Limbaje Formale şi Tehnici de Compilare

Drăgan Mircea

April 19, 2006

P R E F A Ţ Ă

Materialul prezentat constituie notele de curs ţinute studenţilor din primii doi ani la Secţia Informatică a Facultăţii de Matematică şi Informatică de la Universitatea de Vest Timişoara.

În primul capitol, *Introducere* se trec în revistă noţiunile fundamentale din teoria limbajelor formale şi se prezintă problema generală a compilării.

Capitolul al doilea, *Limbaje regulate şi analiza lexicală*, prezintă chestiunile teoretice fundamentale privind teoria automatelor şi echivalenţa cu limbajele de tipul trei, proprietăţi speciale şi mecanisme echivalente cu automatele finite. Sunt prezentate de asemenea şi principiile analizei lexicale, finalizate cu realizarea unui analizor lexical.

Capitolul al treilea, *Limbaje independente de context* este dedicat studiului teoretic al mecanismelor de generare şi de recunoaştere a limbajelor de tipul doi. Sunt prezentate formele normale ale gramaticilor independente de context şi câteva proprietăţi speciale.

Capitolul patru, *Analiza sintactică* este dedicat prezentării principiilor generale de analiză sintactică şi a algoritmilor specializaţi de analiză. Pentru fiecare tip de algoritm analizat s-au considerat exemple practice de aplicare.

Capitolul al cincilea, *Sinteza programelor* tratează formele intermediare uzuale pentru traducerea programelor şi generarea codului obiect pornind de la formatul intermediar. Pentru cazul expresiilor aritmetice sunt prezentaţi şi algoritmi direcţi de generarea a formatului intermediar, împreună cu proceduri standard de optimizare a codului generat.

Ultimul capitol, *Maşina Turing* prezintă succint mecanismul formal ce defineşte modelul de calculabilitate.

La sfârşitul fiecărui capitol sunt date câteva exerciţii (nerezolvate), aplicaţii directe la chestiunile teoretice prezentate. Acestea pot fi parcurse în cadrul orelor de seminar şi laborator.

În expunerea algoritmilor s-a folosit un limbaj mai puţin rigid, fără prea multe reguli stricte. În general s-a urmărit descrierea cât mai simplă a structurilor care apar în text.

Cuprins

1 Introducere **5**
- 1.1 Limbaje formale . . . 5
- 1.2 Gramatici generative de tip Chomsky . . . 8
- 1.3 Ierarhia Chomsky . . . 12
- 1.4 Traducerea programelor . . . 17
- 1.5 Probleme propuse . . . 20

2 Limbaje Regulate şi Analiza Lexicală **23**
- 2.1 Automate finite şi limbaje regulate . . . 23
- 2.2 Proprietăţi speciale ale limbajelor regulate . . . 29
- 2.3 Expresii regulate şi sisteme tranziţionale . . . 34
- 2.4 Analiza lexicală . . . 37
- 2.5 Probleme propuse . . . 48

3 Limbaje Independente de Context **51**
- 3.1 Arbori de derivare . . . 51
- 3.2 Ambiguitate în familia $\mathcal{L}_2$. . . . 54
- 3.3 Forme normale pentru gramatici de tipul 2 . . . 56
- 3.4 Lema Bar–Hillel . . . 62
- 3.5 Automate push-down (APD) . . . 65
- 3.6 Automate push–down deterministe . . . 73
- 3.7 Probleme propuse . . . 76

4 Analiza Sintactică **79**
- 4.1 Algoritmi TOP-DOWN . . . 80
  - 4.1.1 Algoritmul general de analiză top-down . . . 80
  - 4.1.2 Analiza top-down fără reveniri . . . 81
  - 4.1.3 Programarea unui analizor sintactic. Studiu de caz . . . 84
- 4.2 Algoritmi BOTTOM-UP . . . 90
  - 4.2.1 Gramatici cu precedenţă simplă . . . 90
  - 4.2.2 Relaţii de precedenţă . . . 92
  - 4.2.3 Proprietăţi ale gramaticilor cu precedenţă simplă . . . 93

4.2.4 Determinarea relaţiilor de precedenţă pentru gramatici cu precedenţă simplă 95
4.2.5 Studiu de caz 95
4.2.6 Gramatici operatoriale 97
4.2.7 Gramatici operatoriale 100
4.2.8 Determinarea relaţiilor de precedenţă pentru gramatici operatoriale 103
4.2.9 Studiu de caz 104

5 Sinteza Programelor **107**
5.1 Forme interne ale programelor 107
5.1.1 Tabelele compilatorului 107
5.1.2 Cvadruple şi triplete 109
5.2 Generarea formatului intermediar 114
5.2.1 Generarea cvadruplelor 115
5.2.2 Generarea tripletelor 117
5.2.3 Generarea şirului polonez 118
5.3 Generarea formatului intermediar pentru instrucţii clasice 121
5.3.1 Instrucţiunea de atribuire 121
5.3.2 Instructiunea If 121
5.3.3 Variabile indexate 121

6 Masina Turing **123**
6.1 Limbaje de tipul zero 123
6.2 Maşina Turing 126

Capitolul 1

Introducere

1.1 Limbaje formale

Noţiunea generală de limbaj. Se numeşte **alfabet** sau vocabular orice mulţime finită şi nevidă. Elementele unui alfabet V le vom numi **simboluri** (sau litere, caractere, variabile).

Definiţie 1.1 *Un* **cuvânt** *peste un alfabet* V *este o secvenţă* $p = a_1 a_2 \dots a_n, a_i \in V, i = 1, \dots, n.$

Numărul n, deci numărul simbolurilor cuvântului p, se numeşte *lungimea cuvântului* şi va fi notat cu $|p|$ sau $l(p)$. Vom considera şi *cuvântul vid* λ sau e, care nu conţine nici un simbol; evident $|\lambda| = 0$.

Noţiunea de *cuvânt* este fundamentală în teoria limbajelor formale sau în alte domenii ale informaticii; termeni sinonimi utilizaţi în literatură sunt *propoziţie, frază* sau *şir*. Să observăm că nu există o similitudine foarte bună între noţiunile de "alfabet", "cuvânt", etc. din teoria limbajelor formale şi noţiunile corespunzătoare din lingvistică.

Mulţimea tuturor cuvintelor peste un alfabet V o notăm cu V^+. Această mulţime împreună cu cuvântul vid va fi notată cu V^*. În general, vom utiliza litere mari de la sfârşitul alfabetului pentru notarea diverselor alfabete, U, V, W, etc.; litere de la începutul alfabetului (mari sau mici) pentru notarea simbolurilor, $A, B, C, \dots, a, b, c, \dots, i, j, \dots$ (uneori cifre 0,1,2,...); pentru notarea cuvintelor vom utiliza litere mici de la sfârşitul alfabetului, $p, q, r, s, t, u, v, w, x, y, z$, etc. (această convenţie de notare nu va fi absolută).

Fie $p = a_1 \dots a_n, q = b_1 \dots b_m$. Definim pe mulţimea V^* operaţia de *concatenare* sau *juxtapunere* prin $pq = a_1 \dots a_n b_1 \dots b_m$.

Se poate verifica uşor că această operaţie este asociativă. Prin urmare, mulţimea V^+ înzestrată cu această operaţie este un semigrup (semigrupul liber peste V). Mulţimea V^* cu aceiaşi operaţie este un semigrup cu unitate, deci un monoid, unitatea fiind cuvântul vid (monoidul liber peste V). Structura are şi

proprietatea de simplificare la stânga şi la dreapta, adică:

$$ua = ub \Rightarrow a = b,\ respectiv,\ au = bu \Rightarrow a = b,\ \forall a, b, u \in V^*$$

Fie din nou $p, q \in V^*$. Vom spune că q este un subcuvât sau un *infix* (propriu) al lui p dacă $p = uqv,\ u, v \in V^*$ $(u, v \in V^+)$; q este *prefix* (propriu) al lui p dacă $p = qv, v \in V^* (v \in V^+)$; q este *sufix* (propriu) al lui p dacă $p = uq, u \in V^* (u \in V^+)$.

Definiţie 1.2 *Un* **limbaj** L **peste alfabetul** V *este o parte a mulţimii tuturor cuvintelor peste* V*, deci* $L \subseteq V^*$.

Să observăm că V^* (sau V^+) este întotdeauna o mulţime infinită (evident numărabilă); în această accepţiune generală, un limbaj poate să fie o mulţime finită sau infinită, uneori chiar vidă.

Exemplu. Fie $V = \{0, 1\}$. Avem

$$V^+ = \{0, 1, 00, 01, 10, 000, \ldots\},$$

$$V^* = \{\lambda, 0, 1, 00, 01, 10, 000, \ldots\}.$$

Limbaje peste alfabetul V sunt de exemplu mulţimile

$$L_1 = \{\lambda, 00, 11\},$$

$$L_2 = \{1, 11, 111, \ldots\} = \{1^n | n \geq 1\}.$$

Observaţie. Notaţia a^n, unde a este un simbol al unui alfabet, înseamnă cuvântul constituit din n simboluri a, adică $\underbrace{aa \ldots a}$. În particular $a^0 = \lambda$.

Operaţii cu limbaje Limbajele fiind mulţimi se pot efectua cu limbaje operaţiile obişnuite cu mulţimi: *reuniune, intersecţie, diferenţă, complementariere (faţă de* V^**).* Există şi operaţii specifice limbajelor.

În general, o operaţie de n-aritate oarecare (cel mai adesea binară sau unară) pe mulţimea V^* defineşte o operaţie corespunzătoare pe mulţimea limbajelor. Astfel, dacă

$$\alpha : V^* \longrightarrow \mathcal{P}(V^*) \text{ şi } \beta : V^* \times V^* \longrightarrow \mathcal{P}(V^*)$$

sunt două operaţii pe V^* (unară şi respectiv binară) şi L_1, L_2 sunt două limbaje peste V, putem defini limbajele $\alpha(L_1)$ respectiv $\beta(L_1, L_2)$ prin

$$\alpha(L_1) = \bigcup_{x \in L_1} \alpha(x),\ \ \beta(L_1, L_2) = \bigcup_{x \in L_1, y \in L_2} \beta(x, y).$$

Exemple:

1. *Produsul* (concatenarea) a două limbaje definit prin
$$L_1L_2 = \{pq|p \in L_1, q \in L_2\}.$$
Dacă $L_1 = L_2 = L$ vom nota $LL = L^2$. Prin recurenţă, se defineşte L^n astfel
$$L^0 = \{\lambda\}, L^k = L^{k-1}L, k \geq 1.$$

2. *Închiderea (Kleene)* a unui limbaj L este
$$L^* = \bigcup_{k=0}^{\infty} L^k.$$

3. Limbajul $Sub(L)$. Fie V^* şi $Sub(x)$ mulţimea tuturor subcuvintelor lui x (evident Sub este o operaţie unară pe V^*). Dacă L este un limbaj peste V, putem defini limbajul
$$Sub(L) = \bigcup_{x \in L} \{Sub(x)\}.$$
adică limbajul constituit din toate subcuvintele tututor cuvintelor lui L. Semnificaţii analoage vor avea şi limbajele $Pref(L)$ şi $Suf(L)$.

4. Limbajul $Mi(L)$. Fie $x = a_1 \dots a_n$ un cuvânt peste alfabetul V. Cuvântul $Mi(x) = a_n \dots a_1$ se numeşte *răsturnatul* sau *oglinditul* lui x (Mi este prescurtarea cuvântului englez *mirror*). Se mai notează $Mi(x) = \tilde{x}$. Avem atunci şi răsturnatul unui limbaj
$$Mi(L) = \bigcup_{x \in L} \{Mi(x)\}.$$

5. Operaţia de *substituţie*. Fie U şi V două alfabete şi fie aplicaţia $s : V \longrightarrow \mathcal{P}(U^*)$. Extindem (prelungim) această aplicaţie la V^* prin
$$s(\lambda) = \{\lambda\}, s(xy) = s(x)s(y), \forall x, y \in V^*.$$
O astfel de prelungire se numeşte canonică; ea păstrează operaţia de concatenare, în sensul că dacă $p = xy$, atunci $s(p) = s(x)s(y)$ este concatenarea limbajelor $s(x), s(y)$. Operaţia de substituţie a limbajelor este dată de
$$s(L) = \bigcup_{x \in L} s(x).$$
Să observăm că această operaţie transformă un limbaj peste un alfabet V într-un limbaj peste un alfabet U şi că păstrează operaţia de concatenare. Dacă $card(a) < \infty, \forall a \in V$, vom spune că substituţia este finită, iar dacă $card(a) = 1, \forall a \in V$ vom spune că s este un homomorfism.

Operaţiile reuniune, produs şi închidere Kleene se mai numesc **operaţii regulate** asupra limbajelor.

1.2 Gramatici generative de tip Chomsky

Un limbaj peste un alfabet poate să fie o mulţime finită sau infinită. Dacă este o mulţime finită, el poate fi definit prin scrierea efectivă a cuvintelor limbajului. În cazul în care este o mulţime infinită, el poate fi definit în anumite cazuri punând în evidenţă structura cuvintelor lui. De exemplu

$$L_2 = \{01, 0011, 000111, \ldots\} = \{0^n 1^n | n \geq 1\}.$$

Există două procedee mai generale pentru definirea limbajelor:

1. Procedee *generative*, care permit generarea tuturor cuvintelor limbajului. Există mai multe tipuri de mecanisme de generare a limbajelor, între care gramaticile Chomsky, sisteme Lindenmayer,etc.

2. Procedee *analitice*, care determina dacă un cuvânt dat aparţine sau nu limbajului. Sunt aşa-numitele *automate, automate finite, automate push-down*, etc.

Un rol deosebit în teoria limbajelor formale îl au gramaticile Chomsky.

Noţiunea de gramatică Fie V_N şi V_T două alfabete disjuncte,$V_N \cap V_T = \emptyset$ numite respectiv *alfabetul simbolurilor neterminale* (V_N) şi *alfabetul simbolurilor terminale* (V_T). Notăm $V_G = V_N \cup V_T$ *alfabetul general* şi $P \subset V_G^* V_N V_G^* \times V_G^*$ *alfabetul regulilor.*

Mulţimea P va fi deci formată din perechi de forma (u, v), unde $u = u'Au''$, $u', u'' \in V_G^*$, $A \in V_N$ iar $v \in V_G^*$, deci u şi v sunt cuvinte peste V_G, cu observaţia că u trebuie să conţină cel puţin un simbol neterminal. Vom spune că o astfel de pereche este o **regulă** (producţie, regulă de generare, regulă de rescriere) şi o vom nota $u \to v$ (vom spune: u se transformă în v). Apartenenţa unei reguli la P o vom nota în mod obişnuit $(u \to v) \in P$, sau mai simplu, $u \to v \in P$ (nu va exista confuzia cu faptul că v este un element al lui P).

Definiţie 1.3 *O gramatică este un sistem $G = (V_N, V_T, X_0, P)$, unde V_N este alfabetul simbolurilor* neterminale*, V_T este alfabetul simbolurilor* terminale*, $X_0 \in V_N$ şi se numeşte* simbol de start *al gramaticii, iar P este mulţimea de* reguli*.*

Observaţie. Simbolurile alfabetului V_N le vom nota în general cu litere mari $A, B, C, \ldots, X, Y, Z$ (mai putin U, V, W) iar cele ale alfabetului V_T cu litere mici de la început $a, b, c, \ldots$ sau cu cifre $0, 1, 2, \ldots$.

Fie G o gramatică şi $p, q \in V_G^*$. Vom spune că p se **derivează direct** în q şi vom scrie $p \underset{G}{\Rightarrow} q$ (sau mai simplu $p \Rightarrow q$) dacă există cuvintele $r, s, u, v \in V_G^*$ astfel încît $p = rus, q = rvs$ iar $u \to v \in P$. Vom spune că p' se **derivează** în p'' (fără specificaţia direct) dacă există $p_1, p_2, \ldots, p_n$ $n \geq 1$ astfel încât

$$p' = p_1 \underset{G}{\Rightarrow} p_2 \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} p_n = p''.$$

Vom scrie $p' \underset{G}{\overset{+}{\Rightarrow}} p''$ (sau $p' \overset{+}{\Rightarrow} p''$ când nu există nici o confuzie) dacă $n > 1$ şi $p' \underset{G}{\overset{*}{\Rightarrow}} p''$ (sau $p' \overset{*}{\Rightarrow} p''$) dacă $n \geq 1$. Şirul de mai sus va fi numit **derivare** iar numărul de derivări directe din şir îl vom numi *lungimea derivării*; se mai spune că p' *derivă* în p''.

Să observăm că transformările astfel definite $\Rightarrow$, $\overset{+}{\Rightarrow}$, $\overset{*}{\Rightarrow}$ sunt relaţii pe V_G^*. Este clar că relaţia $\overset{+}{\Rightarrow}$ este închiderea tranzitivă a relaţiei $\Rightarrow$, iar relaţia $\overset{*}{\Rightarrow}$ este închiderea tranzitivă şi reflexivă a relaţiei de transformare directă .

Definiţie 1.4 . *Limbajul generat de gramatica G este prin definiţie mulţimea*

$$L(G) = \{p \in V_T^*, X_0 \underset{G}{\overset{*}{\Rightarrow}} p\}.$$

Observaţie. Dacă $p \in V_G^*$ şi $X_0 \underset{G}{\overset{*}{\Rightarrow}} p$ se spune că p este o *formă propoziţională* în gramatica G.

Exemple:

1. Fie $G = (V_N, V_T, X_0, P)$, unde $V_N = \{A\}$, $V_T = \{0, 1\}$, $X_0 = A$ (evident) şi $P = \{A \to 0A1, A \to 01\}$. O derivare în această gramatică este, de exemplu
$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000111 = 0^3 1^3.$$
Este evident că $L(G) = \{0^n 1^n | n \geq 1\}$.
Observaţie. În cazul în care mai multe reguli au aceeaşi parte stângă, le vom scrie compact astfel $u \to v_1|v_2|\ldots|v_n$, simbolul | având sensul de sau; în cazul nostru, $A \to 0A1|01$.

2. $G = (V_N, V_T, X_0, P)$, unde
$V_N = \{$<propoziţie>, <subiect>, <atribut>, <predicat>, <complement>, <substantiv>, <adjectiv>, <verb>, <articol>$\}$,
$V_T = \{$o, orice, matrice, funcţie, derivabilă, continuă, este $\}$,
$X_0 =$<propoziţie>,
$P = \{$<propoziţie>→<subiect><atribut><predicat><complement>,
<subiect>→<articol><substantiv>,
<atribut>→<adjectiv>,
<predicat>→<verb>,
<complement>→<adjectiv>,
<articol>→o|orice,
<substantiv>→matrice|funcţie,
<adjectiv>→derivabilă|continuă,
<verb>→este.
Observaţie. În acest exemplu, "<propoziţie>", "<subiect>", etc., reprezintă fiecare câte un simbol neterminal; de asemenea, "o", "orice", "matrice",

etc., reprezintă simboluri terminale. Se poate uşor observa că această gramatică generează propoziţii simple de forma subiect-atribut-predicat-complement care exprimă judecăţi asupra conceptelor de "matrice" şi "funcţie". De exemplu, se poate forma propoziţia: "orice funcţie derivabilă este continuă", care este din punct de vedere semantic corectă, precum şi propoziţia "orice funcţie continuă este derivabilă", care, după cum se ştie, este falsă. Evident, se pot genera şi numeroase propoziţii care nu au sens. Ceea ce ne interesează în acest moment este aspectul formal, deci din punct de vedere sintactic toate aceste propoziţii sunt corecte; semantic, unele propoziţii pot să fie incorecte sau chiar să nu aibe sens.

