(E) then < instr > | while(E) do < instr > | \{< lista >\} \\ E \rightarrow TE' \\ E' \rightarrow +TE' | -TE' | \lambda \\ T \rightarrow FT' \\ T' \rightarrow *FT' | /FT' | \lambda \\ F \rightarrow (E) | id | num \end{cases}$$

Figura 3.2: Gramatica pentru generarea blocurilor de instrucţiuni.

3.1.4 Programarea unui analizor sintactic. Studiu de caz

Programarea unui analizor sintactic top-down fără reveniri se poate face relativ uşor asociind câte o funcţie la fiecare neterminal. Analiza unui cuvânt revine la un şir de apeluri corespunzătoare tratării simbolurilor ce apar pe parcursul construcţiei derivării extrem stângi. Fiecare funcţie asociată conţine o singură instrucţiune *switch* cu clauze ce corepund regulilor gramaticii. Alegerea regulii se face după următoarea unitate lexicală din textul de analizat.

Să considerăm următoarea gramatică (vezi figura 3.2) ce generează un bloc de instrucţiuni de atribuire, condiţionale de tip **if** (expresie aritmetică) **then** instrucţiune sau repetitive de tip **while**. Instrucţiune poate fi o instrucţie simplă sau bloc de instrucţiuni separate prin delimitatorul ;(SEMICOLON).

Un text sursă ce poate fi analizat de această gramatică este cel din figura 3.3.

Lista de unităţi lexicale furnizate de analizorul lexical este pentru acest caz

LBRACE id LET num SEMI id LET id PLUS num SEMI if LPAR id MINUS id RPAR then LBRACE id LET id LET id minus num SEMI id LET num RBRACE SEMI id LET id ORI num RBRACE.

```
{
 a = 2;
 b = b + 1;
 if ( b-a ) then { x = x-1;
                   a = 3 };
 c = b*72
}
```

Figura 3.3: Program sursa de analizat sintactic

Un program pentru analiza sintactică top-down fără reveniri corespunzător gramaticii din figura 3.2 este reprezentat parţial în figura 3.4. Lista codurilor de unităţi lexicale (terminalele gramaticii) conţine SEMI LPAR RPAR ş.a.m.d. Presupunem că analizorul lexical funcţionează ca funcţie ce returnează codul următoarei unităţi lexicale din textul sursă. Variabila de tip întreg *token* conţine codul returnat de analizorul lexical *ALEX()*. S-au mai definit două funcţii: *err()* pentru tratarea erorilor de sintaxă depistate şi *eat(int tok)* ce consumă din textul sursă unitatea lexicală *tok* pe care ne aşteptăm să o găsim în text.

Rularea programului pentru exemplul din figura 3.3 va produce o secvenţă de apeluri recursive ca în figura 3.5.

Un pic de algebră

Vom da în cele ce urmează o definiţie riguroasă a gramaticilor $LL(k)$ (k simboluri citite în avans) împreună cu condiţiile necesare şi suficiente ca o gramatică să intre în această categorie.

Definiţie Fie $G = (V_N, V_T, S, \mathcal{P})$ o gramatică independentă de context. G se zice de tip $LL(k)$ dacă şi numai dacă oricare ar fi două derivări extrem stângi

$$S \overset{*}{\Rightarrow} uXv \Rightarrow u\alpha v \overset{*}{\Rightarrow} u\gamma$$
$$S \overset{*}{\Rightarrow} uXv \Rightarrow u\beta v \overset{*}{\Rightarrow} u\nu \quad unde\ X \to \alpha \in \mathcal{P},\ X \to \beta \in \mathcal{P},\ \gamma, \nu \in V_T^+$$

din $\gamma^k = \nu^k$ rezultă $\alpha = \beta$ (notaţia γ^k înseamnă primele k simboluri din şirul γ).

```
final int if=1, then=2, LPAR=3, RPAR=4, LBRACE=5, RBRACE=6,
PLUS=7, MINUS=8, ORI=9, DIV=10, SEMI=11, id=12,
while=13, do=14, LET=15;
void err();
int ALEX();
int token = ALEX();
void eat(int tok){
    if (tok==token) token=ALEX else err()
};

void bloc(){
  eat(LBRACE); lista(); eat(RBRACE)
};

void lista(){
  instr();L()
};

void L(){
  switch(token){
    case SEMI: eat(SEMI); instr(); L(); break;
    default: break }
};

void instr(){
  switch(token){
    case if: eat(if); eat(LPAR); E(); eat(RPAR); eat (then); instr(); break;
    case id: eat(id); eat(LET); E(); break;
    case while: eat(while); eat(LPAR); E(); eat(RPAR); eat(do); instr(); break;
    case LBRACE: eat(LBRACE); instr(); eat(RBRACE); break;
    default: printf("syntax error: if, identif, while or left brace expected");
             err()}
};
...
void E(){
 T(); Eprime();
};

void Eprime(){
switch(token) {
  case PLUS: eat(PLUS);T();Eprime();break;
  case MINUS: eat(MINUS);T();Eprime();break;
  default: break;
 }
};
...
```

```
bloc()
   eat(LBRACE);
   lista()
      instr()
         eat(id);
         eat(LET);
         E()
           T()
             F()
               eat(num);
             return_F;
             Tprime()
             return_Tprime;
           return_T;
           Eprime()
           return_Eprime;
         return_E;
      return_instr;
      L()
         eat(SEMI)
         instr()
            .... aici se recunoaste ; b = b + 1
         return_instr;
         L()
            eat(SEMI);
            instr()
              .... aici se recunoaste ; if ( ... }
            return_instr;
            L()
              .... aici se recunoaste ; c = b * 72
            return_L();
         return_L;
      return_L;
   return_lista;
   eat(RBRACE);
return_bloc;
```

Figura 3.5: Execuţia analizei lexicale pentru textul sursa

Pentru $k = 1$ se obţine cazul gramaticilor din secţiunile precedente. Restricţia asupra numărului de simboluri citite în avans restrânge drastic mulţimea limbajelor ce por fi analizate cu astfel de gramatici. Un inconvenient major pentru $k > 1$ este creşterea exponenţială a dimensiunii tabelei de predicţie (câte o intrare în tabel pentru fiecare combinaţie de k simboluri. O variantă mai eficientă de analiză se obţine cu gramatici $LR(1)$ (Left to right parsing, Rightmost derivation, 1 symbol lookahead).

Pentru orice $\alpha \in V_G^+,\ X \in V_N,\ X \to \alpha \in \mathcal{P}$ vom considera următoarele mulţimi:

$$Prim(\alpha) = \{a \in V_T | \alpha \stackrel{*}{\Rightarrow} a\beta,\ \beta \in V_G^*\}$$
$$Urm(X) = \{a \in V_T | S \stackrel{*}{\Rightarrow} uXv,\ a \in Prim(v)\}$$
$$SD(X, \alpha) = \{a \in V_T |\ a \in Prim(\alpha)\ sau\ (\alpha \stackrel{*}{\Rightarrow} \lambda\ si\ a \in Urm(X)\ )\}$$
$$NULL(G) = \{X \in V_N | X \stackrel{*}{\Rightarrow} \lambda\}$$

Dacă un terminal $a \in Prim(\alpha)$ atunci a poate apărea pe prima poziţie într-un şir derivat din α. Mulţimea $SD(X, \alpha)$ poartă denumirea de *mulţimea simbolurilor directoare* asociate regulii $X \to \alpha$, iar $NULL(G)$ conţine neterminalele ce se pot şterge (în unul sau mai mulţi paşi) cu reguli din G.

Teoremă.

$$G \in LL(1) \Leftrightarrow SD(X, \alpha) \cap SD(X, \beta) = \Phi,\ \forall X \to \alpha,\ X \to \beta \in \mathcal{P},\ \beta \neq \alpha$$

Determinarea mulţimilor definite anterior se poate face uşor. Pentru $NULL(G)$, se poate aplica algoritmul definit la eliminarea regulilor de ştergere pentru o gramatică independentă de context.

$NULL = \{X \in V_N | X \to \lambda \in \mathcal{P}\}$
repeat
(1) $AUX = NULL$;
(2) $NULL = NULL \cup \{X \in V_N | X \to p, p \in AUX^+\}$;
until $NULL = AUX$ □

Calculul mulţimilor $Prim(X)$ şi $Urm(X)$ se poate face cu algoritmul următor:

Pas 1: { Iniţializare }
$Prim(a) = \{a\}; \forall a \in V_T$
$Prim(X) = Urm(Y) = \Phi; \forall X, Y \in V_N$
Pas 2: **Repeat**
Pentru fiecare regulă $X \to Y_1Y_2 \dots Y_k$ **Do**
For$(i = 1; i \leq k; i++)$
(2.1)**if** $(i = 1$ *sau* $Y_1Y_2 \dots Y_{i-1} \in NULL^+)$
$Prim(X) = Prim(X) \cup Prim(Y_i);$
(2.2)**if** $(i = k$ *sau* $Y_{i+1} \dots Y_k \in NULL^+)$
$Urm(Y_i) = Urm(Y_i) \cup Urm(X);$
(2.3)**For** $(j = i + 1\ j \leq k; j++)$
if $(j = i + 1$ *sau* $Y_{i+1} \dots Y_{j-1} \in NULL^+)$
$Urm(Y_i) = Urm(Y_i) \cup Prim(Y_j);$
end for
end for
end do
Until $Prim(X), Urm(X)$ nu se schimbă la o iteraţie.□

Calculul mulţimilor $Prim(\alpha)$ revine la definiţia recursivă

$$Prim(X\alpha) = \begin{cases} Prim(X) & daca\ X \notin NULL(G) \\ Prim(X) \cup Prim(\alpha) & daca\ X \in NULL(G) \end{cases}$$

Dacă se consideră gramatica ce generează expresii aritmetice 3.1.3, atunci rezultatul aplicării algoritmilor este prezentat în tabelul următor:

	$NULL$	$Prim$	Urm
E		$($, i	$)$
E'	DA	$+$, $-$	$)$
T		$($, i	$+$, $-$, $)$
T'	DA	$*$, $/$	$+$, $-$, $)$
F		$($, i	$*$, $/$, $)$

iar mulţimile simbolurilor directoare sunt
$SD(E, TE') = \{($, $i\}$
$SD(E', +TE') = \{+\}$
$SD(E', \lambda) = \{)\}$
$SD(T, FT') = \{($, $i\}$

$SD(T', *FT') = \{*\}$
$SD(T', \lambda) = \{+)\}$
$SD(F, (E)) = \{(\}$
$SD(F, id) = \{id\}$
$SD(F, num) = \{num\}$

Funcţiile C sau $Java$ corespunzătoare neterminalelor se modifică puţin, în sensul că vom avea clauze corespunzătoare terminalelor ce se regăsesc în mulţimile simbolurilor directoare asociate regulilor. Dacă la reconstrucţia derivării trebuie aplicată o regulă pentru X şi următorul terminal din textul de analizat nu face parte din mulţimea simbolurilor directoare asociate vreunei reguli cu X în stânga, atunci se apelează modulul de tratare a erorilor sintactice.