Să mai observăm că o gramatică Chomsky este în măsură să constituie un model matematic pentru sintaxa unei limbi, fără să intereseze aspectele semantice. Este ceea ce a încercat să facă Naom Chomsky pentru limba engleză în lucrările sale din anii ′50.

3. $G = (V_N, V_T, X_0, P)$, unde
V_N = {<program>, <instrucţie>, <atribuire>, <if>, <expresie>, <termen>, <factor>, <variabilă>, <index>},
V_T = {begin, end, if, then, stop, t, i, +, *, (,), =, ,, ; },
X_0 =<program>
P = {<program>→begin <linie> end
<linie>→<linie>;<instrucţie> | <instrucţie>
<instrucţie>→<atribuire> | <if> | stop
<atribuire>→<variabilă>=<expresie>
<if>→if(<expresie>) then <atribuire>
<expresie>→<expresie> + <termen> | <termen>
<termen>→<termen> ∗ <factor> | <factor>
<factor>→(<expresie>)| <variabilă>
<variabilă>→t(<index>)|i
<index>→<index>, <expresie> | <expresie>
Gramatica din acest exemplu defineşte un limbaj de programare simplu cu trei tipuri de instrucţii: atribuiri, if-then, stop. Expresiile aritmetice au numai operatorii + şi *; variabilele pot fi simple sau indexate (tablouri), iar i ţine loc de identificator sau constantă. Menţionăm că definirea în acest mod a unui limbaj, inclusiv utilizarea croşetelor pentru desemnarea simbolurilor neterminale, poartă adesea denumirea de **notaţie Backus Naur**; în acest mod s-a definit limbajul ALGOL60.

Tipuri de gramatici După forma regulilor de generare, gramaticile Chomsky se împart în mai multe tipuri; clasificarea obişnuită este următoarea:

- *Gramatici de tipul 0*; sunt gramatici fără restricţii asupra regulilor;
- *Gramatici de tipul 1 (dependente de context)*; sunt gramatici care au reguli de forma

$$uAv \to upv,\ u,p,v \in V_G^*,\ p \neq \lambda,\ A \in V_N$$

sau $A \to \lambda$ şi în acest caz A nu apare în dreapta vreunei reguli.
Observaţie. Evident, regulile de forma a doua au sens numai dacă A este simbolul de start.

- *Gramaticile de tipul 2 (independente de context)*; sunt gramatici care au reguli de forma
$$A \to p,\ A \in V_N,\ p \in V_G^*.$$

- *Gramaticile de tipul 3 (regulate)*; sunt gramatici care au reguli de forma
$$\left\{ \begin{array}{l} A \to Bp \\ C \to q \end{array} \right. \ sau \left\{ \begin{array}{l} A \to pB \\ C \to q \end{array} \right.$$
cu $A, B, C \in V_N$ şi $p, q \in V_T^*$.

Vom nota cu $\mathcal{L}_j,\ j = 0,1,2,3$ familiile de limbaje generate de gramaticile de tipurile $j = 0,1,2,3$; vom avea astfel limbaje de tipul 0, limbaje de tipul 1 (sau *dependente de context*) , limbaje de tipul 2 (sau *independente de context*) şi limbaje de tipul 3 (sau *regulate*). Să observăm că este importantă structura cuvintelor unui limbaj şi nu modul în care sunt notate simbolurile terminale. De exemplu , limbajele

$$L_2' = \{0^n 1^n | n \geq 1\},\ L_2'' = \{a^n b^n | n \geq 1\}$$

sunt în mod practic identice. Putem utiliza o unică notaţie pentru afabetul simbolurilor terminale , de exemplu , $V_T = \{i_1, \ldots, i_n\}$. Clasificarea de mai sus este fundamentală în teoria limbajelor formale, ea a fost introdusă de Naom Chomsky in 1958 şi prin tradiţie noile clase de limbaje sunt raportate la această clasificare. O altă clasificare este următoarea

- *Gramatici de tipul 0*; fără restricţii;

- *Gramatici monotone (de tipul 1)*:
$$u \to v,\ |u| \leq |v|,\ u,v \in V_G^*;$$

- *Gramatici dependente de context*:
$$uAv \to upv,\ u,p,v \in V_G^*,\ p \neq \lambda,\ A \in V_N;$$

- *Gramatici independente de context (de tipul 2)*:
$$A \to p,\ A \in V_N,\ p \in V_G^*;$$

- *Gramatici liniare*:

$$A \to uBv,\ A \in V_N,\ B \in V_N \cup \{\lambda\}\ u, v \in V_T^*;$$

- *Gramatici (stâng) drept liniare*:

$$A \to uB\ (A \to Bv),\ A \in V_N, B \in V_N \cup \{\lambda\}\ u, v \in V_T^*;$$

- *Gramatici regulate (de tipul 3)*; gramatici stâng liniare sau gramatici drept liniare.

Gramaticile monotone ca şi cele dependente de context nu pot avea reguli cu partea dreaptă vidă. Se introduce următoarea convenţie de completare: *într-o gramatică monotonă sau dependentă de context se admite o regula de forma $A \to \lambda$ cu condiţia ca A să nu apară în partea dreaptă a vreunei reguli.* După cum vom vedea , existenţa sau inexistenţa regulilor de forma $A \to \lambda$, reguli numite de ştergere, poate modifica esenţial puterea generativă a unei gramatici. O gramatică în care nu avem astfel de reguli o vom numi *gramatică $\lambda-$liberă*; de asemenea, un limbaj care nu conţine cuvântul vid, îl vom numi *limbaj $\lambda-$liber*. Să mai observăm că existenţa regulilor de ştergere într-o gramatică nu implică în mod necesar existenţa cuvântului vid în limbajul generat.

Două gramatici care generează acelaşi limbaj se numesc **echivalente**.

Gramaticile monotone şi gramaticile dependente de context sunt echivalente; de asemenea, gramaticile drept şi stâng liniare sunt echivalente, justificându-se astfel clasa gramaticilor regulate.

1.3 Ierarhia Chomsky

Lemele care urmează vor avea o utilizare frecventă în cele ce urmează.

Lema 1.1 *(Lema de localizare a gramaticilor independente de context) Fie G o gramatică independentă de context şi fie derivarea*

$$x_1 \dots x_m \stackrel{*}{\Rightarrow} p,\ \textit{unde}\ x_j \in V_G, j = \overline{1,m}\ , p \in V_G^*.$$

Atunci există $p_1, p_2, \dots, p_m \in V_G^*$ astfel încât

$$p = p_1 \dots p_m\ \ \textit{si}\ \ x_j \stackrel{*}{\Rightarrow} p_j,\ j = \overline{1,m}.$$

Demonstraţie. Procedăm prin inducţie asupra lungimii derivării l.

Dacă $l = 0$ atunci $p = x_1 \dots x_m$ şi luăm $p_j = x_j$.

Presupunem că proprietatea este adevărată pentru derivări de lungime l şi fie o derivare de lungime $l+1, x_1 \dots x_m \stackrel{*}{\Rightarrow} p$. Punem în evidenţă ultima derivare

directă $x_1 \dots x_m \overset{*}{\Rightarrow} q \Rightarrow p$. Conform ipotezei inductive, $q = q_1 \dots q_m$ şi $x_j \overset{*}{\Rightarrow} q_j,\ j = 1, \dots, m$.

Fie apoi $A \to u$ regula care se aplică în derivarea directă $q \Rightarrow p$ şi să presupunem că A intră în subcuvântul q_k, deci $q_k = q'_k A q''_k$. Vom lua

$$p_j = \begin{cases} q_j & , j \neq k \\ q'_k u q''_k & , j = k \end{cases}$$

Este evident că $x_j \overset{*}{\Rightarrow} p_j,\ j \neq k$, iar pentru $j = k$ avem

$$x_k \overset{*}{\Rightarrow} q_k = q'_k A q''_k \Rightarrow q'_k u q''_k = p_k . \square$$

Vom pune în evidenţă în continuare o proprietate asupra structurii regulilor gramaticilor Chomsky. Partea dreaptă a unei reguli, pentru toate tipurile de gramatici, este un cuvânt format din terminale sau neterminale. Este convenabil de multe ori ca partea dreaptă a regulilor să conţină un singur tip de simboluri, terminale sau neterminale. Acest lucru este posibil fără modificarea tipului gramaticii.

Lema 1.2 *(Lema $A \to i$) Fie $G = (V_N, V_T, S, P)$ o gramatică de tipul 2. Există o gramatică G' echivalentă, de acelaşi tip, cu proprietatea că dacă o regulă are în partea dreaptă un terminal, atunci ea este de forma $A \to i, A \in V_N, i \in V_T$.*

Demonstraţie. Luăm gramatica G' de forma $G' = ({V_N}', {V_T}', S, P')$ unde ${V_N}'$ şi P' se construiesc astfel: $V_N \subseteq {V_N}'$ şi includem în P' toate regulile din P care convin. Fie acum o regulă din P care nu convine (punem în evidenţă apariţia terminalelor din partea dreaptă):

$$u \to v_1 i_1 v_2 i_2 \dots i_n v_{n+1},\ i_k \in V_T,\ v_k \in V_N^*.$$

Vom introduce în P' următoarele reguli:

$$u \to v_1 X_{i_1} v_2 X_{i_2} \dots X_{i_n} v_{n+1},\ \ X_{i_k} \to i_k,\ k = \overline{1, n}\ ,$$

unde X_{i_k} sunt neterminale noi pe care le adăugăm la ${V_N}'$. Este evident că G' păstrează tipul lui G şi că $L(G') = L(G) \square$.

Observaţie. Construcţia din lemă se poate extinde uşor la gramatici de tipul 0, înlocuind şi terminalele din partea stângă a regulilor (u poate fi un şir arbitrar ce conţine minim un neterminal). Gramatica astfel obţinută este echivalentă cu cea iniţială şi păstrează tipul.

Ierarhia Chomsky. Este evident că $\mathcal{L}_3 \subset \mathcal{L}_2$ şi că $\mathcal{L}_1 \subset \mathcal{L}_0$, deoarece orice regulă a unei gramatici de tipul 3 respectă prescripţiile unei gramatici de tipul 2; analog pentru familiile $\mathcal{L}_1$ şi $\mathcal{L}_0$. Aparent, o regulă de forma $A \to p$ (de tipul 2) este un caz particular a unei reguli de forma $uAv \to upv$ (de tipul 1), pentru

$u = v = \lambda$; totuşi, realitatea nu este aceasta, deoarece la tipul 2 de gramatici sunt permise reguli de ştergere, pe când la tipul 1 se impune condiţia $p \neq \lambda$. Vom arăta că avem $\mathcal{L}_2 \subset \mathcal{L}_1$.

Şirul de incluziuni

$$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$$

poartă denumirea de **ierarhia Chomsky** (vom arăta pe parcursul acestui curs că incluziunile sunt stricte). Această ierarhie caracterizează puterea generativă a celor patru tipuri de gramatici, această putere fiind crescătoare de la 3 la 0. Orice alte mecanisme generative se raportează la această ierarhie fundamentală. Vom demonstra mai întâi următoarea lemă.

Lema 1.3 *(*Lema eliminării regulilor de ştergere*). Orice limbaj independent de context λ-liber poate fi generat de o gramatică de tipul 2 fără reguli de ştergere.*

Demonstraţie. Fie $G = (V_N, V_T, X_0, P)$ o gramatică independentă de context şi $L(G)$ limbajul generat. Prin ipoteză $\lambda \notin L(G)$. Definim prin recurenţă şirul de mulţimi U_k astfel:

$$U_1 = \{X | X \in V_N, X \to \lambda \in P\}$$

$$U_{k+1} = \{X | X \in V_N, X \to p \in P, p \in U_k^*\} \cup U_k,\ k \geq 1.$$

Să observăm că şirul de mulţimi definite este crescător în raport cu relaţia de incluziune. Cum toate aceste mulţimi sunt incluse în V_Nşi $V_N$este finită, rezultă că există o mulţime maximală U_f astfel astfel încât

$$U_1 \subseteq U_2 \subseteq \cdots \subseteq U_f = U_{f+1} = \cdots.$$

Are loc de asemenea şi implicaţia $X \in U_f \Leftrightarrow X \stackrel{*}{\Rightarrow} \lambda$.

Vom ilustra această implicaţie cu un exemplu. Să presupunem că $U_f = U_3$ şi fie $X \in U_3$. Atunci în mod necesar trebuie să avem $X \to p \in P$, $p \in U_2^*$; de exemplu $p = X_1X_2$ şi $X_1, X_2 \in U_2$. În mod analog $X_1 \to Y_1Y_2Y_3$, $X_2 \to Z_1Z_2$ şi $Y_1, Y_2, Y_3, Z_1, Z_2 \in U_1$, prin urmare $Y_1 \to \lambda$, $Y_2 \to \lambda$, $Y_3 \to \lambda$, $Z_1 \to \lambda$, $Z_2 \to \lambda$. Putem scrie derivarea

$$X \Rightarrow X_1X_2 \stackrel{*}{\Rightarrow} Y_1Y_2Y_3Z_1Z_2 \stackrel{*}{\Rightarrow} \lambda.$$

Definim acum următoarea gramatică independentă de context fără reguli de ştergere $G' = (V_N, V_T, X_0, P')$ unde V_N, V_T, X_0 sunt ca în gramatica dată, iar P' se construieşte pornind de la P astfel. Fie $X \to p \in P, p \neq \lambda$. Includem atunci în P' această regulă precum şi toate regulile de forma $X \to p_j$, unde p_j se obţine din p lăsând la o parte, în toate modurile posibile, simbolurile din U_f (se exceptă cazul $p_j = \lambda$). De exemplu dacă $X \to ABC \in P$ şi $A, B \in U_f$, vom induce în P' regulile

$$X \to ABC,\ X \to BC,\ X \to AC,\ X \to C.$$

Să observăm că în acest fel mulţimea P a fost pe de o parte micşorată (au fost excluse regulile de alegere), iar pe de altă parte îmbogăţită cu eventualele noi reguli. Să mai obsevăm că G' este independentă de context şi că nu conţine reguli de ştergere.

Vom arăta că $L(G) = L(G')$.

Mai întâi, să arătăm că $L(G) \subseteq L(G')$.

Fie $p \in L(G)$, deci $X_0 \underset{G}{\overset{*}{\Rightarrow}} p$; vom arăta că $X_0 \underset{G'}{\overset{*}{\Rightarrow}} p$. Vom arăta o implicaţie ceva mai generală, $X \underset{G}{\overset{*}{\Rightarrow}} p$ implică $X \underset{G'}{\overset{*}{\Rightarrow}} p$, pentru orice $X \in V_N$ (relaţia cerută se obţine pentru $X = X_0$). Procedăm prin inducţie asupra lungimii derivării l. Dacă $l = 1$ avem

$$X \underset{G}{\Rightarrow} p \; deci \; X \to p \in P. \; Dar \; p \neq \lambda \; deci \; X \to p \in P'$$

adică $X \underset{G'}{\overset{*}{\Rightarrow}} p$.

Presupunem că afirmaţia este adevărată pentru $l = n$ şi luăm o derivare cu lungimea $l = n + 1$. Punem în evidenţă prima derivare directă

$$X \underset{G}{\Rightarrow} X_1 \dots X_m \underset{G}{\overset{*}{\Rightarrow}} p$$

Conform lemei de localizare avem $p = p_1 \dots p_m$ şi $X_j \underset{G}{\overset{*}{\Rightarrow}} p_j, \; j = 1, \dots, m$. Unele din cuvintele p_j pot să fie vide; pentru precizarea ideilor să presupunem că $p_2, p_3, p_5 = \lambda$. Atunci pentru derivările $X_j \underset{G}{\overset{*}{\Rightarrow}} p_j, \; j \neq 2, 3, 5$, care au lungimea de cel mult n, conform ipotezei inductive avem $X_j \underset{G'}{\overset{*}{\Rightarrow}} p_j$.

Pe de altă parte , pentru $j = 2, 3, 5$ avem $X_2 \underset{G}{\overset{*}{\Rightarrow}} \lambda, \; X_3 \underset{G}{\overset{*}{\Rightarrow}} \lambda, \; X_5 \underset{G}{\overset{*}{\Rightarrow}} \lambda$, deci $X_2, X_3, X_5 \in U_f$. Rezultă că

$$X \to X_1 X_4 X_6 \dots X_m \in P'$$

aşa încât putem scrie

$$X \underset{G'}{\Rightarrow} X_1 X_4 X_6 \dots X_m \underset{G'}{\overset{*}{\Rightarrow}} p_1 p_4 p_6 \dots p_m = p.$$

Deci $X \underset{G'}{\overset{*}{\Rightarrow}} p$ şi luând $X = X_0$ obţinem $p \in L(G')$. Prin urmare $L(G) \subseteq L(G')$.

Să arătăm acum incluziunea inversă , $L(G') \subseteq L(G)$.

Fie $p \in L(G), \; X_0 \underset{G'}{\overset{*}{\Rightarrow}} p$. Punem în evidenţă o derivare directă oarecare

$$X_0 \underset{G'}{\overset{*}{\Rightarrow}} u \underset{G'}{\Rightarrow} v \underset{G'}{\overset{*}{\Rightarrow}} p.$$

Dacă în derivarea directă $u \underset{G'}{\Rightarrow} v$ se aplică o regulă care există şi în G, atunci evident pasul respectiv poate fi făcut şi în G. Să presupunem că se aplică o regulă nou introdusă de exemplu $X \to BC$, deci pasul respectiv va avea forma

$$u = u'Xu'' \underset{G'}{\Rightarrow} u'BCu'' = v$$

Regula $X \to BC \in P'$ a provenit dintr-o regulă din P, lăsând la o parte simboluri din U_f, în cazul nostru din $X \to ABC$, lăsându-l la o parte pe $A \in U_f$. Deoarece $A \overset{*}{\underset{G}{\Rightarrow}} \lambda$, avem

$$u = u'Xu'' \underset{G}{\Rightarrow} u'ABCu'' \overset{*}{\underset{G}{\Rightarrow}} u'ABCu'' = v.$$

Prin urmare orice pas al derivării considerate se poate obţine şi în gramatica G, deci $X_0 \overset{*}{\underset{G}{\Rightarrow}} p$ şi $p \in L(G)$, adică $L(G') \subseteq L(G)\square$.

Teorema 1.1 $\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Demonstraţie. Fie $L \in \mathcal{L}_2$ un limbaj independent de context şi G o gramatica de tipul 2 care îl generează, adică $L = L(G)$.

Presupunem că $\lambda \in L$. Construim gramatica G' ca în lema precedentă; orice cuvânt $p \neq \lambda$ din $L(G)$ se poate obţine în G' şi invers , deci $L(G') = L(G) - \{\lambda\}$. Considerăm atunci o gramatică $G'' = (V_N \cup \{X_0''\}, V_T, X_0'', P' \cup \{X_0'' \to \lambda, X_0'' \to X_0\})$. Evident $L(G'') = L(G)$. Toate regulile lui G'' respectă tipul 1 (cu $u = v = \lambda$) şi conţine o singură regulă de ştergere $X_0'' \to \lambda$ iar X_0'' nu apare în partea dreaptă a vreunei reguli. Deci G'' este de tipul 1 şi orice limbaj independent de context este inclus în $\mathcal{L}_1$, adică $\mathcal{L}_2 \subseteq \mathcal{L}_1$.$\square$

Fiind dată o operaţie binară notată cu "$\bullet$" pe o familie de limbaje L, vom spuna că familia L este închisă la operaţia "$\bullet$" dacă $L_1, L_2 \in L$ implică $L_1 \bullet L_2 \in L$. Definiţia noţiunii de închidere pentru operaţii unare sau cu aritate oarecare este analoagă. Relativ la proprietăţile de închidere a familiilor din clasificarea Chomsky menţionăm următorul rezultat.

Teorema 1.2 *Familiile $\mathcal{L}_j, j = 0, 1, 2, 3$ sunt închise la operaţiile regulate.*

Demonstraţie. Fie $G_k = (V_{N_k}, V_{T_k}, S_k, P_k), k = 1, 2$ două gramatici de acelaşi tip $j, j = 0, 1, 2, 3$. Putem presupune că $V_{N_1} \cap V_{N_2} = \emptyset$. Trebuie să arătăm că limbajele $L(G_1) \cup L(G_2)$, $L(G_1)L(G_2)$, $L(G_1)^*$, sunt de acelaşi tip j. În acest scop vom construi gramatici de tipul j care să genereze le genereze. Vom indica fără demonstrţie modul de construcţie al gramaticilor (pentru detalii vezi [**?**]):

Reuniune

$j = 0, 1, 2, 3$:

$$G = V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1 | S_2\}).$$

Produs
$j = 0, 1, 2$:

$$G = V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, P_1 \cup P_2 \cup \{S \to S_1 S_2\}).$$

$j = 3$:

$$G = V_{N_1} \cup V_{N_2} \cup \{S\}, V_{T_1} \cup V_{T_2}, S_1, P_1' \cup P_2),$$

unde P_1' se obţine din P_1 prin înlocuirea regulilor de forma $A \to p$ cu $A \to pS_2$.
Închidere Kleene
$j = 0, 1$:

$$G^* = (V_N \cup \{S^*, X\}, V_T, S^*, P \cup \{S^* \to \lambda | S | XS, Xi \to Si | XSi, \forall i \in V_T\})$$

$j = 2$:

$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup P^{prime} \cup \{S^* \to S^* S | \lambda\})$$

$j = 3$:

$$G^* = (V_N \cup \{S^*\}, V_T, S^*, P \cup P^{prime} \cup \{S^* \to S | \lambda\})$$

1.4 Traducerea programelor

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.

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 precum şi prin independenţa unor astfel de limbaje faţă de sistemul de calcul. Dintre limbajele de acest tip cu o anumită răspândire în momentul de faţă menţionăm limbajele PASCAL, FORTRAN, C, JAVA, etc.

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.

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 depinde de structura instrucţiilor procesorului, de setul de instrucţii, de posibilităţile de adresare, etc. 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*. De obicei termenul de compilare este utilizat numai în cazul limbajelor evoluate, în cazul limbajelor de asamblare fiind utilizat 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

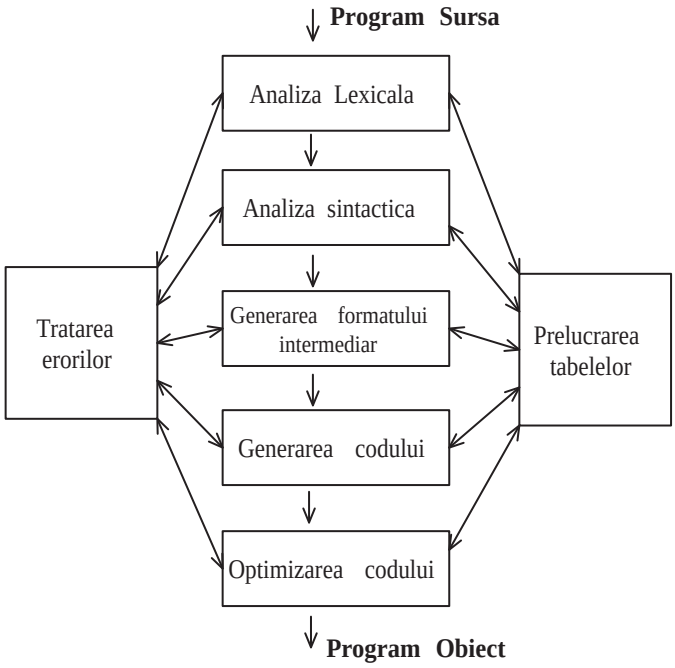

Figura 1.1: Fazele compilării

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).

Analiza lexicală are ca obiectiv principal determinarea unităţilor lexicale ale unui program, furnizarea codurilor acestora şi detectarea eventualelor erori lexicale. Pe lângă aceste operaţii de bază, la analiza lexicală se mai pot efectua anumite operaţii auxiliare precum: eliminarea blank-urilor (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 determină unităţile sintactice ale programului (secvenţe de text pentru care se poate genera format intermediar) 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. Tot la analiza sintactică se definitivează

tabelele de informaţii 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 in care se realizează codul obiect corespunzător programului. Practic în această fază se obţine programul 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.

1.5 Probleme propuse

1. Găsiţi limbajul generat de gramatica $G = (V_N, V_T, S, P)$, precizând tipul gramaticii (cf. clasificării Chomsky):

   (a) $V_N = \{S\}$; $V_T = \{0, 1, 2\}$;
   $P = \{S \to 0S0|1S1|2S2|0\}$.

   (b) $V_N = \{S\}$; $V_T = \{a, b\}$;
   $P = \{S \to aSb|ab\}$.

   (c) $V_N = \{S, A, B\}$; $V_T = \{a, b, c\}$;
   $P = \{S \to abc|aAbc,\ Ab \to bA,\ Ac \to Bbcc,\ bB \to Bb,\ aB \to aaA|aa\}$.

   (d) $V_N = \{S, A, C\}$; $V_T = \{+, -, 0, 1, \ldots, 9\}$;
   $P = \{S \to A| + A| - A,\ A \to AC|C,\ C \to 0|1|\ldots|9\}$.

   (e) $V_N = \{S, A, B\}$; $V_T = \{0, 1\}$;
   $P = \{S \to 0S|1A,\ A \to 0B|1S|0,\ B \to 0A|1B|1\}$.

   (f) $V_N = \{A\}$, $V_T = \{a, b\}$, $S = A$, $P = \{A \to aA|b\}$;

   (g) $V_N = \{x_0\}$, $V_T = \{A, B, \ldots, Z\}$, $S = x_0$, $P = \{x_0 \to x_1D$, $x_1 \to x_2N, x_2 \to E\}$;

(h) $V_N = \{A\}$, $V_T = \{0, 1, 2\}$, $S = A$, $P = \{A \to 0A0|1A1|2A2|\lambda\}$;

(i) $V_N = \{S, A\}$, $V_T = \{0, 1, \ldots, 9, .\}$,
$P = \{S \to A.A, A \to 0A|1A| \ldots |9A|0|1| \ldots |9\}$;

(j) $V_N = \{S\}$, $V_T = \{PCR, PDAR, UDMR\}$,
$P = \{S \to PCR|PDAR|UDMR\}$;

(k) $V_N = \{A, B, C\}$, $V_T = \{0, 1\}$, $S = A$, $P = \{A \to 0A|1B|1$,
$B \to 0C|1A, C \to 0B|1C|0\}$;

(l) $V_N = \{S, A, B, C\}$, $V_T = \{0, 1, \ldots, 9, +, -\}$,
$P = \{S \to +A| - A|A, A \to 0A|1A| \ldots |9A|0|1| \ldots |9\}$;

(m) $V_N = \{S\}$, $V_T = \{(,)\}$, $P = \{S \to S(S)S|\lambda\}$;

(n) $V_N = \{E, T, F\}$, $V_T = \{(,), i, +, *\}$, $S = E$,
$P = \{E \to E + T|T, T \to T * F|F, F \to (E)|i\}$;

(o) $V_N = \{S, A, B\}$, $V_T = \{a, b, c\}$, $P = \{S \to abc|aAbc$,
$Ab \to bA, Ac \to Bbcc, bB \to Bb, aB \to aaA|aa\}$;

(p) $V_N = \{S, A, B, C, D, E\}$, $V_T = \{a\}$, $P = \{S \to ACaB, Ca \to aaC, CB \to DB|E, aD \to Da, AD \to AC, aE \to Ea, AE \to \lambda\}$;

(q) $V_N = \{S, A, B, C, D, E\}$, $V_T = \{a, b\}$, $P = \{S \to ABC$,
$AB \to aAD|bAE, DC \to BaC, EC \to BbC, Da \to aD, Db \to bD, Ea \to aE, Eb \to bE, AB \to \lambda, C \to \lambda, aB \to Ba, bB \to Bb\}$;

2. Precizaţi care dintre gramaticile precedente sunt echivalente.

3. Găsiţi gramatici pentru generarea următoarelor limbaje:

(a) $L = \{\lambda\}$;

(b) $L = \emptyset$;

(c) $L = \{0^n | n \in N\}$;

(d) $L = \{a, ab, aab, ba, baa\}$.

(e) $L = \{a^n c b^n | n \geq 1\}$.

(f) $L = \{w \in \{a, b, c\}^* | w$ conţine cc şi se termină cu $a\}$.

(g) $L = \{BEGIN|END|IF|WHILE|UNTIL\}$.

(h) $L = \{w \in \{0, 1\}^* |$ reprezentarea binară pe 8 biţi a unui întreg $\}$.

(i) $L = \{a^n b^{n+3} a^{n+1} | n \geq 0\}$.

(j) $L = \{w \in \{a, b, c\}^* | w$ începe cu a şi are maxim 4 litere $\}$.

(k) $L = \{a^i b^j c^k | i, j, k > 0\}$.

(l) $L = \{w \in \{0, 1\}^* | w$ multiplu de 8 $\}$.

(m) $L = \{$ constante reale în scrierea obişnuită cu punct zecimal $p_i.p_z\}$.

(n) $L = \{a^i b^j a^i b^j\}$;

(o) $L = \{awbbw' | w, w' \in \{0,1\}^*\}$;

(p) $L = \{w \in \{0,1\}^* | w$ conţine maxim 2 de 0 $\}$;

(q) $L = \{wa\tilde{w} | w \in \{0,1\}^*\}$;

(r) $L = \{w | w$ octet ce reprezintă un număr par $\}$;

(s) $L = \{A, B, C, \ldots, Z\}$;

4. Construiţi o gramatică ce conţine reguli de ştergere, dar generează un limbaj λ-liber.

5. Folosind teorema să se construiască o gramatică independentă de context, fără reguli de ştergere care generează limbajul de la punctul precedent.

Capitolul 2

Limbaje Regulate şi Analiza Lexicală

2.1 Automate finite şi limbaje regulate

Automate finite. Automatele finite sunt mecanisme pentru recunoaşterea limbajelor de tipul 3 (regulate). Un automat finit (AF) se compune dintr-o *bandă de intrare* şi un *dispozitiv de comandă*.

Pe banda de intrare sunt înregistrate simboluri ale unui *alfabet de intrare*, constituind pe bandă un cuvânt p. La fiecare pas de funcţionare banda se deplasează cu o poziţie spre stânga.

Dispozitivul de comandă posedă un *dispozitiv de citire* de pe bandă; dispozitivul se află permanent într-o anumită *stare internă*, element al unei *mulţimi finite de stări*. Schema unui automat finit este redată în figura 2.1.

Automatul finit funcţionează în paşi discreţi. Un pas de funcţionare constă din: dispozitivul de comandă citeşte de pe banda de intrare simbolul aflat în dreptul dispozitivului de citire; în funcţie de starea internă şi de simbolul citit, automatul trece într-o nouă stare şi mută banda cu o poziţie spre stânga. Automatul îşi încetează funcţionarea după ce s-a citit ultimul simbol înregistrat pe bandă; în acest moment el se va afla într-o anumită stare, care, după cum vom vedea, va juca un rol important în recunoaşterea cuvintelor.

Din punct de vedere matematic, un automat finit este un sistem

$$AF = (\Sigma, I, f, s_0, \Sigma_f),$$

unde

Σ este alfabetul (mulţimea) de stări;

I este alfabetul de intrare;

$f : \Sigma \times I \longrightarrow \mathcal{P}(\Sigma)$ este funcţia de evoluţie;

$s_0 \in \Sigma$ este starea iniţială;

$\Sigma_f \subseteq \Sigma$ este mulţimea de stări finale.

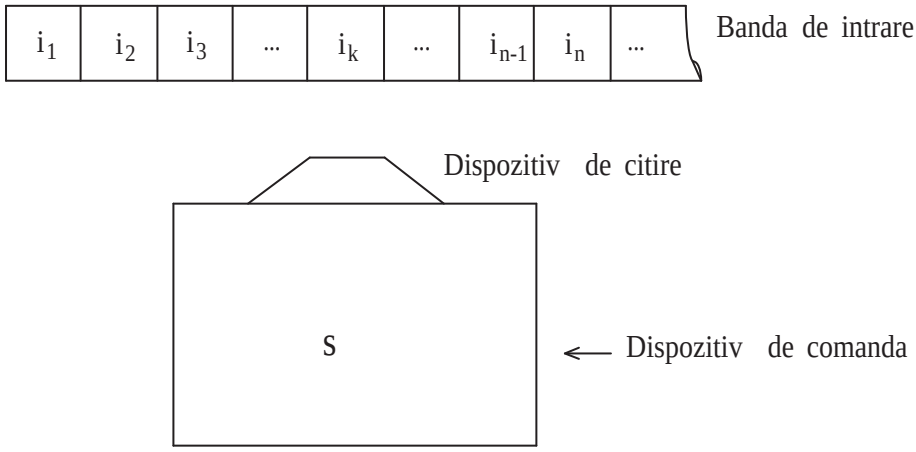

Figura 2.1: Reprezentarea schematică a unui automat finit

Dacă pentru orice $(s, i) \in \Sigma \times I$, avem $|f(s, i)| \leq 1$ automatul se numeşte *determinist*; în caz contrar se numeşte *nedeterminist*.

Observaţii:

1. Funcţionarea unui automat finit se poate bloca în situaţia în care el se află în starea s, citeşte de pe bandă simbolul i şi $f(s, i) = \emptyset$; evident că în acest caz funcţionarea în continuare nu mai este posibilă.

2. În cazul unui automat determinist vom scrie $f(s, i) = s'$ (în loc de $f(s, i) \in \{s'\}$.

3. Definiţia dată este specifică teoriei limbajelor formale. O altă definiţie (mai generală) , întâlnită în teoria automatelor este următoarea: un automat finit este un sistem $AF = (\Sigma, I, O, f, g, s_0, F)$ unde, Σ, I, f, s_0, F au semnificaţia de mai sus, O este *alfabetul de ieşire* iar $g : \Sigma \times I \longrightarrow \mathcal{P}(O)$ este *funcţie de ieşire*. Funcţionalitatea unui astfel de automat finit este analoagă cu cea descrisă mai sus, cu deosebirea că la fiecare pas automatul furnizează o ieşire $o \in g(s, i)$.

Prin *diagrama de stări* a unui automat finit înţelegem un graf orientat care are nodurile etichetate cu stările $s \in \Sigma$ iar arcele se construiesc astfel: nodurile s, s' se unesc cu un arc orientat de la s la s' dacă există $i \in I$ astfel încât $s' \in f(s, i)$; arcul respectiv va fi notat cu i.

Exemplu $AF = (\Sigma, I, f, s_0, \Sigma_f)$ unde $\Sigma = \{s_0, s_1, s_2\}$, $I = \{i_1, i_2\}$, $\Sigma_f = \{s_2\}$, iar f este dată de tabelul

	s_0	s_1	s_2
i_1	$\{s_1\}$	$\{s_2\}$	$\{s_0\}$
i_2	$\emptyset$	$\{s_0, s_1\}$	$\{s_0, s_1\}$

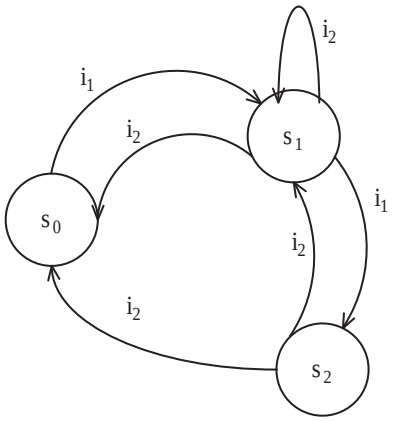

Figura 2.2: Diagrama de stări

Diagrama de stări corespunzătoare este prezentată în figura 2.2.

Funcţia de evoluţie Funcţia f se prelungeşte de la $\Sigma \times I$ la $\mathcal{P}(\Sigma) \times I^*$ deci definim $f' : \mathcal{P}(\Sigma) \times I^* \longrightarrow \mathcal{P}(\Sigma)$, astfel:

(a) $f'(s, \lambda) = \{s\}, \forall s \in \Sigma$,

(b) $f'(\emptyset, i) = \emptyset, \forall i \in I$,

(c) $f'(Z, i) = \bigcup_{s \in Z} f(s, i), \quad Z \in \mathcal{P}(\Sigma), Z \neq \emptyset$,

(d) $f'(Z, pi) = f'(f'(Z, p), i)$.

Prin abuz de limbaj, vom folosi pentru noua funcţie tot notaţia f.Să observăm că relaţiile de mai sus constituie o definiţie prin recurenţă, corectă; fiind dat f, putem defini mai întâi toate valorile $f(Z, i)$ (un număr finit, deoarece I este o mulţime finită), apoi $f(Z, p)$ pentru $|p| = 2$, în continuare $f(Z, p)$ pentru $|p| = 3$, etc.

Proprietăţile funcţiei f:

1. Dacă Z_k este o familie de părţi a lui Σ, avem

$$f(\bigcup_k Z_k, i) = \bigcup_k f(Z_k, i)$$

Demonstraţie. Utilizând (c) putem scrie

$$\bigcup_k f(Z_k, i) = \bigcup_k (\bigcup_{s \in Z_k} f(s, i)) = \bigcup_{s \in \cup Z_k} f(s, i) = f(\bigcup_k Z_k, i). \square$$

2. $f(Z, p) = \bigcup_{s \in Z}(f(s, p)), \ p \in I^*$.