Pentru cazul gramaticii anterioare, câteva din secvenţele de cod asociate neterminalelor sunt prezentate în continuare.

```
...
void E(){
switch(token) {
  case LPAR:
  case id:
  case num: T(); Eprime(); break;
  default: printf("syntax error: (,identifier or number expected");
           err()
 }
}

void Tprime(){
switch(token) {
  case ORI: eat(ORI);F();Tprime();break;
  case DIV: eat(DIV);F();Tprime();break;
  case PLUS:
  case RPAR: break;
  default: printf("syntax error: *,/,+,) expected");
           err()
 }
}
```

Tratarea erorilor este dependentă de proiectantul compilatorului

şi limbaj. Principial, există trei modalităţi de tratare.

- Se opreşte analiza sintactică în punctul unde s-a depistat o eroare, cu transmiterea unui mesaj prietenos către utilizator, de exemplu: *Eroare sintactică: aşteptam să urmeze delimitator de instrucţiune. Mai învaţă sintaxa limbajului! BYE!.*

- Se încearcă repararea greşelii de sintaxă inserând în textul sursă un simbol din mulţimea de simboluri directoare asociate regulii ce s-a aplicat. Desigur că este suficient să presupunem că am inserat în text, astfel încât analiza poate continua (desigur nu vom uita să anunţăm utilizatorul că nu cunoaşte regulile de sintaxă şi l-am corectat noi!). Inserarea poate conduce la cicluri infinite, astfel că nu este recomandabilă totdeauna.

- Se încearcă găsirea unui simbol ce se potriveşte ignorând toate terminalele textului sursă până se întâlneşte un simbol din mulţimea simbolurilor directoare. Se transmite acelaşi mesaj prietenos către autorul textului sursă, apoi se continuă analiza.

3.2 Algoritmi BOTTOM-UP

3.2.1 Gramatici cu precedenţă simplă

Fie $G = (V_N, V_T, x_0, \mathcal{P})$ o gramatică de tipul doi şi $p \in L(G)$ o formă propoziţională, adică un cuvânt peste alfabetul general V_G astfel încât $x_0 \stackrel{*}{\Rightarrow} p$. Vom nota cu $\mathcal{A}_{x_0,p}$ arborele de derivare care are rădăcina x_0 şi frontiera p.

Definiţia 1. Vom spune că f este o frază simplă a lui p dacă f este un subcuvânt al lui p şi în plus:

- $\exists x \in V_N,\ cu\ x \to f \in \mathcal{P}$;

- $\mathcal{A}_{x,f} \subset \mathcal{A}_{x_0,p}$.

Exemplu. Considerăm gramatica G_E 3.2.9 şi forma propoziţională $p = T * F + a * (E + T)$. Arborele de derivare este prezentat în figura 3.6. Se poate observa că frazele simple sunt $T * F, a, E + T$.

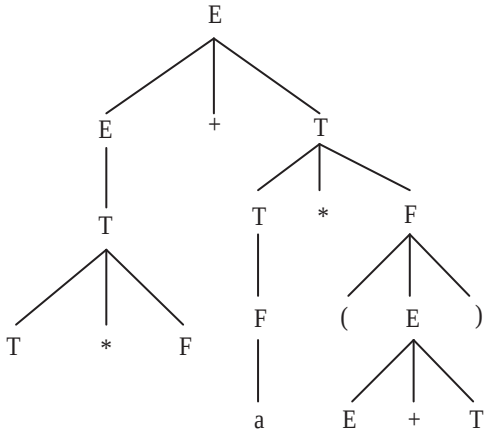

Figura 3.6: Arborele de derivare corespunzător derivării lui $p = T * F + a * (E + T)$

(1) Iniţializare p;
(2) Se determină f_p, fraza simplă stângă a lui p;
(3) Se determină regula $x \to f_p$;
(4) Se **reduce** fraza simplă stângă, adică dacă $p = rf_ps$, se pune $p = rxs$;
GOTO(2).

Figura 3.7: Algoritmul bottom-up

Observaţie. Subcuvântul F respectă prima condiţie dar nu este frază simplă, întrucât nu este satisfăcută condiţia a doua din definiţie.

Definiţia 2. Fraza simplă cea mai din stânga poartă denumirea de *frază simplă stângă.*

Vom nota fraza simplă stângă corespunzătoare lui p cu f_p în exemplul nostru, $f_p = T * F$. După cum vom vedea în continuare, fraza simplă stângă are un rol important în algoritmii de analiză sintactică bottom-up. În principiu, aceşti algoritmi prevăd următorii paşi (descrişi de figura 3.7):

Regulile determinate la pasul (3), aplicate în ordine inversă, ne vor

da derivarea corespunzătoare lui p. Să mai observăm că în acest mod arborele de derivare se construieşte de jos în sus (bottom-up). Problema cea mai dificilă este desigur determinarea frazei simple stângi de la pasul (2). În principiu ar trebui cunoscut anticipat arborele de derivare corespunzător lui p, dar tocmai acesta este scopul analizei sintactice. După cum vom vedea, această dificultate se poate elimina, deci putem determina fraza simplă stângă fără a cunoaşte acest arbore. De fapt, fraza simplă stângă este legată de anumite proprietăţi ale gramaticii pe care le tratăm în continuare.

3.2.2 Relaţii de precedenţă

Definiţia 3. Fie $x, y \in V_G$ două simboluri oarecare ale gramaticii. Vom spune că:
(1) $x \prec y$ (x *precede pe* y) dacă $\exists p$ astfel încât $p = rxys$, $x \notin f_p,\ y \in f_p$;
(2) $x \pm y$ (x *este egal cu* y) dacă $\exists p$ astfel încât $p = rxys$, $x \in f_p,\ y \in f_p$;
(2) $x \succ y$ (y *succede lui* x) dacă $\exists p$ astfel încât $p = rxys$, $x \in f_p,\ y \notin f_p$;

Relaţiile $\prec \pm \succ$ se numesc *relaţii de precedenţă simple* (atunci când nu există posibilitatea de confuzie se folosesc denumirile *mai mic, egal, mai mare* pentru relaţiile de precedenţă). Să observăm că aceste relaţii nu se exclud reciproc, deci că putem avea, de exemplu, $x \prec y$ şi în acelaşi timp $x \succ y$, întrucât pot exista cazuri de gramatici în care putem găsi două forme propoziţionale p_1, p_2 astfel încât să avem simultan cele două relaţii. Să mai facem de asemenea observaţia că există mai multe tipuri de astfel de relaţii, cu definiţii asemănătoare, întrucât relaţiile trebuie să satisfacă condiţii suplimentare care depind şi de gramatică; pentru toate aceste tipuri de relaţii vom utiliza aceleaşi notaţii.

Definiţia 4. O gramatică de tipul 2 spunem că este cu *precedenţă simplă* dacă între oricare două simboluri ale gramaticii **există cel mult o relaţie** de precedenţă.

Pentru algoritmul de analiză sintactică ascendentă este important ca să nu existe reguli cu părţile drepte identice; altminteri, în cadrul pasului (4) din algoritmul de analiză sintactică nu se poate decide la cine să se facă reducerea frazei simple stângi. De asemenea regulile de ştergere nu sunt admise, pentru acest caz noţiunea de frază simplă neavând sens. Aceste condiţii presupunem în mod sistematic că sunt satisfăcute.

Fie acum $V_G = \{x_1, x_2, \ldots, x_n\}$ alfabetul general, unde am adoptat o anumită ordine a simbolurilor. Definim matricea de precedenţă a gramaticii,

$$M = (a_{ij}),\ unde\ a_{ij} = \begin{cases} \prec & daca\ x_i \prec x_j \\ \pm & daca\ x_i \pm x_j \\ \succ & daca\ x_i \succ x_j \end{cases}$$

Exemplu. Considerăm următoarea gramatică

$$G = (\{A, B\}, \{0, 1\}, A, \{A \to A0|1B, B \to 1\}).$$

Matricea de precedenţă va fi

	A	B	0	1
A			$\prec$	
B			$\succ$	
0			$\succ$	
1		$\pm$	$\succ$	$\prec$

3.2.3 Proprietăţi ale gramaticilor cu precedenţă simplă

Proprietatea 1. Fie $p = a_1 \ldots a_n$ o formă propoziţională într-o gramatică cu precedenţă simplă (a_i sunt simboluri neterminale sau terminale ale gramaticii). Atunci $f_p = a_i \ldots a_j$ dacă şi numai dacă

$$a_1 \overset{\pm}{\prec} a_2 \ldots \overset{\pm}{\prec} a_{i-1} \prec a_i \pm a_{i+1} \ldots \pm a_j \succ a_{j+1}\ (*)$$

Demonstraţia acestei proprietăţi se efectuează considerând toate structurile de arbori de derivare posibile într-o situaţie dată. Vom exemplifica această idee pentru cuvânul $p = a_1 \ldots a_7$. Dacă presupunem că $f_p = a_3a_4a_5$, atunci, conform definiţiei frazei simple stângi, avem

$$a_2 \prec a_4,\ a_4 \pm a_5,\ a_5 \succ a_6$$

Mai trebuie arătat că $a_1 \overset{\pm}{\prec} a_2$. În acest scop vom considera toate posibilităţile de structuri arborescente; o parte din ele sunt prezentate în figura 3.8.