Demonstraţie. Prin inducţie asupra lungimii lui p.

Pentru $|p| = 1$ avem $f(Z, i) = \bigcup_{s \in Z} f(s, i)$, adică (c).

Presupunem că relaţia este adevărată pentru $|p| = m$ şi considerăm un p astfel

încât $|p| = m + 1$, deci $p = p'i, |p'| = m$. Putem scrie

$$f(Z,p) = f(Z,p'i) = f(f(Z,p'),i) = f(\bigcup_{s\in Z} f(s,p'),i) = \\ = \bigcup_{s\in Z} f(f(s,p'),i) = \bigcup_{s\in Z} f(s,p'i) = \bigcup_{s\in Z} f(s,p). \quad \square$$

3. $f(s,pq) = f(f(s,p),q),\ p,q \in I^*$.

Demonstraţie. Inducţie asupra lungimii lui q.

Dacă $|q| = 1$, atunci $q = i$ şi relaţia se reduce la (d).

Presupunem că proprietatea este adevărată pentru r şi considerăm $|q| = r+1$. Deci $q = q'i$. Avem

$$f(s,pq) = f(s,pq'i) = f(f(s,pq'),i) = f(f(f(s,p),q'),i) = \\ = f(f(s,p),q'i) = f(f(s,p),q). \quad \square$$

Limbaje regulate.

Definiţie 2.1 *Limbajul recunoscut de automatul finit* $AF = (\Sigma, I, f, s_0, \Sigma_f)$ *este*

$$L(AF) = \{p | p \in I^*, f(s_0,p) \cap \Sigma_f \neq \emptyset\}$$

Deci $p \in L(AF)$ dacă automatul aflându-se în starea iniţială s_0, după $|p|$ paşi de funcţionare poate să ajungă într-o stare finală.

În cazul unui automat finit determinist limbajul recunoscut poate fi definit în modul următor. Pentru fiecare $s \in \Sigma$ definim funcţia $f_s : I^* \longrightarrow \mathcal{P}(\Sigma)$ prin $f_s(p) = f(s,p)$.

Atunci

$$f(s_0,p) \cap \Sigma_f \neq \emptyset \Leftrightarrow f(s_0,p) = f_{s_0}(p) \in \Sigma_f,$$

deci

$$L(AF) = \{p | f(s_0,p) \cap \Sigma_f \neq \emptyset\} = f_{s_0}^{-1}(\Sigma_f)$$

Limbajele recunoscute de automate finite le vom numi *limbaje regulate*; familia acestor limbaje o vom nota cu $\mathcal{R}$. Evident, familia limbajelor recunoscute de automate finite deterministe, $\mathcal{R}_d$, este o parte a lui $\mathcal{R}$, $\mathcal{R}_d \subseteq \mathcal{R}$. Vom arăta că cele două familii coincid.

Teorema 2.1 $\mathcal{R}_d = \mathcal{R}$.

Demonstraţie. Fie $AF = (\Sigma, I, f, s_0, \Sigma_f)$ un automat finit (în general nedeterminist). Construim următorul automat finit determinist $AF' = (\Sigma', I, f', \{s_0\}, \Sigma'_f)$ unde $\Sigma' = \mathcal{P}(\Sigma)$, $f' = f$ (prelungirea la $\Sigma \times I^*$), $\Sigma'_f = \{Z |\ Z \in \mathcal{P}(\Sigma), Z \cap \Sigma_f \neq \emptyset\}$.

Evident, automatul AF' este determinist.

Fie $p \in L(AF)$. Atunci $f(s_0,p) \cap \Sigma_f \neq \emptyset$ şi $f(s_0,p) \in \Sigma'_f$. Pe de altă parte, conform cu proprietatea 2 a funcţiei de evoluţie, avem

$$f'(\{s_0\},p) = f(s_0,p)$$

şi deci $f'(s_0, p) \in \Sigma'_f$, adică $p \in L(AF')$ şi $L(AF) \subseteq L(AF')$.

Pe o cale analoagă se arată că $L(AF') \subseteq L(AF)$.□

Observaţie. Faptul că un cuvânt este recunoscut de un automat finit se poate verifica prin calcul direct sau pe diagrama de stări.

Exemplu. Considerăm automatul din exemplul anterior (figura 2.2) şi fie $p = i_1 i_2 i_1$. Prin calcul direct:

$$f(s_0, i_1 i_2 i_1) = f(f(f(s_0, i_1), i_2), i_1) = f(f(\{s_1\}, i_2), i_1) =$$
$$= f(\{s_0, s_1\}, i_1) = f(\{s_0\}, i_1) \cup f(\{s_1\}, i_1) = \{s_1\} \cup \{s_2\}.$$

Astfel că $f(s_0, i_1 i_2 i_1) \cap \Sigma_f = \{s_2\} \neq \emptyset$ şi $p \in L(AF)$.

Pe diagrama de stări există traiectoriile:

$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_0 \xrightarrow{i_1} s_1;$$
$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_1 \xrightarrow{i_1} s_2;$$

A doua traiectorie ne duce într-o stare finală, deci $p \in L(AF)$.

Limbaje de tipul trei şi limbaje regulate. Vom arăta în cele ce urmează că familia limbajelor de tipul 3 coincide cu familia limbajelor regulate. În prealabil vom pune în evidenţă o formă specială a limbajelor de tipul 3, pe care convenim să o numim *formă normală.*

Definiţie 2.2 *Vom spune că o gramatică de tipul 3 este în* forma normală *dacă are reguli de generare de forma*

$$\begin{cases} A \to iB, \\ C \to j, \end{cases} \quad \text{unde } A, B, C \in V_N,\ i, j \in V_T$$

sau regula de completare $S \to \lambda$ şi în acest caz S nu apare în dreapta vreunei reguli.

Lema 2.1 *Orice gramatică de tipul 3 admite o formă normală.*

Demonstraţie. Dacă $G = (V_N, V_T, S, P) \in \mathcal{G}_3$ este gramatica dată, eliminăm în primul rând λ-regulile (ca în lema eliminării regulilor de ştergere) apoi construim gramatica $G' = (V'_N, V_T, S, P')$, unde V'_N şi P' se definesc astfel: introducem în V'_N toate simbolurile din V_N iar în P' toate regulile din P care convin; fie acum în P o regulă de forma

$$A \to pB,\ p = i_1 \dots i_n$$

Vom introduce în P' regulile:

$$A \to i_1 Z_1,$$
$$Z_1 \to i_2 Z_2,$$
$$\dots,$$
$$Z_{n-1} \to i_n B,$$

iar simbolurile $Z_1, \dots Z_{n-1}$ le includem in V'_N.

În cazul unei reguli de forma $A \to p$ cu $|p| > 1$ procedăm analog, exceptând ultima regulă nou introdusă care va avea forma $Z_{n-1} \to i_n$. Să mai facem observaţia că simbolurile $Z_1, \dots, Z_{n-1}$ le luăm distincte pentru fiecare caz.

Se poate arăta uşor că $L(G) = L(G')$.□

Teorema 2.2 *Familia limbajelor de tipul 3 coincide cu familia limbajelor regulate.*

Demonstraţie. Partea I: $E \in \mathcal{L}_3 \Rightarrow E \in \mathcal{R}$.

Fie E un limbaj de tipul 3 şi $G = (V_N, V_T, S, P)$ gramatica care îl generează; putem presupune că G este în formă normală. Construim automatul finit $AF = (\Sigma, I, f, s_0, \Sigma_f)$ unde $\Sigma = V_N \cup \{X\}$ (X simbol nou), $I = V_T, s_0 = S$ şi

$$\Sigma_f = \begin{cases} \{X, S\} & ,pentru\ \lambda \in E \\ \{X\} & ,pentru\ \lambda \notin E \end{cases}$$

Funcţia de evoluţie este definită de:

dacă $A \to iB \in P$ luăm $B \in f(A, i)$,
dacă $C \to j \in P$ luăm $X \in f(C, j)$,
în rest $\emptyset$

Observaţie. Automatul astfel definit este în general nedeterminist. De exemplu, dacă $A \to 0B|0$ atunci $f(A, 0) = \{B, X\}$.

Fie $p \in L(G)$, $p = i_1 \dots i_n$, deci $S \overset{*}{\Rightarrow} p$. Detaliat, această derivare va avea forma

$$(1)\ \ S \Rightarrow i_1A_1 \Rightarrow i_1i_2A_2 \Rightarrow i_1i_2 \dots i_{n-1}A_{n-1} \Rightarrow i_1i_2 \dots i_n.$$

S-au aplicat regulile:

$$(2)\quad \begin{array}{l} S \to i_1A_1, \\ A_1 \to i_2A_2, \\ \dots \\ A_{n-1} \to i_n. \end{array}$$

În automat avem corespunzător:

$$(3):\quad \begin{array}{l} A_1 \in f(S, i_1), \\ A_2 \in f(A_1, i_2), \\ \dots \\ X \in f(A_{n-1}, i_n) \end{array}$$

Putem scrie traiectoria

$$(4)\ \ S \xrightarrow{i_1} A_1 \xrightarrow{i_2} A_2 \xrightarrow{i_3} \dots \xrightarrow{i_n} X \in \Sigma_f$$

Deci $p \in L(AF)$.

Dacă $p = \lambda$, atunci $S \Rightarrow \lambda$ şi $S \to \lambda \in P$. Dar atunci $\lambda \in L(AF)$, căci automatul este în starea S şi rămîne în această stare după "citirea" lui λ; cum însă în acest caz $S \in \Sigma_f$ rezultă că şi în acest caz $p \in L(AF)$. În consecinţă $L(G) \subseteq L(AF)$.

Fie acum $p = i_1 \dots i_n \in L(AF)$; atunci avem traiectoria (4), relaţiile (3), regulile de generare (2) şi putem scrie derivarea (1), adică $p \in L(G)$ şi $L(AF) \subseteq L(G)$.□

Partea II $E \in \mathcal{R} \Rightarrow E \in \mathcal{L}_3$.

Vom indica numai modul de construcţie a gramaticii. Fie $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care recunoaşte limbajul E, pe care îl presupunem determinist. Construim gramatica $G = (\Sigma, I, s_0, P)$ unde mulţimea P este definită astfel

$$f(A, i) = B \text{ generează regula } A \to iB \in P,$$
$$\text{în plus dacă } B \in \Sigma_f \text{ se generează şi regula } A \to i \in P.$$

Putem arăta că $L(G) = L(AF)$.□

2.2 Proprietăţi speciale ale limbajelor regulate

Caracterizarea algebrică a limbajelor regulate. Limbajele regulate, fiind părţi din I^*, se pot caracteriza algebric, independent de mecanismele de generare (gramaticile de tipul 3) sau de cele de recunoaştere (automatele finite).

Teorema 2.3 *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.

Demonstraţie: Vom arăta următoarele implicaţii: $(a) \Rightarrow (b), (b) \Rightarrow (c), (c) \Rightarrow (a)$.

$(a) \Rightarrow (b)$.

Fie $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care recunoaşte limbajul E. Definim pe I^* relaţia

$$\xi = \{(p, q) | f(s, p) = f(s, q), \forall s \in \Sigma\}.$$

Se poate vedea cu uşurinţă că ξ este o relaţie de echivalenţă (reflexivă , simetrică , tranzitivă). În plus , dacă $r \in I^*$ şi $(p, q) \in \xi$, atunci $(pr, qr) \in \xi$ şi $(rp, rq) \in \xi$. De exemplu, prima apartenenţă se deduce astfel

$$f(s, pr) = f(f(s, p), r) = f(f(s, q), r) = f(s, qr), \ etc.$$

Prin urmare ξ este o relaţie de congruenţă.

Să arătăm că această congruenţă este de rang finit, adică mulţimea cât I^*/ξ este finită.

Fie $\alpha : \Sigma \longrightarrow \Sigma$ o aplicaţie oarecare şi fie mulţimea

$$I^*(\alpha) = \{p | p \in I^*, f(s,p) = \alpha(s), \forall s \in \Sigma\}.$$

Să observăm că dacă α este funcţia identică atunci $\lambda \in I^*(\alpha)$. Deci nu toate $I^*(\alpha)$ sunt vide; în acest caz $I^*(\alpha)$ este o clasă de echivalenţă. Într-adevăr, fie $p \in I^*(\alpha) \subseteq I^*$ fixat şi fie C_p clasa de echivalenţă a lui p. Arătăm că $C_p = I^*(\alpha)$. Dacă $q \in I^*(\alpha)$, atunci $f(s,q) = \alpha(s), \forall s \in \Sigma$, ceea ce înseamnă că $f(s,q) = f(s,p), \forall s \in \Sigma$, şi deci $(p,q) \in \xi$ adică $q \in C_p$ şi $I^*(\alpha) \subseteq C_p$.

Invers dacă $q \in C_p$ atunci $f(s,q) = f(s,p) = \alpha(s), \forall s \in \Sigma$ şi $q \in I^*(\alpha)$, adică $C_p \subseteq I^*(\alpha)$. Aceasta înseamnă că $I^*(\alpha) = C_p$, adică $I^*(\alpha)$ este o clasă de echivalenţă .

Între mulţimea cât I^*/ξ şi mulţimea funcţiilor definite pe Σ cu valori în Σ putem stabili următoarea corespondenţă biunivocă: *unei funcţii* $\alpha : \Sigma \longrightarrow \Sigma$ *îi corespunde clasa de echivalenţă* $I^*(\alpha)$. Invers, fiind dată o clasă de echivalenţă C, luăm $p \in C$ (oarecare) şi ataşăm lui C funcţia $\alpha(s) = f(s,p) \forall s \in \Sigma$. Ţinând cont că dacă $q \in C$ atunci $f(s,p) = f(s,q), \forall s \in \Sigma$, rezultă că funcţia α nu depinde de elementul p ales.

Dar mulţimea funcţiilor definite pe Σ cu valori în Σ este finită, deci I^*/ξ este finită, adică congruenţa ξ este de rang finit.

Fie acum $p \in L(AF)$ şi q astfel ca $(p,q) \in \xi$. Avem

$$f(s_0, q) = f(s_0, p) \in \Sigma_f;$$

adică $q \in L(AF)$. Aceasta înseamnă că odată cu elementul p, $L(AF)$ conţine clasa de echivalenţă a lui p. De aici rezultă că $L(AF)$ este constituit dintr-un anumit număr de clase de echivalenţă a lui ξ.□

(b)⇒(c)

Fie ξ o congruenţă de rang finit şi E o reuniune de clase de echivalenţă. Fie apoi $(p,q) \in \xi$; aceasta înseamnă că $r_1pr_2 \in E \Leftrightarrow r_1qr_2 \in E$, deci

$$\chi_E(r_1pr_2) = \chi_E(r_1qr_2), \forall r_1, r_2 \in I^*.$$

Prin urmare, $(p,q) \in \mu$. Orice clasă de echivalenţă din I^*/ξ este inclusă într-o clasă de echivalenţă din I^*/μ, aşa încât $card(I^*/\mu) < card(I^*/\xi)$, adică congruenţa μ este de rang finit.□

(c)⇒ (a)

Presupunem că μ este o congruenţă de rang finit; considerăm automatul finit $AF = (I^*/\mu, I, f, C_\lambda, \Sigma_f)$, unde funcţia de evoluţie f şi mulţimea de stări finale sunt

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

Vom arăta că $E = L(AF)$. În primul rând să observăm că $f(C_p, q) = C_{pq}$ (se poate arăta prin inducţie asupra lui $|q|$). Avem

$$p \in E \Leftrightarrow C_p \in \Sigma_f \Leftrightarrow f(C_\lambda, p) = C_{\lambda p} = C_p \in \Sigma_f \Leftrightarrow p \in L(AF)$$

adică $E = L(AF)$ şi E este un limbaj regulat.□

Corolar 2.4 *Familia R este închisă la operaţia de răsturnare.*

Demonstraţie. Fie $E \in \mathcal{R}$ şi $\widetilde{E}$ răsturnatul lui E. Conform teoremei de caracterizare, $card(I^*/\mu^E) < \infty$. Avem

$$(p, q) \in \mu^E \Leftrightarrow \chi_E(r_1 p r_2) = \chi_E(r_1 q r_2) \Leftrightarrow$$

şi

$$\chi_{\widetilde{E}}(\widetilde{r_1}\widetilde{p}\widetilde{r_2}) = \chi_{\widetilde{E}}(\widetilde{r_1}\widetilde{q}\widetilde{r_2}) \Leftrightarrow (\widetilde{p}, \widetilde{q}) \in \mu^{\widetilde{E}}$$

Cu alte cuvinte dacă C este o clasă de echivalenţă a lui μ^E atunci $\widetilde{C}$ (răsturnatul lui C) este o clasă de echivalenţă a lui $\mu^{\widetilde{E}}$. Aceasta înseamnă că $card(I^*/\mu^{\widetilde{E}}) = card(I^*/\mu^E) < \infty$ şi conform aceleiaşi teoreme de caracterizare $\widetilde{E} \in \mathcal{R}$.□

Observaţie. Am văzut că limbajele de tipul 3 pot fi definite de gramatici cu reguli de două categorii: drept liniare sau stâng liniare. Este evident că limbajele stâng liniare sunt răsturnatele limbajelor drept liniare. Cum familia limbajelor regulate (drept liniare) este închisă la operaţia de răsturnare, rezultă că cele două familii de limbaje coincid.

Închiderea familiei $\mathcal{L}_3$ la operaţiile $Pref$ şi complementariere.

Operaţiile $Pref$ şi complementariere se definesc în modul următor

$$Pref(E) = \{p | \exists r \in I^*, pr \in E\}, \;\; C(E) = I^* \setminus E.$$

Teorema 2.5 *Familia $\mathcal{L}_3$ este închisă la operaţiile $Pref$ şi complementariere.*

Demonstraţie. Fie $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care recunoaşte limbajul E. Putem presupune că AF este determinist.

Limbajul $Pref(E)$. Construim automatul finit $AF' = (\Sigma, I, f, s_0, \Sigma'_f)$ unde

$$\Sigma'_f = \{s \in \Sigma | s = f(s_0, q), \; q \in Pref(E)\}.$$

Este evident că $Pref(E) \subseteq L(AF')$, căci dacă $q \in Pref(E)$ atunci $s = f(s_0, q) \in \Sigma'_f$ conform definiţiei lui Σ'_f.

Să arătăm acum că $L(AF') \subseteq Pref(E)$. Fie $r \in L(AF')$, atunci $f(s_0, q) \in \Sigma'_f$, deci există $q \in Pref(E)$ astfel încât $f(s_0, r) = f(s_0, q)$. Cum q este prefixul unui cuvânt din E, există $w \in I^*$ astfel încât $qw \in E$, adică $f(s_0, q) \in \Sigma_f$. Dar

$$f(s_0, rw) = f(f(s_0, r), w) = f(f(s_0, q), w) = f(s_0, qw) \in \Sigma_f,$$

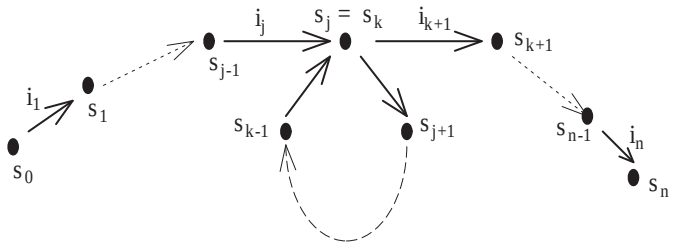

Figura 2.3: Traiectoria automatului finit

deci $rw \in E$ şi $r \in Pref(E)$. Aceasta înseamnă că $L(AF') \subseteq Pref(E)$ şi prin urmare $Pref(E) = L(AF')$, adică $Pref(E)$ este limbaj regulat.□

Limbajul $C(E)$. Avem

$$C(E) = \{p \in I^* | p \notin E\} = \{p \in I^* | f(s_0, p) \notin \Sigma_f\} = \{p \in I^*, f(s_0, p) \in C(\Sigma_f)\}.$$

Prin urmare $C(E) = L(AF_c)$ unde $AF_c = (\Sigma, I, f, s_0, C(\Sigma_f)\}$, adică $C(E)$ este un limbaj regulat.□

Lema de pompare pentru limbaje regulate Sub această denumire (sau lema *uvw*) este cunoscută o proprietate a limbajelor regulate (ca şi a altor familii de limbaje) care ne premite să descompunem cuvintele suficient de lungi ale limbajului în forma *uvw* şi să multiplicăm subcuvântul v de un număr arbitrar de ori, obţinând cuvinte care aparţin de asemenea limbajului. Cu alte cuvinte, putem să "pompăm" în cuvântul dat o anumită parte a sa. Astfel de leme se utilizează deseori pentru a rezolva probleme de neapartenenţă, adică pentru a arăta că un anumit limbaj nu aparţine unei familii date de limbaje.

Lema 2.2 *Fie E un limbaj regulat şi $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care îl recunoaşte. Dacă $p \in E$ şi $|p| \geq card(\Sigma)$ atunci p se descompune în forma $p = uvw,\ v \neq \lambda$ şi $uv^m w \in E,\ \forall m \in N$.*

Demonstraţie. Fie $p = i_1 \dots i_n,\ n \geq card(\Sigma)$; fie $s_0, s_1, \dots, s_n$ stările parcurse de automat la citirea cuvântului p. Atunci, $s_j = f(s_{j-1}, i_j),\ j = \overline{1,n},\ s_n \in \Sigma_f$. Există în mod necesar două stări egale, $s_j = s_k,\ j < k$. Traiectoria va avea o buclă (vezi figura 2.3).

Descompunem cuvântul p în forma $p = uvw$ unde $u = i_1 \dots i_j,\ v = i_{j+1} \dots i_k,\ w = i_{k+1} \dots i_n$. Este clar că $v \neq \lambda$, căci $j < k$. Pe de altă parte, putem parcurge traiectoria făcând de mai multe ori bucla, adică

$$f(s_0, uv^m w) = s_n \in \Sigma_f.$$

Prin urmare $uv^m w \in E$.□

Consecinţă 2.3 *Incluziunea $\mathcal{L}_3 \subseteq \mathcal{L}_2$ este strictă.*

În adevăr, fie limbajul $L_2 = \{0^n 1^n | n \geq 1\}$. Ştim că acest limbaj este de tipul 2 şi că poate fi generat de gramatica $G = (\{A\}, \{0,1\}, A, \{A \to 0A1|01\})$. Să arătăm că L_2 nu este de tipul 3.

Să presupunem că L_2 este de tipul 3 şi fie $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care îl recunoaşte. Cum L_2 conţine cuvinte oricât de lungi, fie $p \in L_2$ astfel încât $p \geq card(\Sigma)$. Conform lemei de pompare, p se descompune în forma $p = uvw$, $v \neq \lambda$ şi $uv^m w \in E$. Putem avea una din situaţiile:

$$(1)\ p = \underbrace{0\ldots0}_{u}\underbrace{0\ldots0}_{v}\underbrace{0\ldots01\ldots1}_{w},$$
$$(2)\ p = \underbrace{0\ldots01\ldots1}_{u}\underbrace{1\ldots1}_{v}\underbrace{1\ldots1}_{w},$$
$$(3)\ p = \underbrace{0\ldots0}_{u}\underbrace{0\ldots1}_{v}\underbrace{1\ldots1}_{w}.$$

Primele două cazuri nu pot avea loc deoarece multiplicându-l pe v, numărul de simboluri 0 şi 1 nu s-ar păstra egal. În al treilea caz, luând de exemplu, $m = 2$ obţinem

$$p_2 = 0\ldots00\ldots10\ldots11\ldots1 \in L_2$$

ceea ce din nou nu este posibil, întrucât se contrazice structura cuvintelor lui L_2. Prin urmare L_2 nu este de tipul 3.□

Observaţie. Este interesant de observat că limbajele simple de forma lui L_2 sunt semnificative pentru clasele din clasificarea Chomsky. Astfel

$$L_1 = \{a^n | n \geq 1\},\ L_1 \in \mathcal{L}_3;$$
$$L_2 = \{a^n b^n | n \geq 1\},\ L_2 \in \mathcal{L}_2,\ L_2 \notin \mathcal{L}_3;$$
$$L_3 = \{a^n b^n c^n | n \geq 1\},\ L_3 \in \mathcal{L}_1(?),\ L_3 \notin \mathcal{L}_2;$$

Ne-am putea aştepta ca limbajul L_3, un exemplu analog lui L_2, să fie de tip 1, adică $L_3 \in \mathcal{L}_1,\ L_3 \notin \mathcal{L}_2$. În adevăr, se poate arăta că $L_3 \notin \mathcal{L}_2$, dar după cunoştinţa autorului, aparenţa $L_3 \in \mathcal{L}_1$ este o problemă deschisă.

Consecinţă 2.4 *Fie E un limbaj regulat şi $AF = (\Sigma, I, f, s_0, \Sigma_f)$ automatul finit care îl recunoaşte. Atunci E este infinit dacă şi numai dacă există $p \in E$ astfel încât $|p| \geq card(\Sigma)$.*

Dacă limbajul este infinit, este clar că există $p \in E$ cu $|p| \geq card(\Sigma)$. Invers, dacă există $p \in E$ cu $|p| \geq card(\Sigma)$ atunci $p = uvw,\ v \neq \lambda$ şi $uv^m w \in E, \forall m \in N$, deci limbajul este infinit.□

2.3 Expresii regulate şi sisteme tranziţionale

Expresii regulate Fie V un alfabet. Expresiile regulate sunt cuvinte peste alfabetul $V \cup \{\bullet, *, |\} \cup \{(,)\}$ la care se adaugă simbolul $\emptyset$, Simbolurile "|"-sau, "$\bullet$"-produs, "$*$"-închidere, le vom considera operatori. Expresiile regulate se definesc astfel:

(1) λ şi $\emptyset$ sunt expresii regulate;

(2) pentru orice $a \in V$, cuvântul a este o expresie regulată;

(3) dacă R şi S sunt expresii regulate, atunci $(R|S)$ $(R \bullet S)$ şi (R^*) sunt expresii regulate;

(4) orice expresie regulată se obţine prin aplicarea de un numar finit de ori a regulilor (1)-(3).

Parantezele sunt utilizate pentru a pune în evidenţă ordinea de aplicare a operatorilor. Pentru simplificarea scrierii vom considera că operatorul $*$ are ponderea cea mai mare, apoi operatorul $\bullet$ şi $|$ ponderea cea mai mică. Astfel parantezele redundante pot fi eliminate. De exemplu, prin $R|S^*$ vom înţelege $(R|(S^*))$.

Observaţie. Expresiile regulate se pot defini cu ajutorul gramaticii $G = (\{E\}, V \cup \{|, \bullet, *, (,)\}, E, P)$ unde

$$P = \{E \to (E|E) \mid (E \bullet E) \mid (E^*) \mid a \mid \lambda \mid \emptyset\}.$$

Unei expresii regulate îi putem asocia un anumit limbaj peste V; vom spune că expresia regulată reprezintă (desemnează, notează) acel limbaj. Modul în care asociem un limbaj unei expresii regulate este

(1) λ reprezintă limbajul $\{\lambda\}$;
(1′) $\emptyset$ reprezintă limbajul $\emptyset$;
(2) a reprezintă limbajul $\{a\}$;
(3) dacă R şi S sunt expresii regulate ce reprezintă limbajele L_R respectiv L_S atunci
  (i) $R|S$ reprezintă limbajul $L_R \cup L_S$;
  (ii) $R \bullet S$ reprezintă limbajul $L_R L_S$;
  (iii) R^* reprezintă limbajul $(L_R)^*$.

Fie R, S, P trei expresii regulate şi L_R, L_S, L_P limbajele reprezentate. Avem :

$$L_{(R|S)|P} = (L_R \cup L_S) \cup L_P = L_R \cup (L_S \cup L_P) = L_{R|(S|P)},$$

întrucât operaţia de reuniune este asociativă. Vom scrie $(R|S)|P = R|(S|P)$. În mod analog se pot obţine şi alte proprietăţi. De exemplu:

$S|R = S|R$,
$R|R = R$,
$(R \bullet S) \bullet P = R \bullet (S \bullet P)$,
$R \bullet (S|P) = (R \bullet S)|(R \bullet P)$, etc.

În cazul în care nu există pericol de confuzie, vom nota cu L (fără indice) limbajul reprezentat de o anumită expresie regulată.

Exemple.

1. $R = a^*$; $L = \bigcup_{j=0}^{\infty}\{a^j\} = \{\lambda, a, a^2, \ldots\}$.
2. $R = aa^*$; $L = a \bullet \{\lambda, a, a^2, \ldots\} = \{a, a^2, a^3 \ldots\}$.
3. $R = (a|b)^*$; $L = (L_a \cup L_b)^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$;
$\{a, b\}^* = \{\lambda\} \cup \{a, b\}^1 \cup \{a, b\}^2 \cup \ldots = \{\lambda, a, b, aa, ab, ba, bb, \ldots\}$, adică
$(a|b)^*$ reprezintă mulţimea tuturor cuvintelor peste alfabetul $\{a, b\}$.
4. $R = a|ba^*$; $L = \{a\} \cup \{b\} \bullet \{\lambda, a, a^2, \ldots\} = \{a, b, ba, ba^2, \ldots\}$.

Limbajele reprezentate de expresii regulate constituie o anumită familie de limbaje; o vom nota cu $\mathcal{L}_{lr}$. Apare următoarea problemă: *care este poziţia acestei familii în ierarhia Chomsky*? Vom arăta că $\mathcal{L}_{lr}$ coincide cu familia limbajelor regulate.

Sisteme tranziţionale

Definiţie 2.3 *Un sistem tranziţional este un sistem de forma*
$ST = (\Sigma, I, f, \Sigma_0, \Sigma_f, \delta)$
unde:

Σ *este o mulţime (finită) de stări;*
I *este alfabetul de intrare;*
$f : \Sigma \times I \longrightarrow \mathcal{P}(\Sigma)$ *este funcţia de tranziţie;*
$\Sigma_0 \subseteq \Sigma$ *este mulţimea de stări iniţiale;*
$\Sigma_f \subseteq \Sigma$ *este mulţimea de stări finale;*
$\delta \subset \Sigma \times \Sigma$ *este relaţia de tranziţie .*

Exemplu. $\Sigma = \{s_0, s_1, s_2\}$, $I = \{0, 1\}$, $\Sigma_0 = \{s_0, s_1\}$, $\Sigma_f = \{s_2\}$ iar funcţia şi relaţia de tranziţie sunt date de:

f	s_0	s_1	s_2
0	$\{s_1\}$	$\{s_2\}$	$\{s_0\}$
1	$\emptyset$	$\{s_0, s_1\}$	$\{s_0, s_2\}$

$\delta = \{(s_0, s_1), (s_2, s_1)\}$.

Ca şi în cazul unui automat finit putem construi diagrama de stări completată cu relaţia δ (arcele punctate). În cazul exemplului nostru diagrama de stări este prezenta în figura 2.4

Observaţie: δ fiind o relaţie, are sens δ^* (închiderea tranzitivă şi reflexivă).

Fie $i \in I \cup \{\lambda\}$; vom spune că sistemul tranziţional evoluează direct din starea s' în starea s'' dacă:

(1) $i = \lambda$ şi $(s', s'') \in \delta^*$. Pe diagrama de stări vom avea o traiectorie punctată de la starea s' la starea s'';

$$s' \longrightarrow O \longrightarrow O \longrightarrow \ldots \longrightarrow O \longrightarrow s''.$$

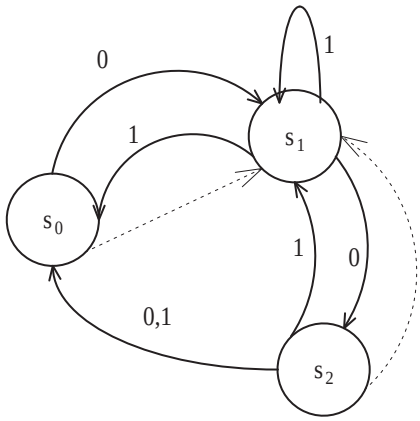

Figura 2.4: Diagrama de stări a sistemului tranziţional

(2) $i \neq \lambda$ şi există $s_1, s_2 \in \Sigma$ astfel încât $(s', s_1) \in \delta^*$, $s_2 \in f(s_1, i)$ şi $(s_2, s'') \in \delta^*$. Pe diagrama de stări, putem ajunge din s' în s'' pe o traiectorie punctată, apoi un pas pe un arc plin şi din nou pe o traiectorie punctată.

$$s' \longrightarrow O \longrightarrow \ldots \longrightarrow s_1 \xrightarrow{i} s_2 \longrightarrow O \longrightarrow \ldots \longrightarrow s''.$$

Vom scrie $s' \overset{i}{\vdash} s''$.

Fie acum $p = i_1 \ldots i_n$. Vom spune că sistemul evoluează din starea s' în starea s'' dacă există $s_0, s_1, \ldots, s_n$ astfel încât

$$s' = s_0 \overset{i_1}{\vdash} s_1 \overset{i_2}{\vdash} \ldots \overset{i_n}{\vdash} s_n = s''.$$

Vom scrie $s' \overset{p}{\vdash} s''$.

Definiţie 2.4 *Limbajul recunoscut de un sistem tranziţional ST este*

$$L(ST) = \{p | p \in I^*,\ \exists s_0 \in \Sigma_0,\ s_0 \overset{p}{\vdash} s,\ s \in \Sigma_f\}.$$

Vom nota cu $\mathcal{L}_{ST}$ familia limbajelor recunoscute de sisteme tranziţionale. Este evident că orice automat finit este un sistem tranziţional particular în care $card(\Sigma_0) = 1$ iar $\delta = \emptyset$ (nu există arce punctate). Prin urmare $\mathcal{R} \subseteq \mathcal{L}_{ST}$.

Teorema 2.6 $\mathcal{R} = \mathcal{L}_{ST}$.

Demonstraţie. Evident, trebuie să arătăm incluziunea $\mathcal{L}_{ST} \subseteq \mathcal{R}$. Fie $ST = (\Sigma, I, f, \Sigma_0, \Sigma_f, \delta)$ un sistem tranziţional. Construim automatul finit $AF = (\mathcal{P}(\Sigma), I, f', \Sigma_0, \Sigma'_f)$ unde

$$f'(Z, i) = \{s | \exists s' \in Z,\ s' \overset{i}{\vdash} s\},$$
$$\Sigma'_f = \{Z | Z \cap \Sigma_f \neq \emptyset\}.$$

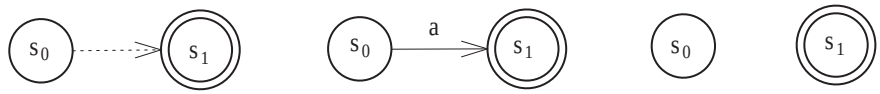

Figura 2.5: Sistemele tranziţionale ce recunosc limbajele λ, a, $\emptyset$.

Fie $p = i_1 \dots i_n \in L(ST)$ şi fie următoarea evoluţie a sistemului tranziţional

$$s_0 \overset{i_1}{\vdash} s_1 \overset{i_2}{\vdash} \dots \overset{i_n}{\vdash} s_n \in \Sigma_f.$$

Putem construi o traiectorie a lui AF de forma

$$\Sigma_0 \xrightarrow{i_1} Z_1 \xrightarrow{i_2} \dots \xrightarrow{i_n} Z_n,$$

unde $Z_1 = f'(\Sigma_0, i_1)$, $Z_k = f'(Z_{k-1}, i_k)$, $k = 2, \dots, n$. Să observăm că $s_0 \in \Sigma_0$ şi că dacă $s_{k-1} \in Z_{k-1}$, atunci conform definiţiei funcţiei f', avem $s_k \in f'(Z_{k-1}, i_k) = Z_k$. Asftel, $s_k \in Z_k, k = 1, \dots, n$; pentru $k = n$ avem $s_n \in Z_n$ şi cum $s_n \in \Sigma_f$ rezultă că $Z_n \cap \Sigma_f \neq \emptyset$, adică $Z_n \in \Sigma'_f$. Deci automatul ajunge într-o stare finală, $p \in L(AF)$ şi $L(ST) \subseteq L(AF)$.

Incluziunea inversă se arată în mod analog.□

Construcţia sistemelor tranziţionale pentru expresii regulate. Fiind dată o expresie regulată putem întotdeauna construi un sistem tranziţional care recunoaşte limbajul reprezentat de expresia respectivă.

Construcţia se face cu ajutorul diagramelor de stări.

Sistemele tranziţionale (diagramele de stări) corespunzătoare expresiilor regulate λ, a şi $\emptyset$ sunt prezentate în figura 2.5.

Dacă R şi S sunt expresii regulate şi notăm cu ST_R şi ST_S sistemele tranziţionale corespunzătoare, atunci sistemele tranziţionale pentru $R|S$, $R \bullet S$ şi R^* sunt redate în figura 2.6.

În acest mod putem construi succesiv (recurent) un sistem tranziţional corespunzător unei expresii regulate.

Exemplu. Sistemul tranziţional corespunzător expresiei $R = a|b \bullet a^*$ este redat în figura 2.7.

Consecinţă Dându-se o expresie regulată, putem construi sistemul tranziţional care recunoaşte limbajul reprezentat de expresia respectivă. Cum orice limbaj recunoscut de un sistem tranziţional este regulat, rezultă că limbajele reprezentate de expresii regulate sunt limbaje regulate.