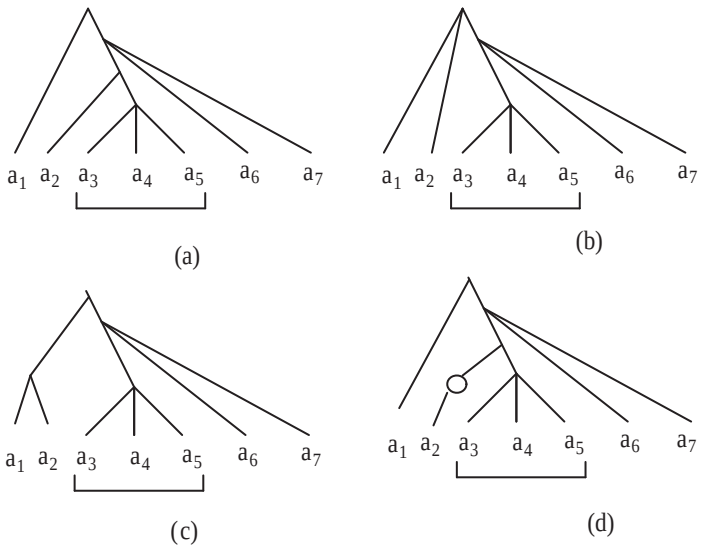

Figura 3.8: Arbori de derivare posibili

Se poate observa că în ipoteza că $f_p = a_3a_4a_5$, singurele situaţii posibile sunt (a) şi (b). Celelalte situaţii contrazic această ipoteză, de exemplu, în cazurile (c) şi (d) fraza simplă stângă ar fi , respectiv, a_1a_2 sau a_2. Acum, în cazul (a), prin reducerea succesivă a frazei simple stângi, vom obţine un arbore în care fraza simplă stângă va fi a_1a_2, adică $a_1 \pm a_2$ iar în cazul (b), fraza simplă stângă va fi a_2 şi atunci $a_1 \prec a_2$.

Implicaţia inversă se bazează pe cea directă; se presupune că $f_p = a_l \ldots a_k$cu $l \neq i$ şi $k \neq j$ şi se consideră în continuare toate posibilităţile de poziţionare a indicilor l, k faţă de i, j. De exemplu, dacă $f_p = a_2a_3$ atunci, conform implicaţiei directe, avem

$$a_1 \prec a_2 \pm a_3 \succ a_4$$

ceea ce contrazice relaţia (*), conform căreia $a_3 \pm a_4$ iar gramatica are precedenţă simplă.

Observaţie La delimitarea frazei simple stângi se procedează astfel: parcurgem şirul de simboluri de la stânga la dreapta până se găseşte

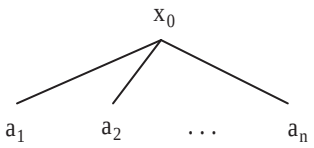

Figura 3.9: Arborele de derivare pentru $ni = 1$

relaţia $\succ$ (sau marginea dreaptă a formei propoziţionale) pentru determinarea ultimului simbol din f_p, apoi se parcurge şirul spre stânga, trecând peste relaţii $\pm$, până la găsirea unei relaţii $\prec$ (sau marginea dreaptă a formei propoziţionale) pentru determinarea primului simbol din f_p.

Proprietatea 2. O gramatică cu precedenţă simplă este neambiguă.

Schiţă de demonstraţie. Fie p o formă propoziţională şi $\mathcal{A}_{x_0,p}$ arborele de derivare corespunzător. Vom arăta că acest arbore este unicul arbore de derivare care are rădăcina x_0 şi frontiera p. Procedăm prin inducţie asupra numărului de noduri interne ni. Dacă $ni = 1$ atunci arborele de derivare va avea forma din figura 3.9 şi unicitatea acestuia este evidentă. Presupunem acum că proprietatea este adevărată pentru un ni oarecare şi considerăm cazul $ni + 1$. Să presupunem că ar exista doi arbori diferiţi cu rădăcina x_0 şi frontiera p; fie aceştia $\mathcal{A}'_{x_0,p}$ şi $\mathcal{A}_{x_0,p}$.

Deoarece gramatica este cu precedenţă simplă, rezultă că fraza simplă stângă, care este aceiaşi în cei doi arbori, este situată în aceiaşi poziţie, $p = r f_p s$. Efectuăm reducerea frazei simple f_p (conform regulii aplicate $x \to f_p$) şi vom obţine arborii $\mathcal{A}'_{x_0,q}$ şi $\mathcal{A}_{x_0,q}$ unde $q = rXs$ iar cei doi arbori trebuie să fie diferiţi. Dar numărul de noduri interne este ni, în contrazicere cu ipoteza.

3.2.4 Determinarea relaţiilor de precedenţă pentru gramatici cu precedenţă simplă

Aşa cum s-a precizat deja, relaţiile de precedenţă sunt proprietăţi intrinseci ale gramaticii, nu depind de contextul în care se află cele

două simboluri. Prin urmare, cunoaşterea anticipată a acestor relaţii împreună cu proprietatea 1, rezolvă complet problema pasului 2 de la algoritmul de analiză bottom-up, adică determinarea frazei simple stângi. În acest subparagraf vom prezenta o teoremă de caracterizare pe baza căreia se pot determina relaţiile de precedenţă simple şi implicit faptul dacă gramatica este sau nu cu precedenţă simplă.

Vom defini două relaţii specifice gramaticilor şi care vor fi utilizate în continuare, numite, respectiv, **F** (First) şi **L** (Last). Fie $x \in V_N$ şi $y \in V_G$ două simboluri ale gramaticii. Vom spune că

$$xFy \; daca \; \exists x \to yu \in \mathcal{P}, \; u \in V_G^*$$
$$xLy \; daca \; \exists x \to uy \in \mathcal{P}, \; u \in V_G^*$$

Închiderile tranzitive, respectiv, tranzitive şi reflexive ale acestor relaţii le vom nota cu $F^+, \; L^+$ şi respectiv, $F^*, \; L^*$. Există numeroşi algoritmi direcţi care permit calcularea acestei închideri,cu complexitate redusă.

Teorema 2. Avem

(1) $x \prec y \Leftrightarrow \exists u \to \ldots xv \ldots \in \mathcal{P}, vF^+y$;
(2) $x \pm y \Leftrightarrow \exists u \to \ldots xy \ldots \in \mathcal{P}$;
(3) $x \succ y \Leftrightarrow \exists u \to \ldots vw \ldots \in \mathcal{P}, vL^+x, wF^*y$;

Schiţă de demonstraţie. Se analizează posibilităţile de arbori de derivare care răspund cerinţelor teoremei. De exemplu, pentru cazul (1), trebuie să existe structura de arbore ca în figura 3.10.

Este clar că dacă $x \prec y$, atunci conform definiţei frazei simple stângi, trebuie să existe p astfel încât $p = rxys, \; x \notin f_p, \; y \in f_p$, adică să existe structura din 3.10. Invers, dacă există o astfel de structură, atunci fie o formă propoziţională care conţine u. Efectuăm reduceri succesive până când vom obţine o formă propoziţională pentru care u aparţine frazei simple stângi; la arborele astfel obţinut, adăugăm subarborele de mai sus. Vom avea în mod evident $x \notin f_p, \; y \in f_p$.

3.2.5 Studiu de caz

Să considerăm cazul gramaticii pentru generarea expresiilor aritmetice simple G_E, cu regulile obişnuite

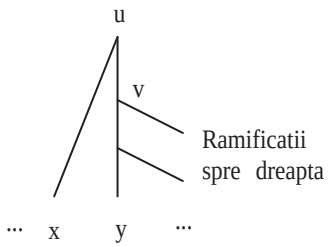

Figura 3.10: Arborele de derivare corespunzator cazului $x \prec y$

$$\begin{cases} E \to E + T | T \\ T \to T * F | F \\ F \to (E) | a \end{cases}$$

Relaţiile First, Last şi închiderile acestora sunt:

$E\ F\ \{E, T\}$	$E\ F^+\ \{E, T, F, (, a\}$	$E\ F^*\ \{E, T, F, (, a\}$
$T\ F\ \{T, F\}$	$T\ F^+\ \{T, F, (, a\}$	$T\ F^*\ \{T, F, (, a\}$
$F\ F\ \{(, a\}$	$F\ F^+\ \{(, a\}$	$F\ F^*\ \{F, (, a\}$

$E\ L\ \{T\}$	$E\ L^+\ \{T, F, a,)\}$	$E\ F^*\ \{E, T, F, a,)\}$
$T\ L\ \{F\}$	$T\ L^+\ \{F, a,)\}$	$T\ F^*\ \{T, F, a,)\}$
$F\ L\ \{a,)\}$	$F\ L^+\ \{a,)\}$	$F\ F^*\ \{F, a,)\}$

Conform teoremei de calcul a relaţiilor de precedenţă vom obţine matricea de precedenţă (vezi figura 3.11) asociată simbolurilor gramaticii G_E.

Deci gramatica G_E nu are proprietatea de precedenţă simplă, astfel că nu poate fi aplicat direct algoritmul de analiză sintactică bottom up.

Vom exemplifica algoritmul de analiză pentru gramatica din figura 3.12, matricea de precedenţă asociată este cea alăturată.

Reconstituirea derivării cuvântului $p = aaabaababbbabb$ este prezentată în tabelul 3.13, unde fiecare linie corespunde formei propoziţionale, în care fraza simplă stângă este subliniată, precedată de regula identificată.