2.4 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

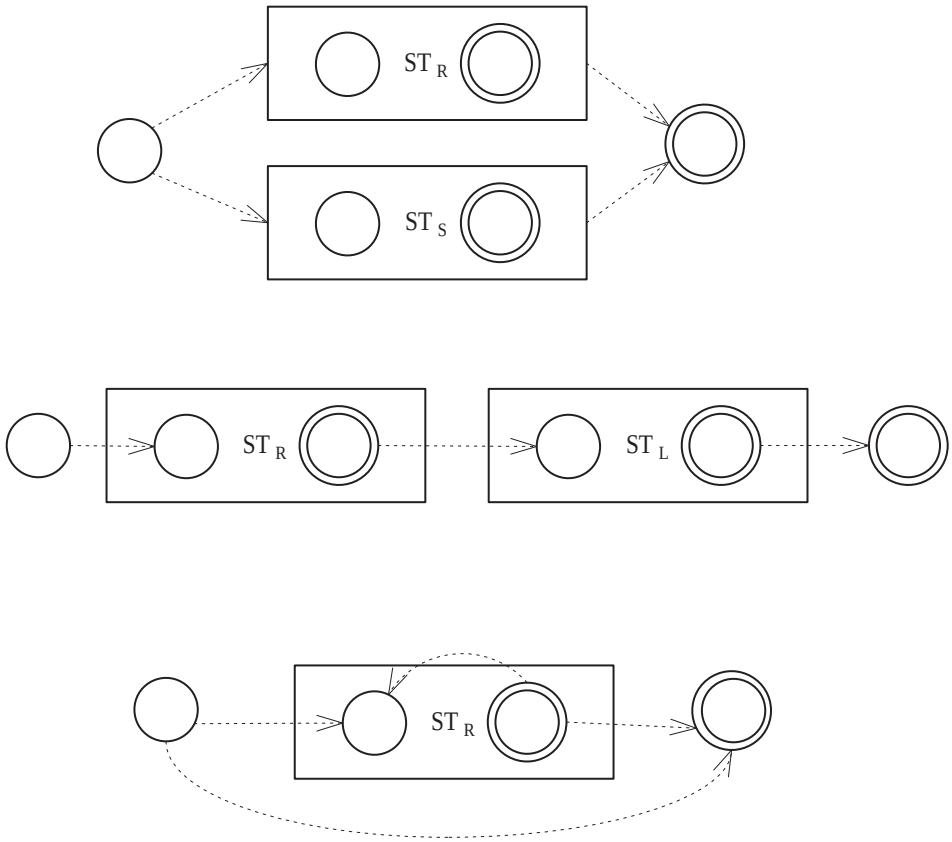

Figura 2.6: Sistemele tranziţionale ce recunosc limbajele $R|S$, $R \bullet S$ şi

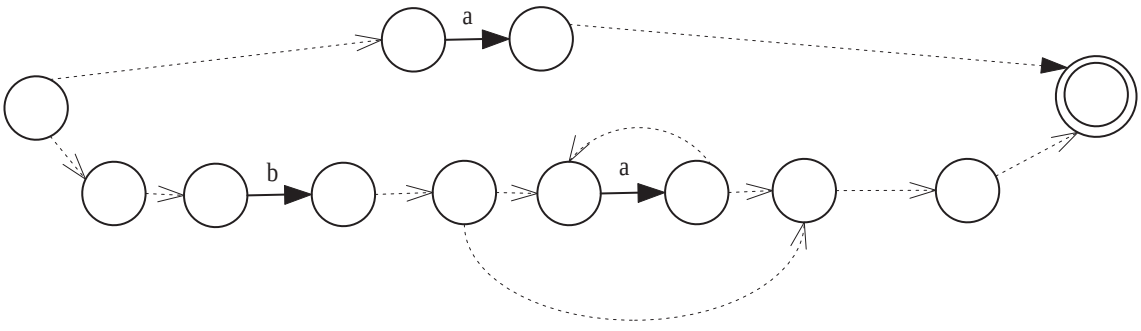

Figura 2.7: Sistemul tranziţional corespunzător expresiei regulate

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 2.8.

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;
  }
```

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 2.8: Coduri asociate unităţilor lexicale

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:

```
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)*

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 2.9 (stările au fost notate prin numere întregi):

Etapa III. Se construieşte sistemul tranziţional (vezi figura 2.10) 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-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).

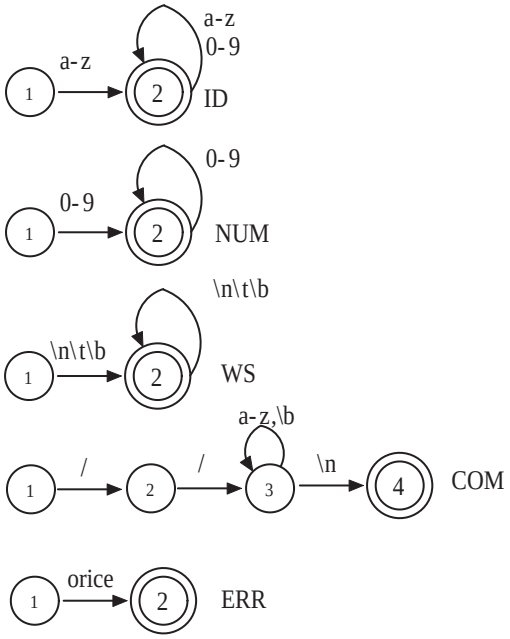

Figura 2.9: Automatele finite corespunzătoare expresiilor regulate

În cazul particular al automatului nostru, diagrama de stări este dată în figura 2.11.

Etapa IV. *Programarea analizorului lexical.*

Automatul finit obţinut are stările finale asociate cu clasele de cuvinte recunoscute. Se asociază acţiuni stărilor finale, corespunzător 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 2.12 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

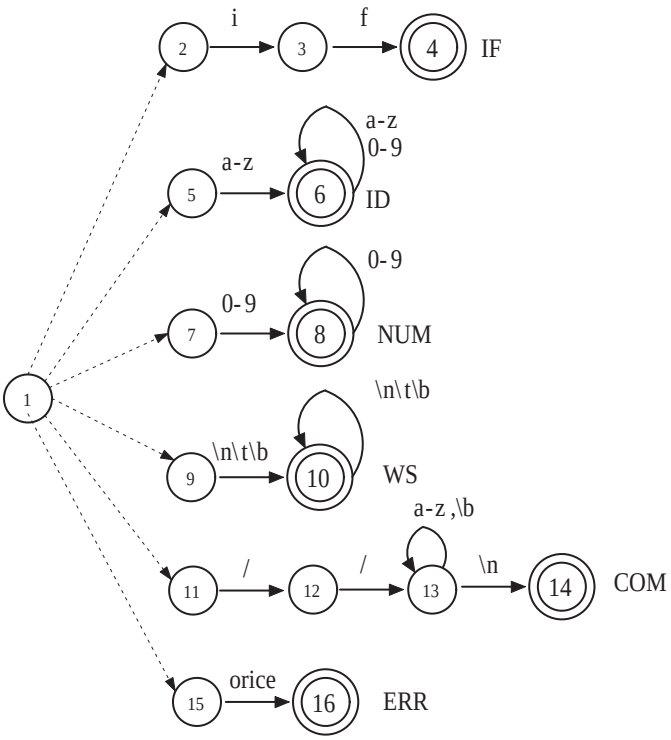

Figura 2.10: Sistemul tranziţional ce recunoaşte reuniunea limbajelor

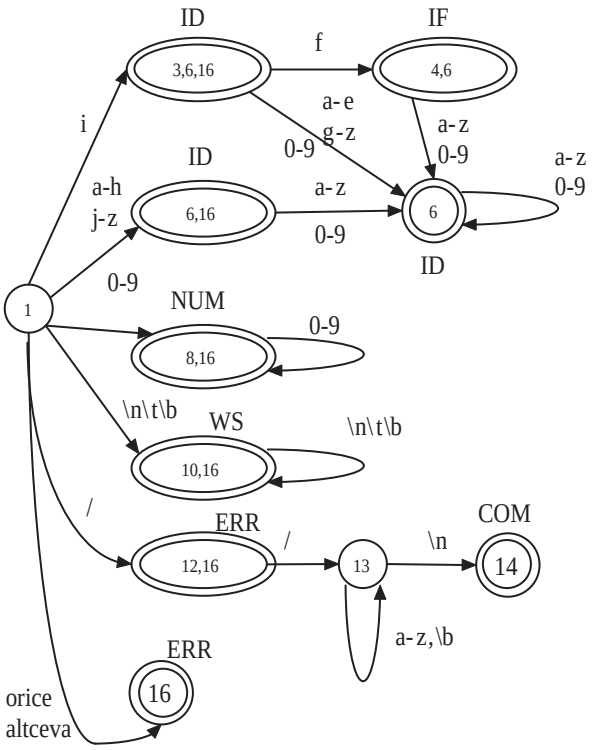

Figura 2.11: Automatul finit determinist ce recunoaşte reuniunea limbajelor

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

Figura 2.12: 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 2.13: Reprezentarea funcţiei de evoluţie a automatului finit

funcţia de evoluţie asociată automatului finit determinist se memorează sub forma unui tablou bidimensional de întregi, ca în figura 2.13. 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.

2.5 Probleme propuse

1. Construiţi automate finite pentru recunoaşterea limbajelor:
   (a) $L = \{PSDR, PNL, PUNR\}$;
   (b) $L = \{w|$ şiruri de 0 şi 1 terminate cu 1 $\}$;
   (c) $L = \{w|w$ identificator PASCAL $\}$;
   (d) $L = \{w|w$ constantă întreagă cu semn în PASCAL $\}$;
   (e) $L = \{w \in \{0,1\}^* | w$ multiplu de 3 $\}$;
   (f) $L = \{a^i b^j | i, j > 0\}$;
   (g) $L = \emptyset$.
2. Construiţi automate finite echivalente cu gramaticile de tipul trei de la problema 1 capitolul 1.

3. Construiţi automate finite deterministe echivalente cu cele nedeterministe obţinute la problema precedentă.

4. Găsiţi gramatici regulate echivalente cu automatele de la problema 1.

5. Folosind lema de pompare pentru limbaje regulate dovediţi că următoarele limbaje nu sunt regulate:

   (a) $L = \{0^{i^2} | i \geq 1\}$;
   (b) $L = \{0^{2^n} | n \geq 1\}$;
   (c) $L = \{0^n | n$ este număr prim $\}$;
   (d) $L = \{0^m 1^n 0^{m+n} | m \geq 1, n \geq 1\}$;

6. Specificaţi limbajele denotate de următoarele expresii regulate:

   (a) $(11|0)^*(00|1)^*$;
   (b) $(1|01|001)^*(\lambda|0|00)$;
   (c) $10|(0|11)0^*1$;
   (d) $((0|1)(0|1))^*$;
   (e) $01^*|1$;
   (f) $((11)^*|101)^*$.

7. Construiţi sisteme tranziţionale ce recunosc limbajele specificate la problema precedentă. Pentru fiecare sistem tranziţional construiţi un automat finit determinist echivalent.

Capitolul 3

Limbaje Independente de Context

3.1 Arbori de derivare

Caracterizarea familiei $\mathcal{L}_2$ cu arbori de derivare. Una din caracteristicile de bază ale limbajelor independente de context este aceea că o derivare într-un astfel de limbaj poate fi reprezentată de un arbore, numit in acest context *arbore de derivare*. Această reprezentare este importantă în mod special pentru faptul că permite o imagine intuitivă simplă a unei derivări şi deci posibilitatea de a lucra uşor cu limbaje de tipul 2.

Vom prezenta în primul rând câteva noţiuni elementare de teoria grafurilor, cu scopul de a preciza notaţiile şi terminologia.

Un *graf orientat* $\mathcal{G}$ este o pereche $\mathcal{G} = (V, \Gamma)$ unde V este o mulţime finită iar Γ o aplicaţie $\Gamma : V \longrightarrow \mathcal{P}(V)$. Mulţimea V se numeşte mulţimea vârfurilor (nodurilor) grafului iar dacă $v_2 \in \Gamma(v_1)$, perechea (v_1, v_2) este un arc în graf; v_1 este *originea* iar v_2 este *extremitatea* arcului. Un *drum* de la vîrful v' la vârful v'' în graful $\mathcal{G}$ este o mulţime de arce $(v_1, v_2)(v_2, v_3) \dots (v_{n-1}, v_n)$ cu $v' = v_1$ şi $v'' = v_n$. Numărul $n - 1$ este *lungimea drumului*. Un drum pentru care $v_1 = v_n$ se numeşte *circuit*. Un circuit de lungime 1 poartă numele de *buclă*.

Definiţie 3.1 *Un arbore este un graf fără circuite, cu* $card(V) \geq 2$ *şi care satisface următoarele două condiţii :*

1. $\exists v_0 \in V$ *astfel încât* $v_0 \notin \Gamma(v), \forall v \in V$*;* v_0 *se numeşte* **rădăcina arborelui***;*

2. $\forall v \in V \setminus \{v_0\}$, $\exists! w$ *cu* $v \in \Gamma(w)$*; altfel spus orice vârf diferit de* v_0 *este extremitatea unui singur arc.*

Exemplu. $V = \{v_0, v_1, v_2, v_3, v_4\}$ iar funcţia Γ este dată de:

x	v_0	v_1	v_2	v_3	v_4
$\Gamma(x)$	$\{v_1, v_2\}$	$\emptyset$	$\{v_3, v_2\}$	$\emptyset$	$\emptyset$

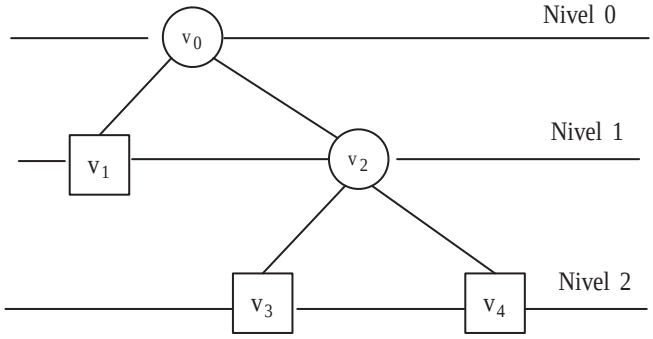

Figura 3.1: Reprezentarea grafică a arborelui $\mathcal{G} = (V, \Gamma)$.

Reprezentarea în plan a acestui arbore este dată in figura 3.1

Nodurile v pentru care $\Gamma(v) = \emptyset$ se numesc noduri *terminale* (finale); celelalte se numesc *interne*. Mulţimea nodurilor terminale constituie *frontiera* arborelui. În general vom nota un arbore cu litere mari, specificînd ca indici rădăcina şi frontiera; de exemplu $\mathcal{A}_{v_0,\, v_1v_2v_3}$. Un arbore comportă mai multe *ramuri*; în exemplu avem următoarele ramuri : v_0v_1, $v_0v_2v_3$, $v_0v_2v_4$.

Fie $G = (V_N, V_T, S, P)$ o gramatică de tipul 2.

Definiţie 3.2 *Un arbore de derivare în gramatica G este un arbore cu următoarele două proprietăţi .*

(1) Nodurile sunt etichetate cu elementele din V_G;

(2) Dacă un nod v are descendenţi direcţi $v_1, \ldots, v_n$ atunci $v \to v_1v_2 \ldots v_n \in P$.

Exemplu. $G = (\{A, B\}, \{a, b\}, A, P)$ unde

$$P = \{A \to aBA|Aa|a, B \to AbB|ba|abb\}.$$

Arborele $\mathcal{A}_{A,\, aabbaa}$ reprezentat în figura 3.2 (Varianta 1) este un arbore de derivare (poate fi desenat coborând frontiera pe nivelul ultim , Varianta 2):

Teorema 3.1 *Fie G o gramatică de tipul 2. Atunci $X \stackrel{*}{\Rightarrow} p$ dacă şi numai dacă există un arbore $\mathcal{A}_{X,\, p}$.*

Demonstraţie. $X \stackrel{*}{\Rightarrow} p$ implică $\exists \mathcal{A}_{X,\, p}$.

Procedăm prin inducţie asupra lungimii derivării l.
Dacă $l = 1$, $X \Rightarrow p = i_1 \ldots i_n$ şi $X \to i_1 \ldots i_n \in P$. Arborele din figura 3.3 corespunde cerinţelor teoremei.

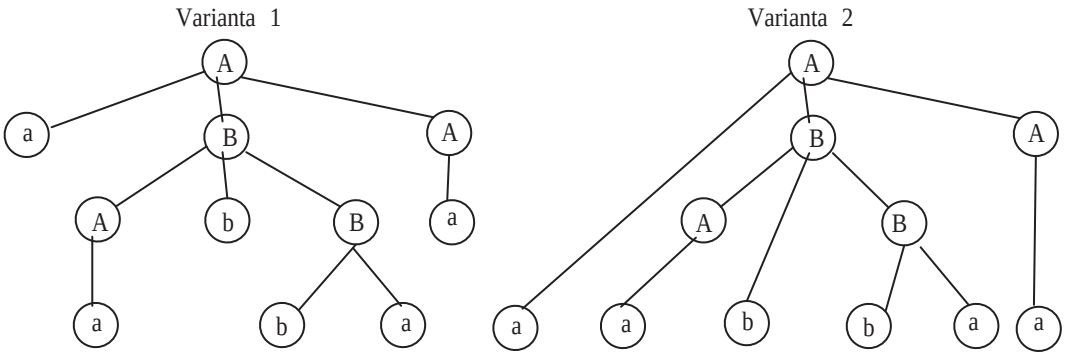

Figura 3.2: Variante de reprezentare a arborelui $\mathcal{A}_{A,\, aabbaa}$.

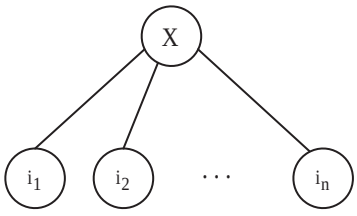

Figura 3.3: Arbore corespunzător unei derivări directe.

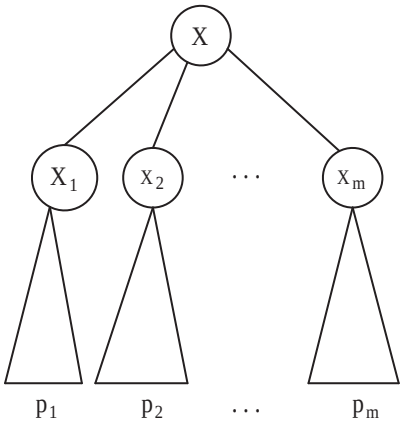

Figura 3.4: Construcţia arborelui $\mathcal{A}_{X,p_1\dots p_m}$.

Presupunem că proprietatea este adevărată pentru derivări de lungime l şi considerăm o derivare de lungime $l+1$, $X \stackrel{*}{\Rightarrow} p$. Punem în evidenţă prima derivare directă

$$X \Rightarrow X_1 \dots X_n \stackrel{*}{\Rightarrow} p$$

Conform lemei de localizare, $p = p_1 \dots p_n$ şi $X_j \stackrel{*}{\Rightarrow} p_j$, $j = \overline{1,m}$. Putem face următoarea construcţie: conform ipotezei inductive, fiecărei derivări $X_j \stackrel{*}{\Rightarrow} p_j$ îi corespunde câte un arbore $\mathcal{A}_{X_j,p_j}$; unim apoi toate nodurile X_j în nodul X plasat la nivelul zero. Obţinem astfel un arbore $\mathcal{A}_{X,p_1\dots p_m} = \mathcal{A}_{X,p}$ (vezi figura 3.4) care corespunde cerinţelor teoremei.

Pentru implicaţia , $\exists \mathcal{A}_{X,p} \Rightarrow X \stackrel{*}{\Rightarrow} p$, se parcurge o cale inversă, făcând o inducţie asupra numărului de nivele. De exemplu, dacă acest număr este 2, arborele de derivare trebuie să arate ca în 3.3 şi deci avem $X \to i_1 i_2 \dots i_n = p \in P$ şi $X \stackrel{*}{\Rightarrow} p$, etc. □

3.2 Ambiguitate în familia $\mathcal{L}_2$.

Fie G o gramatică de tipul 2. O derivare $S = u_0 \Rightarrow u_1 \Rightarrow \dots \Rightarrow u_n$ în care la fiecare derivare directă se înlocuieşte simbolul neterminal cel mai din stânga (dreapta) se numeşte *derivare extrem stângă (dreaptă).* Să observăm că în particular într-o gramatică de tipul 3 orice derivare este extrem dreaptă (scrierea drept liniară).

Definiţie 3.3 *O gramatică G de tipul 2 în care există un cuvânt $p \in L(G)$ care se poate obţine cu două derivări extrem stângi (drepte) distincte, se numeşte ambiguă. In caz contrar este neambiguă.*

Exemplu. Gramatica $A \to aBA|Aa|a,\ B \to AbB|ba|abb$ este ambiguă. Într-adevăr, avem

$$A \Rightarrow aBA \Rightarrow aBa \Rightarrow aAbBa \Rightarrow aAbbaa \Rightarrow aabbaa;$$

$$A \Rightarrow Aa \Rightarrow aBAa \Rightarrow aBaa \Rightarrow aabbaa.$$

Definiţie 3.4 *Un limbaj este ambigu dacă toate gramaticile care îl generează sunt ambigue. În caz contrar (adică dacă există o gramatică neambiguă care să îl genereze) limbajul este neambigu.*

Dacă G este ambiguă şi $p \in L(G)$ este un cuvânt care se poate obţine cu două derivări extrem stângi distincte, atunci există arborii $\mathcal{A}_{S,\,p}$ şi $\mathcal{A}'_{S,\,p}$, diferiţi, dar care au aceiaşi rădăcină şi frontieră.

Teorema 3.2 *Dacă L_1 şi L_2 sunt limbaje disjuncte neambigue, atunci $L_1 \cup L_2$ este neambigu.*

Demonstraţie. Fie $G_k = (V_{N_k}, V_{T_k}, S_k, P_k),\ k = 1, 2$ două gramatici de tipul 2 neambigue şi fie $G = (V_{N_1} \cup V_{N_2}, V_{T_1} \cup V_{T_2}, S, P_1 \cup P_2 \cup \{S \to S_1|S_2\})$ gramatica ce generează limbajul $L(G_1) \cup L(G_2)$.

Să presupunem că $L(G)$ este ambiguă. Atunci există $p \in L(G)$ care se poate obţine cu două derivări extreme stângi diferite. Să presupunem că $p \in L(G_1),\ p \notin L(G_2)$. Atunci obţinem două derivări distincte în gramatica G

$$(1)\ S \underset{G}{\Rightarrow} S_1 \overset{*}{\underset{G}{\Rightarrow}} p,\ deci\ S_1 \overset{*}{\underset{G_1}{\Rightarrow}} p;$$

$$(2)\ S \underset{G}{\Rightarrow} S_1 \overset{*}{\underset{G}{\Rightarrow}} p,\ deci\ S_1 \overset{*}{\underset{G_2}{\Rightarrow}} p,$$

deci şi două derivări extrem stângi în gramatica G_1. Aceasta ar însemna că G_1 este ambiguă. Contradicţie cu ipoteza!□

Teorema 3.3 *Limbajele de tipul 3 sunt neambigue.*

Demonstraţie. Fie L un limbaj de tipul 3 şi G gramatica care îl generează; fie apoi AF automatul finit care recunoaşte limbajul L şi AFD automatul finit echivalent determinist. Construim gramatica G' astfel încât $L(G') = L(AFD)$. Reamintim că regulile lui G' se construiesc astfel $f(A, a) = B \Rightarrow A \to aB,\ \ f(A, a) \in \Sigma_f \Rightarrow A \to a$.

Să presupunem acum că L este ambigu; atunci orice gramatică care îl generează, inclusiv G', este ambiguă. Aceasta înseamnă că există un $p \in L(G')$ astfel încât

$$S \Rightarrow i_1A_1 \Rightarrow i_1i_2A_2 \Rightarrow \ldots \Rightarrow i_1 \ldots i_{n-1}A_{n-1} \left\{ \begin{array}{l} \Rightarrow i_1 \ldots i_nA'_n \Rightarrow \\ \Rightarrow i_1 \ldots i_nA''_n \Rightarrow \end{array} \right\} \overset{*}{\Rightarrow} p.$$

Deci există regulile $A_{n-1} \to i_nA'_n$ şi $A_{n-1} \to i_nA''_n$, adică în automatul AFD avem

$$f(A_{n-1}, i_n) = A'_n,\ \ f(A_{n-1}, i_n) = A''_n,$$

ceea ce contrazice faptul că AFd este determinist.□

3.3 Forme normale pentru gramatici de tipul 2

Forma normală Chomsky.

Definiţie 3.5 *O gramatică în forma normală Chomsky este o gramatică cu reguli de forma*

$$A \to BC,$$
$$D \to i,$$

unde $A, B, C, D \in V_N$ *şi* $i \in V_T$. *Se acceptă şi regula de completare* $S \to \lambda$ *cu condiţia ca* S *să nu apară în dreapta vreunei reguli.*

Lema 3.1 *(lema substituţiei). Fie* G *o gramatică de tipul 2 şi* $X \to uYv$ *precum şi* $Y \to p_1 \dots p_n$ *toate regulile din* G *care au* Y *în stânga . Atunci* G *este echivalentă cu o gramatică* G' *în care am făcut "substituţiile"; adică facem următoarea înlocuire*

$$X \to uYv \text{ se înlocuieşte cu } X \to up_1v| \dots |up_nv$$

(Regulile $Y \to p_1| \dots |p_n$ *le vom păstra neschimbate).*

Demonstraţie. Fie $p \in L(G)$ şi $S \overset{*}{\Rightarrow} p$. Punem în evidenţă doi paşi consecutivi oarecare:

$$G: \quad S \overset{*}{\Rightarrow} r \Rightarrow s \Rightarrow t \overset{*}{\Rightarrow} p.$$

Dacă în $r \Rightarrow s$ se utilizează regula $X \to uYv$ atunci în mod necesar în pasul următor se utilizează una din regulile $Y \to p_1| \dots |p_n$, să presupunem $Y \to p_j$ (evident, este posibil ca această regulă să nu se aplice în pasul imediat următor, dar ea poate fi "adusă" în această poziţie). Prin urmare

$$(A) \quad G: \ r = r'Xr'' \Rightarrow r'uYvr'' \Rightarrow r'up_jvr'' = t.$$

Aceşti doi paşi se pot obţine şi în G' (într-un singur pas):

$$(B) \quad G': \ r = r'Xr'' \Rightarrow r'up_jr'' = t.$$

Deci $S \underset{G'}{\overset{*}{\Rightarrow}} p, p \in L(G')$ şi $L(G) \subseteq L(G')$.

Invers, dacă $p \in L(G')$ şi $S \underset{G'}{\overset{*}{\Rightarrow}} p$, atunci dacă la un pas se utilizează o regulă nou introdusă (pasul (B)), transformarea respectivă se poate obţine şi în G cu doi paşi (paşii (A)); deci $p \in L(G)$ şi $L(G') \subseteq L(G)$.□

Corolar 3.4 *Orice gramatică de tipul 2 este echivalentă cu o gramatică de acelaşi tip în care mulţimea de reguli nu conţine* redenumiri. *(O redenumire este o regulă de forma* $A \to B, \ A, B \in V_N$*).*

Intr-adevăr, dacă $A \to B \in P$ este o redenumire şi $B \to p_1|\dots|p_n$ sunt toate regulile care au B în stânga, efectuăm substituţiile, deci înlocuim regula $A \to B$ cu $A \to p_1|\dots|p_n$. În cazul în care printre acestea apare o nouă redenumire, repetăm procedeul.□

Exemplu. Gramatica G_E care generează expresii aritmetice $E \to E+T|T, T \to T*F|F, F \to (E)|i$ se poate pune sub următoarea formă (fără redenumiri) :

$$E \to E+T|T*F|(E)|i$$
$$T \to T*F|(E)|i$$
$$F \to (E)|i$$

Teorema 3.5 *(teorema lui Chomsky de existenţă a formei normale). Orice gramatică independentă de context este echivalentă cu o gramatică în forma normală Chomsky. ema2*

Demonstraţie. Putem porni de la o gramatică G care nu are redenumire şi ale cărei reguli cu terminale au forma $A \to i, A \in V_N, i \in V_T$. De asemenea presupunem că G nu are reguli de ştergere.

Rezultă că regulile lui G au una din formele:
(1) $A \to BC$,
(2) $D \to i$,
(3) $X \to X_1 \dots X_n, n > 2$.

Construim o gramatică $G' = (V'_N, V_T, S, P')$ unde $V_N \subseteq V'_N$ şi P' conţine toate regulile din P de forma (1) şi (2). Fiecare regulă de forma (3) o înlocuim cu:

$$X \to X_1 Z_1,$$
$$Z_1 \to X_2 Z_2,$$
$$\dots$$
$$Z_{n-2} \to X_{n-1} X_n$$

şi includem neterminalele $Z_1, \dots, Z_{n-2}$ (altele pentru fiecare regulă de forma (3) în V'_N.

Se poate relativ uşor arăta că $L(G) = L(G')$. De exemplu, dacă $u \Rightarrow v$ (direct) în G şi de aplică o regulă de forma (1) sau (2), atunci evident derivarea respectivă se poate obţine şi în G'; în cazul în care se aplică o regulă de forma (3), avem

$$G: \quad u = u'Xu'' \Rightarrow u'X_1 \dots X_n u'' = v.$$

Această derivare se poate obţine şi în G' în mai mulţi paşi şi anume

$$G': \quad u = u'Xu'' \Rightarrow u'X_1Z_1u'' \Rightarrow u'X_1X_2Z_2u'' \Rightarrow \dots \Rightarrow u'X_1 \dots X_n u'' = v.\square$$

Observaţie. O gramatică ce are reguli de forma $A \to BC, A \to B, A \to a$ unde $A, B, C \in V_N$ şi $a \in V_T$ spunem că este în *forma 2–canonică.* Este evident că orice gramatică de tip 2 este echivalentă cu o gramatică în formă 2–canonică.

Gramatici recursive

Definiţie 3.6 *Un simbol neterminal X al unei gramatici de tipul 2 este recursiv dacă există o regulă de forma $X \to uXv,\ u, v \in V_G^*$.*

Dacă $u = \lambda$ ($v = \lambda$) simbolul X este stâng (drept) recursiv. O gramatică ce are cel puţin un simbol recursiv se numeşte recursivă. De exemplu, gramatica G_E care generează expresiile aritmetice are două simboluri stâng recursive, E şi T.

Existenţa simbolurilor stâng recursive poate provoca dificultăţi în aplicarea algoritmilor de analiză top-down. Într-adevăr, într-o astfel de gramatică, încercarea de a construi arborele de derivare corespunzător unui cuvânt p prin aplicarea întotdeauna a primei reguli pentru simbolul cel mai din stânga, poate să conducă la un ciclu infinit (de exemplu în G_E s-ar obţine $E \Rightarrow E+T \Rightarrow E+T+T \Rightarrow \ldots$).

Teorema 3.6 *Orice limbaj de tipul 2 poate să fie generat de o gramatică fără recursie stângă.*

Demonstraţie. Fie $G = (V_N, V_T, S, P)$ o gramatică de tipul 2; presupunem că G are un singur simbol recursiv X şi fie

$$(A)\ \ X \to u_1|u_2|\ldots|u_n|Xv_1|\ldots|Xv_m$$

toate regulile care au X în stânga. Construim gramatica $G' = (V_N', V_T, S, P')$, unde $V_N \subset V_N',\ P \subset P'$ cu excepţia regulilor (A); acestea se înlocuiesc cu

$$\begin{array}{l} X \to u_1|u_2|\ldots|u_n|u_1Y|u_2Y|\ldots|u_nY, \\ Y \to v_1|\ldots|v_m|v_1Y|\ldots|v_mY \end{array}$$

G' este de tipul 2 şi nu are simboluri stâng recursive; se vede însă că Y este un simbol drept recursiv.

Fie $p \in L(G), S \overset{*}{\underset{G}{\Rightarrow}} p$. Dacă în această derivare nu intervine simbolul recursiv, atunci evident că $S \overset{*}{\underset{G'}{\Rightarrow}} p$. Să presupunem că X intervine la un anumit pas: $S \Rightarrow u \Rightarrow p$, unde $u = u'Xu''$. Putem aplica, începând de la u spre dreapta, în primul rând regulile pentru X şi să urmărim numai subarborele respectiv, deci

$$G:\ \ X \underset{G}{\Rightarrow} Xv_{j_1} \underset{G}{\Rightarrow} Xv_{j_2}v_{j_1} \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} Xv_{j_s}\ldots v_{j_1} \underset{G}{\Rightarrow} u_jv_{j_s}\ldots v_{j_1}.$$

Aceeaşi formă propoziţională o putem obţine şi în gramatica G' astfel

$$G':\ \ X \underset{G'}{\Rightarrow} u_jY \underset{G'}{\Rightarrow} u_jv_{j_s}Y \underset{G'}{\Rightarrow} \ldots \underset{G'}{\Rightarrow} u_jv_{j_s}\ldots v_{j_1}.$$

Prin urmare avem $S \underset{G'}{\Rightarrow} u \underset{G'}{\Rightarrow} p$, adică $p \in L(G')$ şi $L(G) \subseteq L(G')$. Analog, $L(G') \subseteq L(G)$.□

Forma normală Greibach.

Definiţie 3.7 *O gramatică în forma normală Greibach este o gramatică cu reguli de forma*

$$A \to ip, \ \ unde \ \ A \in V_N, \ i \in V_T \ p \in V_N^*.$$

Se acceptă şi regula de completare $S \to \lambda$ cu condiţia ca S să nu apară în dreapta vreunei reguli.

Teorema 3.7 *(Teorema de existenţă a formei normale Greibach). Orice gramatică de tipul 2 este echivalentă cu o gramatică în forma normală Greibach.*

Demonstraţie. Fie G o gramatică de tipul 2 în forma normală Chomsky şi fie $V_N = \{S = X_1, X_2, \ldots, X_n\}$. Vom construi o gramatică echivalentă care să satisfacă cerinţele din forma normală Greibach în mai multe etape.
Etapa I. Vom modifica regulile de generare astfel încât toate regulile care nu sunt de forma $X \to i$ să satisfacă condiţia $X_j \to X_k p, \ j < k, \ p \in V_N^*$. Acest lucru îl facem cu un algoritm pe care îl prezentăm într-un limbaj nestandard de publicare (tip PASCAL):

$j := 1$;
e1: **begin**
Se elimină recursiile stângi; neterminalele
noi le notăm cu $Y_1, Y_2, \ldots$
end
if $j = n$ **then** STOP;
$j := j + 1$;
$l := 1$;
e2: **begin**
Fie $X_j \to X_l p, \ p \in V_N^*$ şi $X_l \to p_1 \ldots p_m$
toate regulile care au X_l în stânga; se efectuează toate substituţiile.
end
$l := l + 1$;
if $l < j - 1$ **then goto** e2
goto e1

Să observăm că pentru $j = 1$ şi după eliminarea recursiilor stângi condiţia cerută este evident îndeplinită; în plus, dacă au fost recursii, vom avea reguli cu partea stângă neterminale noi $Y_1, Y_2, \ldots$. Mai departe, luăm toate regulile care au în stânga X_2 ($j := j + 1 = 2$) şi efectuăm substituţiile; acestea se vor transforma în $X_2 \to X_k p$ cu $k \geq 2$ şi după o nouă eliminare a recursiilor stângi vom avea $k > 2$ plus reguli cu partea stângă neterminale noi. În felul acesta toate regulile care au în stânga X_1 şi X_2 satisfac condiţia cerută; în continuare $j := j + 1 = 3$, etc.

Etapa II. Avem acum trei categorii de reguli:

(1) $X_j \to i$;
(2) $X_j \to X_k p,\ j < k, p \in V_N^*$;
(3) $Y \to iq,\ q \in V_N^*,\ i = 1, \ldots, m$.

Aranjăm toate neterminalele într-un şir unic, la început $Y_1, \ldots, Y_m$ apoi $X_1, \ldots, X_n$ şi le redenumim, de exemplu cu $X_1, \ldots, X_{m+n}$:

$$\begin{array}{cccccccc} Y_1, & Y_2, & \ldots, & Y_m, & X_1, & X_2, & \ldots, & X_n \\ X_1, & X_2, & \ldots, & X_m, & X_{m+1}, & X_{m+2}, & \ldots, & X_{m+n} \end{array}$$

Vom nota $n + m = N$. În felul acesta regulile gramaticii vor avea numai formele (1) şi (2).

Etapa III. Toate regulile care au X_N în stânga vor avea forma (1). Fie $X_{n-1} \to X_N p_1 | \ldots | X_N p_n$ toate regulile care au X_{N-1} în stânga şi care nu sunt de forma (1). Efecuăm substituţiile lui X_N; în acest fel regulile care au X_N şi X_{N-1} în stânga satisfac cerinţele din forma normală Greibach. În continuare, considerăm toate regulile care au X_{N-2} în stânga şi efectuăm substituţiile, etc.□

Forma normală operator

Una din formele importante pentru gramatici independente de context, utilizată în analiza sintactică prin metoda precedenţei, este forma operator a acestor gramatici.

Definiţie 3.8 *O gramatică independentă de context $G = (V_N, V_T, S, P)$ se spune că este în* forma normală operator *dacă oricare ar fi producţia $A \to \beta \in P$, în β nu apar două neterminale (*variabile*) consecutive, adică*

$$P \subseteq V_N \times [(V_N \cup V_T)^* \setminus (V_N \cup V_T)^* V^2 (V_N \cup V_T)^*].$$

Teorema 3.8 *Orice gramatică independentă de context este echivalentă cu o gramatică în forma normală operator.*

Demonstraţie. Fie $G = (V_N, V_T, S, P)$ o gramatică de tipul 2 şi $L(G)$ limbajul generat. Fără a restrânge generalitatea presupunem că $\lambda \notin L(G)$ şi G este în forma 2–canonică (regulile sunt de forma $A \to BC,\ A \to B,\ A \to a$ vezi teorema **??**). Definim o gramatică echivalentă $G' = (V'_N, V_T, S, P')$ astfel: $V'_N = \{S\} \cup (V_N \times V_T)$, iar $P' = P_1 \cup P_2 \cup P_3 \cup P_4$ unde

$i)$ $P_1 = \{S \to (S, a)a |\ a \in V_T\}$;
$ii)$ $P_2 = \{(A, a) \to \lambda |\ A \in V_N, a \in V_T, A \to a \in P\}$;
$iii)$ $P_3 = \{(A, a) \to (B, a) |\ A, B \in V_N, a \in V_T, A \to B \in P\}$;
$iv)$ $P_4 = \{(A, a) \to (B, b)b(C, a) |\ A, B, C \in V_N, a, b \in V_T, A \to BC \in P\}$.