	E	T	F	$+$	$*$	$($	$)$	a
E				$\pm$			$\pm$	
T				$\succ$	$\pm$			
F				$\succ$	$\succ$			
$+$		$\prec \pm$	$\prec$			$\prec$		$\prec$
$*$			$\pm$			$\prec$		$\prec$
$($	$\prec \pm$	$\prec$	$\prec$			$\prec$		$\prec$
$)$				$\succ$	$\succ$		$\succ$	
a				$\succ$	$\succ$		$\succ$	

Figura 3.11: Matricea de precedenţă pentru G_E

$G: \ S \to aSSb \mid ab,$

	S	a	b
S	$\pm$	$\prec$	$\pm$
a	$\pm$	$\prec$	$\pm$
b	$\succ$	$\succ$	$\succ$

Figura 3.12: Exemplu de gramatică cu precedenţă simplă

Regula	*Forma propozitionala*
	$a \prec a \prec \underline{a \pm b} \succ aababbbabb$
$S \to ab$	$a \prec a \pm S \prec a \prec \underline{a \pm b} \succ abbbabb$
$S \to ab$	$a \prec a \pm S \prec a \pm S \prec \underline{a \pm b} \succ bbabb$
$S \to ab$	$a \prec a \pm S \prec \underline{a \pm S \pm S \pm b} \succ babb$
$S \to aSSb$	$a \prec \underline{a \pm S \pm S \pm b} \succ abb$
$S \to aSSb$	$a \pm S \prec \underline{a \pm b} \succ b$
$S \to ab$	$a \pm S \pm S \pm b$
$S \to aSSb$	S

Figura 3.13: Reconstrucţia derivării cuvântului $p = aaabaababbbabb$

3.2.6 Gramatici operatoriale

Gramaticile operatoriale sunt gramatici cu o structură specială a regulilor de rescriere, în care anumite simboluri sunt considerate operatori iar celelalte operanzi, operatorii trebuind să separe operanzii. Această structură este preluată de la gramaticile care generează expresiile aritmetice iar principiul de analiză este de asemenea preluat de la algoritmul de evaluare a expresiilor aritmetice. În continuare vom prezenta acest principiu pentru evaluarea expresiilor aritmetice fără paranteze, cazul expresiilor cu paranteze putându-se reduce la cel fără paranteze, de exemplu utilizând tehnica de modificare a ponderilor operatorilor la întâlnirea parantezelor.

În primul rând se atribuie operatorilor anumite ponderi care vor defini ordinea de efectuare a operaţiilor. De obicei, se consideră $p(+) = p(-) = 1 < 2 = p(*) = p(/)$, ceea ce înseamnă că mai întâi se fac adunările şi scăderile, apoi înmulţirile şi împărţirile, etc. Efectuarea unei operaţii depinde de contextul în care se află operatorul respectiv, şi anume, dacă ponderea operatorului precedent este mai mare sau egală cu ponderea operatorului curent, atunci se poate efectua operaţia definită de operatorul precedent. Din acest motiv, acest tip de gramatici se numesc *cu operator de precedenţă* (operator precedence grammars).

Putem realiza un algoritm simplu de evaluare utilizând două stive

- **P**- stiva operatorilor;
- **S**- stiva operanzilor.

Vom utiliza indicii i şi k pentru a indica nivelele celor două stive. Prin urmare, notaţia p_i, respectiv s_k au sensul de operatorul, respectiv operandul din stivă aflat pe nivelul i, respectiv k. Pentru simplificare, vom utiliza aceiaşi notaţie p_i atât pentru operator cât şi pentru ponderea operatorului respectiv.

Algoritmul de evaluare:
PAS 1: *Iniţializări*: Se introduce primul operator şi primul operand în stivele P şi S şi se iniţializează nivelele celor două stive, $i = k = 2$;
PAS 2: Se introduce următorul operator în stiva P şi se actualizează nivelul, $i = i + 1$;

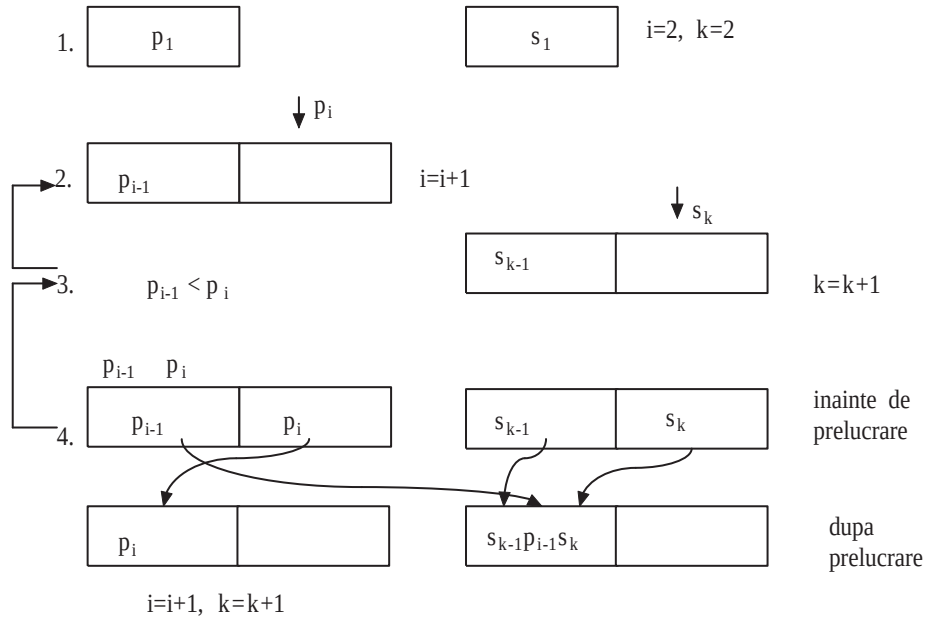

Figura 3.14: Prelucrările efectuate asupra celor două stive

PAS 3: Dacă $p_{i-1} < p_i$ atunci se introduce următorul operand în lista S, se actualizează nivelul stivei, $k = k + 1$, şi se revine la pasul 2;
PAS 4: Dacă $p_{i-1} \geq p_i$ atunci:

în stiva P: p_{i-1} se înlocuieşte cu p_i;

în stiva S: s_{k-1} se înlocuieşte cu $s_{k-1}p_{i-1}s_k$;

se actualizează nivelele celor două stive, $i = i - 1; k = k - 1$ şi se revine la pasul 3.

Observaţie. Pentru uniformitatea algoritmului se introduce operatorul marcaj, notat cu # , cu ponderea cea mai mică; orice expresie se încadrează între două marcaje iar oprirea algoritmului are loc la pasul 4 în cazul în care $p_1 = p_2 = \#$. Schematic, acest algoritm este prezentat în figura 3.14.

Exemplu. Tabelul 3.15 conţine starea dinamică a celor două stive pentru expresia $\#a + b * c/d - f\#$. Menţionăm că în cadrul paşilor 2 şi 4 s-a completat câte un rând nou în tabel.

Observaţie. Ponderile operatorilor sunt legate de ierarhia lor în gramatică, cu cât un operator este mai *jos* cu atât ponderea lui este

	1	2	3	4	1	2	3	4
1	$\#_0$				a			
2	$\#_0$	$+_1$			a	b		
3	$\#_0$	$+_1$	$-_1$		a	b		
4	$\#_0$	$-_1$			$a+b$	c		
5	$\#_0$	$-_1$	$*_2$		$a+b$	c	d	
6	$\#_0$	$-_1$	$*_2$	$/_2$	$a+b$	c	d	
7	$\#_0$	$-_1$	$/_2$		$a+b$	$c*d$	f	
8	$\#_0$	$-_1$	$/_2$	$\#_0$	$a+b$	$c*d$	f	
9	$\#_0$	$-_1$	$\#_0$		$a+b$	$c*d/f$		
10	$\#_0$	$\#_0$			$a+b-c*d/f$			

Figura 3.15: Evaluarea expresiei $\#a+b-c*d/f\#$

mai mare. Acest fapt este ilustat în figura 3.16.

Pentru cazul expresiilor aritmetice cu paranteze algoritmul se păstrează, dar este aplicat după preprocesarea expresiei prin modificarea dinamică a ponderilor operatorilor şi renunţarea la paranteze. Se parcurge expresia de la stânga la dreapta alocând ponderi operatorilor; la întâlnirea unei paranteze deschise ponderile se incrementează cu 10, iar la întâlnirea unei paranteze închise se decrementează ponderile operatorilor cu 10. În continuare parantezele sunt ignorate. De exemplu, expresia

$$a*(b-c*d)-(x+y/(z-t))*h$$

devine prin preprocesare

$$a\,*_2\,b\,-_{11}\,c\,*_{12}\,d\,-_1\,x\,+_{11}\,y/_{12}z\,-_{21}\,t*_2$$

ceea ce va asigura ordinea corectă de evaluare.

3.2.7 Gramatici operatoriale

Fie $G=(V_N,V_T,x_0,\mathcal{P})$ o gramatică de tipul 2. Vom considera că simbolurile neterminale V_N sunt operanzii limbajului iar simbolurile terminale V_T sunt operatorii. Să observăm că în cazul gramaticii G_E

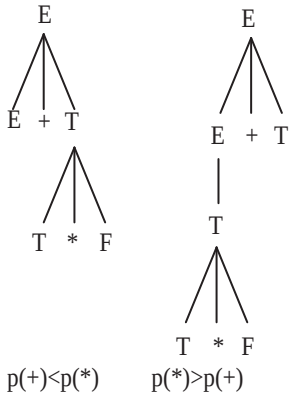

Figura 3.16: Ierarhia operatorilor

care generează expresii aritmetice simple o asemenea convenţie este justificată, deoarece oricare din neterminalele E, T, F derivează în terminalul a (generic, acest terminal reprezintă operanzii), sau într-o expresie (eventual între paranteze) care are rolul de operand.

Definiţia 1. Vom spune că o gramatică este operatorială dacă regulile de rescriere au forma

$$X \to N_i T_i N_{i+1} T_{i+1} \dots T_j N_{j+1},$$

unde N_k sunt neterminale (*operanzi*), inclusiv cuvântul vid, iar T_k sunt terminale (*operatori*).

Observaţie. Într-o gramatică operatorială orice formă propoziţională are structura:

$$p = N_1 T_1 N_2 T_2 \dots T_n N_{n+1}$$

Aceasta înseamnă că într-o formă propoziţională a unei gramatici operatoriale între oricare doi operanzi există cel puţin un operator; pot însă exista mai mulţi operatori alăturaţi, de exemplu, $\dots((a \dots$.

Definiţia 2. Fie $p = N_1 T_1 N_2 T_2 \dots T_n N_{n+1}$ o formă propoziţională într-o gramatică operatorială. Vom spune că f este o *frază simplă* a lui p dacă $f = N_i T_i N_{i+1} T_{i+1} \dots T_j N_{j+1}$ este un subcuvânt al lui p care conţine *cel puţin un simbol terminal* şi în plus:

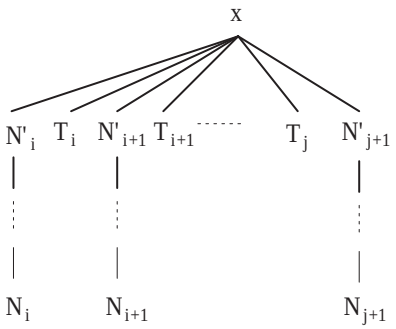

Figura 3.17: Structura subarborelui

(1) $\exists x \to N'_i T_i N'_{i+1} T_{i+1} \dots T_j N'_{j+1} \in \mathcal{P}$, şi $N'_k \stackrel{*}{\Rightarrow} N_k \ \forall k = i, \dots, j$.
(2) $\mathcal{A}_{x,f} \subset \mathcal{A}_{x_0,p}$.
Structura de arbore corespunzătoare unei fraze simple este prezentată în figura 3.17.

Ca şi în cazul gramaticilor cu precedenţă simplă, fraza simplă cea mai din stânga poartă denumirea de *frază simplă stângă*. Principiul de analiză sintactică este identic cu cel de la gramatici cu precedenţă simplă. Problema principală este şi aici determinarea frazei simple stângi. Aceasta operaţie poate fi făcută utilizând relaţiile de precedenţă dintre simbolurile gramaticii care pentru gramatici operatoriale au o definiţie puţin modificată.

Definiţia 3. Fie $x, y \in V_T$. Vom spune că:
(1) $x \stackrel{\circ}{<} y$ (*x precede pe y*) dacă $\exists p$ astfel încât $p = rxNys,\ N \in V_N,\ x \notin f_p,\ y \in f_p$;
(2) $x \stackrel{\circ}{=} y$ (*x este egal cu y*) dacă $\exists p$ astfel încât $p = rxNys,\ N \in V_N,\ x \notin f_p,\ y \in f_p$;
(2) $x \stackrel{\circ}{>} y$ (*y succede lui x*) dacă $\exists p$ astfel încât $p = rxys,\ N \in V_N,\ x \in f_p,\ y \notin f_p$;

Matricea de precedenţă se defineşte numai pe mulţimea simbolurilor terminale, ceea ce conduce la o simplificare a algoritmului de analiză.

Definiţia 4. O gramatică operatorială se spune că are *precedenţă simplă* dacă între două simboluri ale gramaticii există cel mult o relaţie

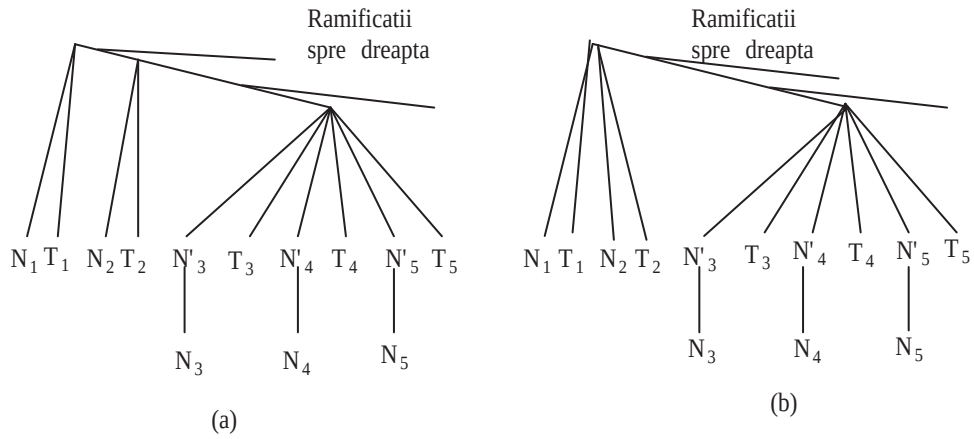

Figura 3.18: Structurile posibile de arbori

de precedenţă.

Ca şi la celelalte tipuri de gramatici, în vederea efectuării pasului 3 din algoritmul de analiză, vom presupune că nu există reguli cu părţile drepte identice. De asemenea, sunt valabile proprietăţile de caracterizare şi de neambiguitate.

Proprietatea 1. Fie $p = N_1T_1N_2T_2 \dots T_nN_{n+1}$ o formă propoziţională într-o gramatică cu precedenţă simplă. Atunci $f_p = N_iT_iN_{i+1}T_{i+1} \dots T_jN_{j+1}$ dacă şi numai dacă

$$T_1 \overset{\circ}{\leq} T_2 \dots \overset{\circ}{\leq} T_{i-1} \overset{\circ}{<} T_i \overset{\circ}{=} T_{i+1} \dots \overset{\circ}{=} T_j \overset{\circ}{>} T_{j+1} \ (*)$$

Schiţă de demonstraţie. Presupunem că $p = N_1T_1N_2T_2 \dots T_nN_{n+1}$ şi că fraza simplă stângă este $f_p = N_3T_3N_4T_4N_5T_5$. Atunci din definiţia frazei simple stângi, rezultă $T_2 \overset{\circ}{<} T_3 \overset{\circ}{=} T_4 \overset{\circ}{=} T_5 \overset{\circ}{>} T_6$. Mai trebuie să arătăm că $T_1 \overset{\circ}{<} T_2$ sau $T_1 \overset{\circ}{=} T_2$. În acest scop analizăm structurile posibile de arbori (figura 3.18).

Efectuăm reduceri succesive până când $T_2 \in f_p$. În acest moment avem $T_1 \overset{\circ}{<} T_2$ pentru cazul (a) şi $T_1 \overset{\circ}{=} T_2$ pentru cazul (b). Pentru implicaţia inversă presupunem că $f_p = N_kT_kN_{k+1}T_{k+1} \dots T_lN_{l+1}, l \neq$

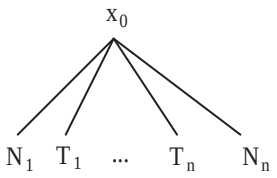

Figura 3.19: Arborele de derivare pentru $ni = 1$

$j, k \neq i$ şi considerăm toate cazurile de poziţionare a indicilor k, l în raport cu i, j. De exemplu, dacă $l < j$ atunci, în conformitate cu implicaţia directă, rezultă $T_l \overset{\circ}{>} T_{l+1}$ iar conform relaţiei $(*)$ din ipoteză, avem $T_l \overset{\circ}{<} T_{l+1}$ sau $T_l \overset{\circ}{=} T_{l+1}$ ceea ce contrazice ipoteza iniţială conform căreia gramatica este cu precedenţă simplă.

Proprietatea 2. O gramatică operatorială cu precedenţă simplă este neambiguă.

Schiţă de demonstraţie. Fie p o formă propoziţională şi $\mathcal{A}_{x_0,p}$ arborele de derivare corespunzător. Vom arăta că acest arbore este unicul arbore de derivare care are rădăcina x_0 şi frontiera p. Procedăm prin inducţie asupra numărului de noduri interne ni. Dacă $ni = 1$ atunci arborele de derivare va avea forma din figura 3.19 şi unicitatea acestuia este evidentă. Presupunem acum că proprietatea este adevărată pentru un ni oarecare şi considerăm cazul $ni + 1$. Să presupunem că ar exista doi arbori diferiţi cu rădăcina x_0 şi frontiera p; fie aceştia $\mathcal{A}'_{x_0,p}$ şi $\mathcal{A}_{x_0,p}$.

Deoarece gramatica este cu precedenţă simplă, rezultă că fraza simplă stângă, care este aceiaşi în cei doi arbori, este situată în aceiaşi poziţie, $p = r f_p s$. Efectuăm reducerea frazei simple f_p (conform regulii aplicate $x \to f_p$) şi vom obţine arborii $\mathcal{A}'_{x_0,q}$ şi $\mathcal{A}_{x_0,q}$ unde $q = rXs$ iar cei doi arbori trebuie să fie diferiţi. Dar numărul de noduri interne este cel mult ni, ceea ce contrazice ipoteza inductivă.

3.2.8 Determinarea relaţiilor de precedenţă pentru gramatici operatoriale

Vom defini mai întâi relaţiile $F_{1,2}$ şi $L_{1,2}$,analoage cu F şi L de la gramaticile cu precedenţă simplă.

Definiţie Fie $x \in V_N$ şi $y \in V_G$. Vom spune că

$$xF_{1,2}y \text{ daca } \exists x \to yu \in \mathcal{P}, \text{ sau } x \to Nyu \in \mathcal{P},\ u \in V_G^*, N \in V_N$$
$$xL_{1,2}y \text{ daca } \exists x \to uy \in \mathcal{P}, \text{ sau } x \to uyN \in \mathcal{P},\ u \in V_G^*, N \in V_N$$

Închiderile tranzitive, respectiv, tranzitive şi reflexive ale acestor relaţii le vom nota cu $F_{1,2}^+$, $L_{1,2}^+$ şi respectiv, $F_{1,2}^*$, $L_{1,2}^*$. Acum, determinarea relaţiilor de precedenţă operatoriale se poate face cu ajutorul următoarei teoreme.

Teorema (*determinarea relaţiilor de precedenţă*). Avem

(1) $x \overset{\circ}{<} y \Leftrightarrow \exists u \to \ldots xv \ldots \in \mathcal{P}, vF_{1,2}^+ y$;
(2) $x \overset{\circ}{=} y \Leftrightarrow \exists u \to \ldots xNy \ldots \in \mathcal{P}$;
(3) $x \overset{\circ}{>} y \Leftrightarrow \exists u \to \ldots vy \ldots \in \mathcal{P}, vL_{1,2}^+ x$;

Schiţă de demonstraţie. Demonstraţia se poate face prin analiza structurilor posibile ale arborilor de derivare Pentru cazul (1), trebuie să existe structura de arbore ca în figura 3.20.

Această teoremă ne permite să determinăm matricea de precedenţă a gramaticii pe baza căreia putem apoi aplica algoritmul de analiză bottom-up. Problema neterminalelor de la capetele frazei simple stângi se poate rezolva analizând părţile drepte ale regulilor de rescriere. Într-adevăr, teorema ne dă numai terminalele care intră în fraza simplă stângă şi neterminalele interioare; cele două eventuale neterminale de la capete vor aparţine sau nu frazei simple stângi depinzând de forma părţii drepte ale regulii respective. Evident, se pot face şi ipoteze de neambiguitate suplimentare pentru gramatică, similare cu condiţia de a nu exista reguli cu părţile drepte identice.

3.2.9 Studiu de caz

Să considerăm cazul gramaticii pentru generarea expresiilor aritmetice simple G_E, cu regulile obişnuite

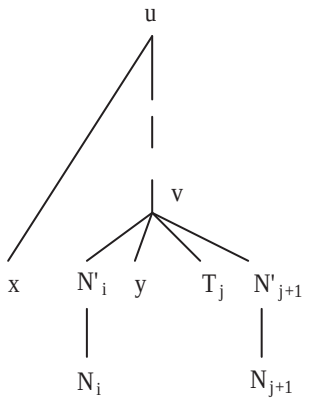

Figura 3.20: Structura posibilă de arbore

$$\begin{cases} E \to E + T | T \\ T \to T * F | F \\ F \to (E) | a \end{cases}$$

Relaţiile $F_{1,2}, L_{1,2}$ şi închiderile acestora sunt:

$E\ F_{1,2}\ \{E, T, +\}$	$E\ F^+_{1,2}\ \{E, T, F, +, *, (, a\}$	$E\ F^*_{1,2}\ \{E, T, F, +, *, (, a\}$
$T\ F_{1,2}\ \{T, F, *\}$	$T\ F^+_{1,2}\ \{T, F, *, (, a\}$	$T\ F^*_{1,2}\ \{T, F, *, (, a\}$
$F\ F_{1,2}\ \{(, a\}$	$F\ F^+_{1,2}\ \{(, a\}$	$F\ F^*_{1,2}\ \{F, (, a\}$
$E\ L_{1,2}\ \{T, +\}$	$E\ L^+_{1,2}\ \{T, +, F, *, a,)\}$	$E\ F^*_{1,2}\ \{E, T, +, F, *, a,)\}$
$T\ L_{1,2}\ \{F, *\}$	$T\ L^+_{1,2}\ \{F, *, a,)\}$	$T\ F^*_{1,2}\ \{T, F, *, a,)\}$
$F\ L_{1,2}\ \{a,)\}$	$F\ L^+_{1,2}\ \{a,)\}$	$F\ F^*_{1,2}\ \{F, a,)\}$

Conform teoremei de calcul a relaţiilor de precedenţă vom obţine matricea de precedenţă (vezi figura 3.21) asociată simbolurilor gramaticii G_E.

Deci gramatica G_E are proprietatea de precedenţă operatorială simplă, astfel că se poate aplica direct algoritmul de analiză sintactică bottom up.

	$+$	$*$	$($	$)$	a
$+$	$\overset{\circ}{>}$	$\overset{\circ}{<}$	$\overset{\circ}{<}$	$\overset{\circ}{>}$	$\overset{\circ}{<}$
$*$	$\overset{\circ}{>}$	$\overset{\circ}{>}$	$\overset{\circ}{<}$	$\overset{\circ}{>}$	$\overset{\circ}{<}$
$($	$\overset{\circ}{<}$	$\overset{\circ}{<}$	$\overset{\circ}{<}$	$\overset{\circ}{=}$	$\overset{\circ}{<}$
$)$	$\overset{\circ}{>}$	$\overset{\circ}{>}$		$\overset{\circ}{>}$	
a	$\overset{\circ}{>}$	$\overset{\circ}{>}$		$\overset{\circ}{>}$	

Figura 3.21: Matricea de precedenţă operatorială pentru G_E

Reconstituirea derivării cuvântului $p = a+(a+a)*a*a$ este prezentată în tabelul 3.22, unde fiecare linie corespunde formei propoziţionale, în care fraza simplă stângă este subliniată, precedată de regula identificată.

3.3 Analiza sintactică LR(k)

Denumirea LR(k) provine de la *Left to right parsing, Rightmost derivation, k–token lookahead* şi reprezintă o categorie de algoritmi de analiză sintactică ascendentă în care se încearcă reconstituirea derivării extrem drepte pentru un şir de intrare prin identificarea la fiecare pas a frazei simple stâangi prin examinarea formei propoziţionale curente şi a maxim k simboluri citite în avans din şirul de intrare rămas de examinat. Formalizarea riguroasă a acestor algoritmi a fost facută de către D. Knuth în 1965, iar o variantă pentru $k = 1$ a devenit populară odată cu sistemul operare Unix prin apariţia utilitarului YACC (Yet Another Compiler Compiler) dezvoltat de AT& T Bell Laboratories în anii 70.

Prezentarea algoritmului urmăreste în mare măsură cea folosită de A. Appel în lucrarea *Modern compiler implementation in Java.*

Analizorul foloseşte o **stivă** şi un **şir de intrare** care prin concatenare reprezintă forma propoziţională curentă din derivarea extrem dreaptă corespunzătoare cuvântului de examinat. În funcţie de primele k simboluri citite în avans din şirul de intrare (*the lookahead*) şi conţinutul

Regula	*Forma propozitionala*
	$\underline{a}^{\circ}> + {}^{\circ}<(a+a)*a*a$
$F \to a$	$F + {}^{\circ}<({}^{\circ}<\underline{a}^{\circ}> + a)*a*a$
$F \to a$	$F + {}^{\circ}<({}^{\circ}<F + {}^{\circ}<\underline{a}^{\circ}>)*a*a$
$F \to a$	$F + {}^{\circ}<({}^{\circ}<\underline{F+F}^{\circ}>)*a*a$
$T \to T*F\ , T \to F$	$F + {}^{\circ}<\underline{(T=}^{\circ})^{\circ}>*a*a$
$F \to (E)\ , E \to T$	$F + {}^{\circ}<F*{}^{\circ}<\underline{a}^{\circ}>*a$
$F \to a$	$F + {}^{\circ}<\underline{F*F}^{\circ}>*a$
$T \to T*F\ , T \to F$	$F + {}^{\circ}<T^{\circ}<*{}^{\circ}<\underline{a}$
$F \to a$	$F + {}^{\circ}<\underline{T*F}$
$T \to T*F$	$\underline{F+T}$
$E \to E*T\ , E \to T\ , T \to F$	E

Figura 3.22: Reconstrucţia derivării cuvântului $p = a + (a + a) * a * a$

stivei analizorul execută una din următoarele acţiuni:

- **Shift:** Mută primul simbol din şirul de intrare în stivă;
- **Reduce:** Alege o regulă a gramaticii, de exemplu $X- > ABC$, extrage de pe stivă simbolurile C, B, A şi pune în vârful stivei neterminalul X. Acţiunea corespunde reducerii frazei simple stângi din algoritmul general boottom up.

Iniţial, stiva este vidă şi cuvântul de examinat constituie şirul de intrare al analizorului. Decizia asupra acţiunii ce trebuie efectuate (shift/reduce) se ia cu ajutorul unui automat finit determinist care examinează conţinutul stivei. Acest automat încearcă recunoaşterea frazei simple stângi (adică partea dreaptă a ultimei reguli aplicate in derivarea extrem dreaptă) atunci când aceasta apare în vârful stivei. Să considerăm gramatica ce generează expresii aritmetice simple ce conţin doar adunarea (3.3) la care s-a adăugat simbolul $ ca sfârşit de şir.

$$\begin{cases} S \to E\$ & (0) \\ E \to T + E & (1) \\ E \to T & (2) \\ T \to F * T & (3) \\ T \to T & (4) \\ F \to (E) & (5) \\ F \to x & (6) \end{cases}$$

Analiza cuvântului $p = (x + x) * x + x\$$ decurge ca în figura 3.23 şi foloseşte automatul finit descris în tabelul 3.24.

Semnificaţia elementelor tabloului este următoarea:

- **s**n Shift şi analizorul intră în starea n.
- **g**n Analizorul trece în starea n. (Goto n).
- **r**k Efectuează reducerea folosind regula cu numărul k. Dacă X este neterminalul nou ce trebuie pus pe stivă atunci se urmează acţiunea indicată de ultima stare la care s-a ajuns înainte de primul simbol al părţii drepte reduse şi neterminalul X.
- **a** Acceptă cuvântul de analizat.

Elementele necompletate din tabel corespund situaţiei de găsire a unei erori sintactice.

În practică se folosesc gramatici ce admit o formă echivalentă de tip LR(1).

3.3.1 Construcţia analizorului pentru gramatici LR(0)

Cazul $k = 0$ corespunde situaţiei când decizia se poate lua prin examinarea stivei, mai exact când în vârful stivei se regăseşte fraza simplă stângă. Deşi există puţine gramatici cu această proprietate, modul de construcţie a tabelei de analiză este uşor de adaptat pentru cazurile uzuale.

Definim noţiunea de **element** sau *item* printr-o regulă a gramaticii ce conţine în partea dreaptă un marcaj suplimentar, de obicei notat

Stiva	*Input*	*Actiune*
1	$(x+x)*x+x\$$	$shift$
$1(2$	$x+x)*x+x\$$	$shift$
$1(2x8$	$+x)*x+x\$$	$reduce\ F \to x$
$1(2F5$	$+x)*x+x\$$	$reduce\ T \to F$
$1(2T10$	$+x)*x+x\$$	$shift$
$1(2T10+17$	$x)*x+x\$$	$shift$
$1(2T10+17x19$	$)*x+x\$$	$reduce\ F \to x$
$1(2T10+17F5$	$)*x+x\$$	$reduce\ T \to F$
$1(2T10+17T10$	$)*x+x\$$	$reduce\ E \to T$
$1(2T10+17E18$	$)*x+x\$$	$reduce\ E \to E+T$
$1(2E3$	$)*x+x\$$	$shift$
$1(2E3)6$	$*x+x\$$	$reduce\ F \to (E)$
$1F4$	$*x+x\$$	$shift$
$1F4*15$	$x+x\$$	$shift$
$1F4*15x8$	$+x\$$	$reduce\ F \to x$
$1F4*15F4$	$+x\$$	$reduce\ T \to F$
$1F4*15T16$	$+x\$$	$reduce\ T \to F*T$
$1T9$	$+x\$$	$shift$
$1T9+11$	$x\$$	$shift$
$1T9+11x8$	$\$$	$reduce\ F \to x$
$1T9+11F4$	$\$$	$reduce\ T \to F$
$1T9+11T9$	$\$$	$reduce\ E \to T$
$1T9+11E12$	$\$$	$reduce\ E \to T+E$
$1E7$	$\$$	$ACCEPT$

Figura 3.23: Analiza sintactică $shift - reduce$ pentru cuvântul $p = (x+x)*x+x\$$.

Stare	x	(	)	+	$*$	\$	E	T	F
1	$s8$	$s2$					$g7$	$g9$	$g4$
2	$s8$	$s2$					$g3$	$g10$	$g5$
3			$s6$						
4				$r4$	$s15$	$r4$			
5			$r4$	$r4$	$s13$				
6				$r6$	$r6$	$r6$			
7						a			
8				$r5$	$r5$	$r5$			
9				$s11$		$r2$			
10			$r2$	$s17$					
11	$s8$	$s2$					$g12$	$g9$	$g4$
12						$r1$			
13	$s8$	$s2$						$g14$	$g5$
14			$r3$	$r3$					
15	$s8$	$s2$						$g16$	$g4$
16				$r3$		$r3$			
17	$s19$	$s20$					$g18$	$g10$	$g5$
18			$r1$						
19			$r5$	$r5$	$r5$				
20	$s19$	$s20$					$g3$	$g10$	$g5$

Figura 3.24: Funcţia de evoluţie a automatului finit determinist pentru analiza shift–reduce.

cu punct "." pentru a indica poziţia curentă a analizorului în şirul de intrare reprezentat de forma propoziţionala curentă din derivarea extrem dreapta. Vom folosi în continuare denumirea de **input** pentru acest şirul ramas de citit din cuvântul de analizat, aflat în forma propoziţională după continutul actual al stivei analizorului . O **stare** a analizorului este o mulţime de elemente (*itemi*).

Să considerăm gramatica din figura 3.3.1.

$$\begin{cases} S' \to S\$ & (0) \\ S \to (L) & (1) \\ S \to x & (2) \\ L \to S & (3) \\ L \to L, S & (4) \end{cases}$$

Iniţial, analizorul are stiva vidă iar input va fi un şir generat pornind de la S urmat de marcajul \$, adică un şir generat de S'. Notăm acest lucru prin itemul $S' \longleftarrow .S\$$, unde punctul indică poziţia curentă a analizorului. În această situaţie, când şirul input începe cu terminale derivate din S, înseamnă că rezultatul derivării va începe cu simboluri obţinute prin aplicarea de reguli ce au S în stânga, adică obţinute din părţile drepte ale regulilor lui S. Vom nota această situaţie prin $\boxed{\begin{array}{l} S' \to .S\$ \\ S \to .x \\ S \to .(L) \end{array}}^{1}$ şi o vom numi *starea 1.*

Acţiuni SHIFT. În starea 1, analizăm efectul introducerii simbolului x în stiva analizorului, adică $shift\ x$ Acest tip de acţiune va fi indicată prin mutarea punctului peste x în regula $S \to x$. Celelalte două elemente ale stării nu sunt relevante pentru această acţiune deoarece punctul nu se află în faţa lui x, astfel că sunt ignorate. Se obţine o nouă stare după shiftare. $\boxed{S \to x.}^{2}$.

Dacă din starea 1 se shiftează o paranteză stânga, în vârful stivei apare paranteza stângă şi poziţia analizorului va fi la începutul unui şir derivat din neterminalul L urmat de o paranteză dreapta. Un astfel se şir începe cu terminale ce se pot deriva din L, ceea ce conduce la introducerea în noua stare a elemetelor provenite din regulile lui L, deci în consecinţă şi elemente ce provin din regulile lui S. Efectul este

o nouă stare, $\boxed{\begin{array}{l} S \to (.L) \\ L \to .L, S \\ L \to .S \\ S \to .x \\ S \to .(L) \end{array}}^{3}$.

Acţiuni Goto. În starea 1 vom analiza efectul obţinut prin găsirea în input a unui şir de terminale derivat din neterminalul S (Acest lucru se poate întâmpla dacă s-a shiftat un x, eventual urmat de şir derivat din L apoi)). Se vor efectua reduceri succesive ale regulilor, pana la neterminalul S, ceea ce echivaleaza cu o secvenţa de extrageri de simboluri din stivă, anume cele ce corespund cu partea dreaptă a regulilor, iar în final S ajunge în vârful stivei iar în input \$. Din starea 1 acţiunea **goto** pentru neterminalul S v-a fi simulată prin mutarea punctului ;in item peste simbolul S, ceea ce conduce la o nouă stare 4: $\boxed{S' \to S.\$}^{4}$.

Acţiuni Reduce. În starea 2 avem punctul la sfârşitul unui element. Asta înseamnă că în vârful stivei se află partea dreaptă a regulii $S \to x$ pregătită pentru reducere, deci efectul va fi extragerea simbolurilor părtii drepte şi depunerea pe stivă a neterminalului S aflat în stânga regulii.

Operaţiile de bază efectuate asupra stărilor analizorului sunt **Closure**(I) şi **Goto**(I, X), unde I este o mulţime de elemente, iar X este un simbol al gramaticii. **Closure** adaugă elemente noi la o mulţime când punctul se află în faţa unui neterminal. iar **Goto** mută punctul peste simbolul X în toate elementele unei mulţimi. Formal, cele două operaţii sunt descrise astel:

Closure(I) =
  repeat
    for orice item $(A \to \alpha.X\beta) \in I$
      for orice regulă $X \to \gamma$
        $I \leftarrow I \bigcup \{X \to .\gamma\}$
  until I nu se modifică.
  return $I\square$

Goto(I, X) =
  $J \longleftarrow \{\}$
  for orice item $A \to \alpha.X\beta$ din I
    adaug $A \to \alpha X.\beta$ la J
  return Closure(J) $\square$

Cu aceste două proceduri simple se poate descrie algoritmul de construcţie a tabelei de analiză sintactică LR(0). Se consideră gramat-

ica augumentată cu regula suplimentară $S' \leftarrow S\$$, mulţimea stărilor T şi mulţimea acţiunilor (arce automatului finit determinist) E.

Iniţializează T cu { **Closure**($\{S' \leftarrow .S\$\}$)}
Iniţializează E cu mulţimea vidă
repeat
    for pentru fiecare stare I din T
        for pentru fiecare item $A \rightarrow \alpha.X\beta$ din I
            let $J \leftarrow$ **Goto**(I, X)
            $T \leftarrow T \bigcup \{J\}$
            $E \leftarrow E \bigcup \{I, XJ\}$
until E şi T nu se modifică în iteraţie □

Pentru simbolul nou introdus \$ nu se calculează **Goto**$(I, \$)$, ci se schimbă starea în *Accept*.

Diagrama de stări a automatului ce defineşte acţiunile analizorului este dată în figura 3.25

iar tabela de analiză este cea din figura 3.26.

Determinarea acţiunilor *reduce* din tabelă s-a făcut folosind algoritmul următor:

$R \longleftarrow \{\}$ *(iniţializează multimea acţiunilor reduce)*
for pentru fiecare stare I din T
  for pentru fiecare item $A \rightarrow \alpha.$ din I
    $R \longleftarrow R \bigcup \{(I, A \rightarrow \alpha)\}$
□

3.3.2 Analizoare SLR

Deoarece foarte puţine limbaje permit analiza LR(0) datorită existenţei mai multor acţiuni pentru un element al tabelei de analiză, se foloseşte o variantă îmbunătăţită numită **simple LR**. Aceasta elimină intrările multiple de tipul *shift/reduce*, ca în cazul gramaticii următoare 3.3.2:

$$\begin{cases} S \rightarrow E\$ & (0) \\ E \rightarrow T + E & (1) \\ E \rightarrow T & (2) \\ T \rightarrow x & (3) \end{cases}$$

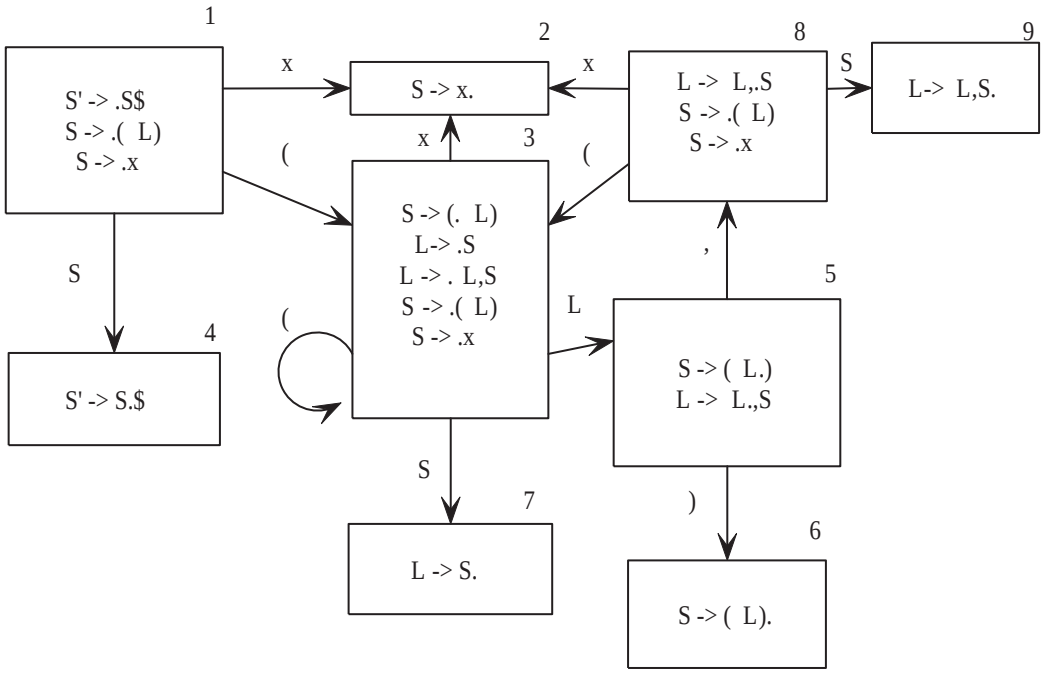

Figura 3.25: Starile automatului finit LR0

Stare	(	)	x	,	$	S	L
1	$s3$		$s2$			$g4$	
2	$r2$	$r2$	$r2$	$r2$	$r2$		
3	$s3$		$s2$			$g7$	$g5$
4					a		
5		$s6$		$s8$			
6	$r1$	$r1$	$r1$	$r1$	$r1$		
7	$r3$	$r3$	$r3$	$r3$	$r3$		
8	$s3$		$s2$			$g9$	
9	$r4$	$r4$	$r4$	$r4$	$r4$		

Figura 3.26: Funcţia de evoluţie a automatului finit determinist pentru analiza shift–reduce.

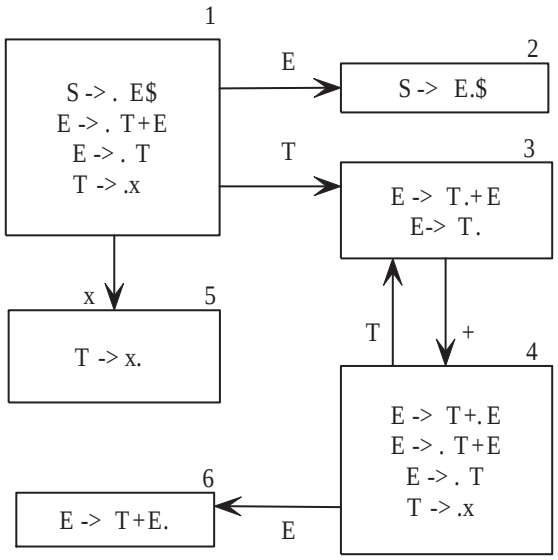

Figura 3.27: Starile analizorului **SLR** pentru gramatica 3.3.2.

Diagrama de stări a automatului finit determinist asociat analizei LR(0) este dată în figura 3.27 iar tabela de analiză din figura 3.28 conţine două acţiuni pentru starea 3 şi simbolul de intrare +. Eliminarea intrării multiple se face prin introducerea de acţiuni *reduce* în tabela de analiză LR(0) doar în coloanele corespunzătoare terminalelor ce fac parte din mulţimea **URM** (vezi algoritmul **??** de la analiza sintactică top down, secţiunea *Un pic de algebră*) .

Algoritmul de calcul al muţimii acţiunilor reduce devine:

$R \longleftarrow \{\}$ *(iniţializează multimea acţiunilor reduce)*
for pentru fiecare stare I din T
  for pentru fiecare item $A \to \alpha.$ din I
    for pentru fiecare terminal $X \in URM(A)$
      $R \longleftarrow R \bigcup \{(I, X, A \to \alpha)\}$
□

iar tabela SLR pentru gramatică considerată va fi cea din figura 3.29.

	x	$+$	\$	E	T
1	$s5$			$g2$	$g3$
2			a		
3		$s4, r2$	$r2$		
4	$s5$			$g6$	$g3$
5		$r3$	$r3$		
6			$r1$		

Figura 3.28: Tabela de analiza LR(0).

	x	$+$	\$	E	T
1	$s5$			$g2$	$g3$
2			a		
3		$s4$	$r2$		
4	$s5$			$g6$	$g3$
5		$r3$	$r3$		
6			$r1$		

Figura 3.29: Tabela de analiza SLR.

Notaţia $(I, X, A \to \alpha)$ indică faptul că din starea I, dacă simbolul X urmează în sirul input (*lookahead symbol*) atunci se face reducerea regulii $A \to \alpha$). Spunem că o gramatică este din clasa **SLR** dacă tabela de analiză SLR nu are cel mult o valoare pentru fiecare element al tabelului.

3.3.3 Gramatici LR(1) şi analizoare LALR(1)

Majoritatea limbajelor de programare pentru care sintaxa este descrisă cu gramatici independente de context admit o formă a gramaticii de tip LR(1), adică se poate construi pentru acestea un analizor sintactic de tip LR(1). Algoritmul de construcţie a tabelei de analiză este similar construcţiei tabelei SLR, dar noţiunea de *element* sau *item* este puţin mai complicată.

Un *item* LR(1)constă dintr-o regulă a gramaticii, o poziţie a analizorului în partea dreaptă a regulii (marcată prin punct) şi un simbol citit în avans din şirul input (*lookahead symbol*). Notaţia uzuală este $(A \to \alpha.\beta, x)$ şi are semnificaţia obişnuită: şirul α este în vârful stivei, iar şirul de intrare poate fi derivat din βx. Desigur, o stare este o mulţime de elemente, iar primitivele de calcul **Closure** şi **Goto** vor incorpora simbolul ce urmează la generare în şirul input. Pseodocodul asociat primitivelor va fi următorul:

Closure(I) =
  repeat
    for oricare $(A \to \alpha.X\beta, z) \in I$
      for orice regula $X \to \gamma$
        for $w \in PRIM(\beta z)$
        $I \leftarrow I \bigcup \{(X \to .\gamma, w)\}$
  until I nu se modifica.
  return I □

Goto(I, X) =
  $J \longleftarrow \{\}$
  for oricare $(A \to \alpha.X\beta, z) \in I$
    $J \leftarrow J \bigcup \{(A \to \alpha X.\beta, z)\}$
  return Closure(J)□

jhsdghjsd Starea iniţială a automatului finit va fi închiderea itemului $(S' \to .S\$, ?)$ unde ? înseamnă că simbolul citit în avans nu contează, deoarece terminatorul \$ nu va shiftat. Acţiunile **reduce** sunt selectate cu următorul algorithm:

$R \longleftarrow \{\}$ *(iniţializează multimea acţiunilor reduce)*
for pentru fiecare stare I din T
  for pentru fiecare item $(A \to \alpha., z)$ din I
    $R \longleftarrow R \bigcup \{(I, z, A \to \alpha)\}$
□

similar cazului gramaticilor SLR.

O gramatică ce nu este din categoria SLR, dar pentru care se poate construi o tabelă de analiză LR(1) are următoarele reguli (3.3.3):

$$\begin{cases} S' \to S\$ & (0) \\ S \to V = E & (1) \\ S \to E & (2) \\ E \to V & (3) \\ V \to x & (4) \\ V \to *E & (5) \end{cases}$$

	x	$*$	$=$	\$	S	E	V
1	$s8$	$s6$			$g2$	$g5$	$g3$
2				a			
3			$s4$	$r3$			
4	$s11$	$s13$				$g9$	$g7$
5				$r2$			
6	$s8$	$s6$				$g10$	$g12$
7				$r3$			
8			$r4$	$r4$			
9				$r1$			
10			$r5$	$r5$			
11				$r4$			
12			$r3$	$r3$			
13	$s11$	$s13$				$g14$	$g7$
14				$r5$			

Figura 3.30: Tabela de analiza LR(1).

Figura **??** conţine diagrama de stări ale automatului finit determinist calculat prin aplicarea primitivelor descrise in acest subcapitol. Atunci când apar mai multi itemi ce diferă doar prin simbolul citit în avans, aceştia au fost descrişi pe scurt prin listarea simbolurilor în dreapta regulii. Este cazul stării 1 din diagramă.

Tabela de analiză LR(1) asociată (vezi figura 3.30) conţine doar intrări simple, ceea ce va însemna că gramatica precedentă are tipul LR(1).

Deoarece tabela de analiză are de obicei dimensiuni mari se foloseşte o versiune prescurtată, numită tabelă **LALR(1)** obţinută prin concatenarea în diagrama de stări a automatului finit LR(1) a acelor stări care diferă doar prin simbolurile citite în avans. Este cazul stărilor 7,12 respectiv 8,11; 6,13; 10,14. Prin renumerotare vom obţine următoarea tabelă (figura 3.31) iar analizorul va fi de tip LALR(1) (*lookahead LR(1)*).

	x	$*$	$=$	\$	S	E	V
1	$s8$	$s6$			$g2$	$g5$	$g3$
2				a			
3			$s4$	$r3$			
4	$s8$	$s6$				$g9$	$g7$
5				$r2$			
6	$s8$	$s6$				$g10$	$g7$
7			$r3$	$r3$			
8			$r4$	$r4$			
9				$r1$			
10			$r5$	$r5$			

Figura 3.31: Tabela de analiza LALR(1).

Cuprins

Bibliografie

1. Andrew W. Appel, *Modern compiler implementation in JAVA*, Cambridge University Press, 2002.

2. Octavian C. Dogaru, *Bazele informaticii. Limbaje formale*, Tipografia Universităţii din Timişoara, 1989.

3. Gheorghe Grigoraş, *Limbaje formale şi tehnici de compilare*, Tipografia Universităţii "Alexandru Ioan Cuza", Iaşi, 1984.

4. J. E. Hopcroft , R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation. Second edition*, Addison Wesley, 2001.

5. Teodor Jucan, Limbaje Formale, Editura Academiei, Bucureşti, 1993.

6. Ştefan Măruşter, *Curs de Limbaje formale şi tehnici de compilare*, Tipografia Universităţii din Timişoara, 1980.

7. Luca Dan Şerbănaţi, *Limbaje de programare şi compilatoare*, Editura Academiei, Bucureşti, 1987.