Să observăm că G' este în forma normală operator. Pentru a demonstra că $L(G) = L(G')$ vom defini mai întâi o funcţie $\phi : P_2 \cup P_3 \cup P_4 \longrightarrow P$ astfel:

$\phi((A, a) \to \lambda) = A \to a$ *pentru* $(A, a) \to \lambda \in P_2$;
$\phi((A, a) \to (B, a)) = A \to B$ *pentru* $(A, a) \to (B, a) \in P_3$;
$\phi((A, a) \to (B, b)b(C, a)) = A \to BC$ *pentru* $(A, a) \to (B, b)b(C, a) \in P_4$.

Funcţia ϕ se extinde în mod natural la $\phi' : (P_2 \cup P_3 \cup P_4)^* \longrightarrow P^*$. Vom arăta că în gramatica G, $\forall w \in V_T^*,\ \forall a \in V_T$ are loc derivarea extrem dreaptă $A \underset{G}{\overset{*}{\Rightarrow}} wa$ folosind producţiile $\pi_1, \pi_2, \ldots, \pi_n$ dacă şi numai dacă există în P' producţiile $\pi_1', \pi_2', \ldots, \pi_n'$ astfel ca $\phi(\pi_i') = \pi_i,\ 1 \leq i \leq n$ şi în G' are loc derivarea extrem dreaptă $(A, a) \underset{G'}{\overset{*}{\Rightarrow}} w$ folosind producţiile $\pi_1', \pi_2', \ldots, \pi_n'$.

Să demonstrăm afirmaţia prin inducţie după n, lungimea derivării.

Dacă $n = 1$ atunci $w = \lambda, A \to a \in P$ şi în P' există producţia $(A, a) \to \lambda$, deci $(A, a) \Rightarrow \lambda$ şi $\phi((A, a) \to \lambda) = A \to a$.

Invers, dacă $(A, a) \Rightarrow w$ în G' atunci $w = \lambda$ (după forma producţiilor din G') şi are loc proprietatea enunţată.

Să presupunem afirmaţia adevărată pentru derivări de lungime cel mult $n-1$ şi să o demonstrăm pentru derivări de lungime $n > 1$. Fie aşadar $A \underset{G}{\overset{*}{\Rightarrow}} wa$ o derivare de lungime n în gramatica G şi punem în evidenţă prima derivare directă. Distingem două cazuri:

I. Prima producţie utilizată în derivare este $A \to B$. Atunci,

$$A \underset{G}{\Rightarrow} B \underset{G}{\overset{*}{\Rightarrow}} wa$$

şi conform ipotezei inductive avem în G' o derivare $(B, a) \underset{G'}{\overset{*}{\Rightarrow}} w$ (de lungine $n-1$) cu producţii satisfăcând condiţiile arătate. Dar cum $A \to B \in P$, în P' avem producţia $(A, a) \to (B, a)$ deci $(A, a) \underset{G'}{\overset{*}{\Rightarrow}} w$ în gramatica G'.

II. Prima producţie este de forma $A \to BC$. Atunci

$$A \underset{G}{\Rightarrow} BC \underset{G}{\overset{*}{\Rightarrow}} wa.$$

În acest caz $wa = ubva$ (conform lemei de localizare), astfel că $B \underset{G}{\overset{*}{\Rightarrow}} ub$ şi $C \underset{G}{\overset{*}{\Rightarrow}} va$. După ipoteza inductivă, vom avea în G' derivările:

$$(B, b) \underset{G'}{\overset{*}{\Rightarrow}} u, \quad (C, a) \underset{G'}{\overset{*}{\Rightarrow}} v.$$

Cum $A \to BC \in P$ vom avea în P' producţia $(A, a) \to (B, b)b(C, a)$ şi în G' putem scrie derivarea extrem dreaptă

$$(A, a) \Rightarrow (B, b)b(C, a) \underset{G'}{\overset{*}{\Rightarrow}} (B, b)bv \underset{G'}{\overset{*}{\Rightarrow}} ubv = w$$

şi producţiile care s-au aplicat îndeplinesc condiţiile din enunţ.

În mod analog se demonstrează reciproca.

Din această afirmaţie, luând în particular $A = S$, obţinem:

$$S \underset{G}{\overset{*}{\Rightarrow}} wa \quad \Leftrightarrow \quad (S, a) \underset{G'}{\overset{*}{\Rightarrow}} w, \ \ \forall w \in V_T^*, \forall a \in V_T.$$

Cum în G' există şi producţia $S \to (S, a)a$, am găsit: $wa \in L(G)$ dacă şi numai $wa \in L(G')$, deci cele două gramatici sunt echivalente.

Pentru a încheia demonstraţia trebuie să considerăm şi cazul $\lambda \in L(G)$. Aplicăm construcţia de mai sus unei gramatici ce generează $L(G) \setminus \{\lambda\}$ şi obţinem $G' = (V'_N, V_T, S, P')$ gramatica operator corespunzătoare. Considerăm acum gramatica $G_1 = (V_{N_1}, V_T, S_1, P_1)$ unde $V_1 = V' \cup \{S_1\}$, $P_1 = P' \cup \{S_1 \to \lambda, S_1 \to S\}$ care este în forma normală operator şi $L(G_1) = L(G)$.□

3.4 Lema Bar–Hillel

Ca şi în cazul limbajelor regulate, lema Bar–Hillel pune în evidenţă următoarea proprietate a unui limbaj independent de context: orice cuvânt al unui astfel de limbaj, suficient de lung, conţine un subcuvânt (nevid) pe care multiplicându–l de un număr arbitrar de ori, obţinem noi cuvinte care aparţin de asemenea limbajului. Mai poartă denumirea de "lema de pompare" sau "lema *uvwxy*".

Ne vom referi în cele ce urmează la gramatici de tipul 2 în formă normală Chomsky. Într-o gramatică de această formă arborii de derivare sunt <u>întotdeauna</u> arbori binari.

Lema 3.2 *Fie G o gramatică în forma normală Chomsky şi $X \overset{*}{\Rightarrow} p, p \in V_G^*$. Dacă cea mai lungă ramură din arborele $\mathcal{A}_{X,p}$ conţine m noduri, atunci $|p| \leq 2^{m-1}$.*

Demonstraţie. Procedăm prin inducţie asupra lui m.

Pentru $m = 2$ arborele are două nivele şi în mod evident $|p| \leq 2 = 2^{m-1}$.

Presupunem că proprietatea este adevărată pentru un m oarecare şi considerăm un arbore care are pe cea mai lungă ramură $m + 1$ noduri. În derivarea $X \overset{*}{\Rightarrow} p$ punem în evidenţă prima derivare directă

$$X \Rightarrow Y\,Z \overset{*}{\Rightarrow} p.$$

Conform lemei de localizare, $p = p_1 p_2$ şi $Y \overset{*}{\Rightarrow} p_1, Z \overset{*}{\Rightarrow} p_2$. Arborii de derivare $\mathcal{A}_{Y,p_1}$ şi $\mathcal{A}_{Z,p_2}$ conţin, fiecare pe cea mai lungă ramură cel mult m noduri; conform ipotezei de inducţie, avem $|p_1| \leq 2^{m-1}, |p_2| \leq 2^{m-1}$. Prin urmare

$$|p| = |p_1 p_2| = |p_1| + |p_2| \leq 2^{m-1} + 2^{m-1} = 2^m.\square$$

Observaţie. Dacă $X \overset{*}{\Rightarrow} p$ şi $|p| > 2^{m-1}$ atunci există în arborele $\mathcal{A}_{X,p}$ cel puţin o ramură cu $m + 1$ noduri.

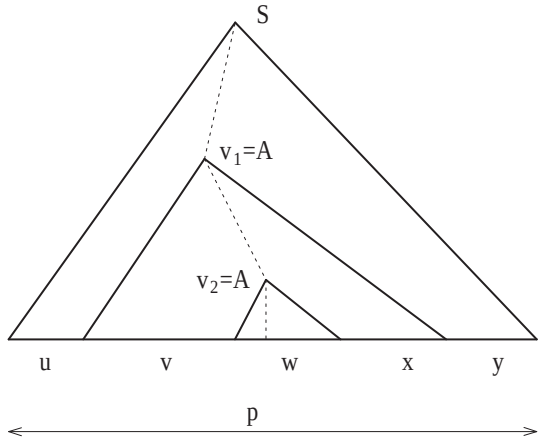

Figura 3.5: Descompunerea cuvântului p.

Lema 3.3 *Fie G o gramatică de tipul 2 şi fie $X \Rightarrow X_1 \dots X_m \stackrel{*}{\Rightarrow} p$. Atunci, conform lemei de localizare, $p = p_1 \dots p_m$ şi $X_j \stackrel{*}{\Rightarrow} p_j, j = 1, \dots, m$. Fie $\mathcal{A}_{Y,q} \subseteq \mathcal{A}_{X,p}$. atunci q este subcuvânt într–unul din cuvintele p_j.*

Demonstraţie. Neterminalul Y aparţine unuia din subarborii $\mathcal{A}_{X_j,p_j}$, fie acesta $\mathcal{A}_{X_k,p_k}$; atunci $\mathcal{A}_{Y,q} \subseteq \mathcal{A}_{X_k,p_k}$ şi $q \in Sub(p_k)$. □

Lema 3.4 *(Lema Bar–Hillel) Fie E un limbaj independent de context. Atunci există un număr natural k astfel încât, dacă $p \in E$ şi $|p| > k$ atunci p se poate descompune în forma $p = uvwxy$ cu următoarele proprietăţi:*

1. *$vx \neq \lambda$;*
2. *$|vwx| \geq k$;*
3. *$uv^j wx^j y \in E, \forall j \in N$,*

Demonstraţie. Fie $n = card(V_N)$. Luăm $k = 2^n$. Cum $|p| > k = 2^n$, conform observaţiei de la prima lemă, există în arborele $\mathcal{A}_{S,p}$ cel puţin o ramură cu $n+2$ noduri; pe această ramură ultimul nod este terminal, deci există $n+1$ noduri etichetate cu acelaşi simbol A. Descompunem cuvântul p ca în figura 3.5, $p = uvwxy$.

(1) Considerăm subarborele $\mathcal{A}_{A,vwx}$ căruia îi corespunde derivarea $A \stackrel{*}{\Rightarrow} vwx$. Punem în evidenţă primul pas

$$A \Rightarrow BC \stackrel{*}{\Rightarrow} vwx,$$

şi vwx se descompune în $vwx = p_B p_C, B \stackrel{*}{\Rightarrow} p_B, C \stackrel{*}{\Rightarrow} p_C$ şi $p_B, p_C \neq \lambda$. Cum $\mathcal{A}_{A,w} \subseteq \mathcal{A}_{A,vwx}$, rezultă că w este subcuvânt în p_B sau în p_C. Să presupunem că $w \in Sub(p_C)$; atunci $p_B \in Sub(v)$ şi cum $p_B \neq \lambda$ rezultă $v \neq \lambda$.

(2)Putem alege nodurile v_1 şi v_2 astfel încât pe ramura punctată începând de la v_1 în jos până la frontieră să avem exact $n+1$ noduri. Rezultă conform lemei precedente, $|vwx| \geq 2^n = k$.

(3) putem scrie derivările $A \stackrel{*}{\Rightarrow} w, A \stackrel{*}{\Rightarrow} vAx$. Deci

$$S \stackrel{*}{\Rightarrow} uAy \stackrel{*}{\Rightarrow} uvAxy \stackrel{*}{\Rightarrow} uv^2Ax^2y \stackrel{*}{\Rightarrow} \ldots \stackrel{*}{\Rightarrow} uv^jwx^jy. \square$$

Problema închiderii familiei $\mathcal{L}_2$ la intersecţie

Teorema 3.9 *Familia $\mathcal{L}_2$ nu este închisă la intersecţie.*

Demonstraţie. Considerăm limbajul $L_3 = \{a^nb^nc^n | n \geq 1\}$. Să rătăm că $L_3 \notin \mathcal{L}_2$. Într–adevăr, să presupunem că $L_3 \in \mathcal{L}_2$ şi fie k constanta din lema Bar–Hillel. Luăm $n > k/3$ şi fie $p = a^nb^nc^n$; avem $|p| > k$, deci conform acestei leme p se descompune în forma $p = uvwxy$ cu $vx \neq \lambda$ şi $uv^jwx^jy \in L_3, j \in N$.

Vom analiza posibilităţile de constituire a subcuvintelor v şi x. Să presupunem că în v (sau x) intră două din simbolurile a, b, c; de exemplu $v = aabbb$. atunci considerăm cuvântul $p_2 = uv^2wx^2y = uaabbaabbwx^2y$ care nu are structura cuvintelor lui L_3 ("a" nu poate să urmeze după "b") dar conform lemei Bar-Hillel aparţine lui L_3. Deci, în v (sau x) nu pot intra două (sau trei) din simbolurile a, b, c. Să presupunem că intră un singur simbol; de exemplu $v \in \{a\}^*$ şi $x \in \{b\}^*$. Atunci multiplicând în p subcuvintele v şi x, puterile simbolurilor "a" şi "b" se măresc iar "c" rămâne pe loc, contrazicându–se din nou structura cuvintelor din L_3. În concluzie L_3 nu este independent de context.

Fie acum limbajele

$$L_3' = \{a^mb^nc^n | m, n \geq 1\}$$
$$L_3^{"} = \{a^nb^nc^m | m, n \geq 1\}.$$

Se poate vedea uşor că aceste limbaje sunt de tipul 2; gramaticile care le generează sunt $S \to aS|aA, A \to bAc|bc$ şi respectiv $S \to Sc|Ac, A \to aAb|ab$.

Avem $L_3' \cap L_3^{"} = L_3$, deci intersecţia a două limbaje de tipul 2 este un limbaj care nu aparţine lui $\mathcal{L}_2$.$\square$

Corolar 3.10 *Familia $\mathcal{L}_2$ nu este închisă la complementariere.*

Într–adevăr, să presupunem că $\mathcal{L}_2$ este închisă la complementariere şi fie $E_1, E_2 \in \mathcal{L}_2$. Cum familia $\mathcal{L}_2$ este închisă la reuniune, ar rezulta $C(E_1) \cup C(E_2) \in \mathcal{L}_2$. Dar (de Morgan) $C(E_1) \cup C(E_2) = C(E_1 \cap C(E_2) \ \in \mathcal{L}_2$.

Complementul limbajului $C(E_1 \cap E_2)$ este $E_1 \cap E_2$ şi conform presupunerii ar trebui ca $E_1 \cap E_2 \in \mathcal{L}_2$, oricare ar fi E_1, E_2, ceea ce nu este adevărat.$\square$

Generalizări ale lemei Bar–Hillel Lema lui Bar–Hillel reprezintă o condiţie necesară ca un limbaj să fie independent de context. cu ajutorul ei se poate demonstra relativ uşor că anumite limbaje nu sunt independente de context. Ideea este aceea că prin multiplicarea subcuvintelor v şi x, de un număr suficient de ori, se contrazice structura limbajului.

Totuşi, nu orice limbaj care nu este de tipul 2 poate fi respins de lema lui Bar–Hillel. De exemplu, această lemă nu poate respinge limbajul

$$L = \{a^n b^{2^m} | n, m \geq 1\} \cup \{b\}^*,$$

care nu este de tipul 2 (se demonstrează pe altă cale). Într–adevăr, pentru orice $p = a^n b^{2^m}$ luăm

$$u = \lambda, v = a, w = \lambda, x = \lambda, y = a^{n-1} b^{2^m}.$$

Putem itera subcuvintele v şi x fără să contrazicem structura cuvintelor limbajului.

O generalizare a lemei Bar–Hillel este

Lema 3.5 *(Lema lui Ogden) pentru orice limbaj independent de context L există o constantă k astfel încât orice $p \in L$ pentru care cel puţin k simboluri sunt "marcate", se poate descompune în forma $p = uvwxy$ astfel încât*

1. *sau u, v, w sau w, x, y conţin fiecare câte un simbol marcat;*

2. *vwx conţine cel mult k simboluri marcate;*

3. *$uv^j wx^j y \in L, \forall j \in N$*

Cu ajutorul acestei leme se poate arăta că limbajul de mai sus nu este de tipul 2, considerând marcate simbolurile b.

3.5 Automate push-down (APD)

Automatele push-down sunt mecanisme pentru recunoaşterea limbajelor independente de context.

Un APD se compune din (vezi figura 3.6):

1. O bandă de intrare care conţine simboluri ale unui *alfabet de intrare*; aceste simboluri constituie pe o bandă un anumit cuvânt peste alfabetul de intrare. Banda se mişcă numai spre stânga;

2. O *memorie push-down* (memorie inversă, stivă, pilă, etc) care conţine simboluri ale unui alfabet propriu, numit *alfabetul memoriei push-down*. Această memorie funcţionează ca o stivă - ultimul introdus, primul extras (Last In, First Out);

3. Un *dispozitiv de comandă* care se află permanent într-o anumită stare internă aparţinând unei mulţimi finite de stări. Dispozitivul de comandă posedă un *dispozitiv de citire* de pe banda de intrare şi un *dispozitiv de scriere-citire* în memoria push-down.

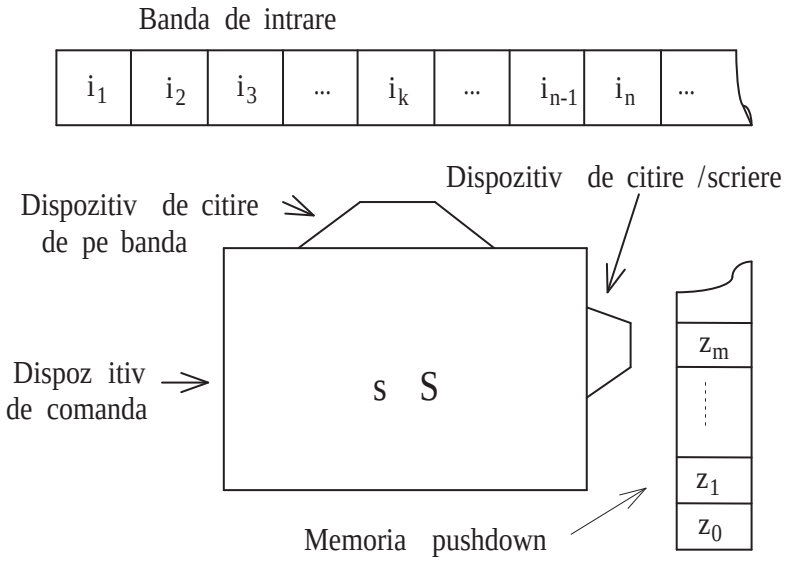

Figura 3.6: Reprezentare schematică a unui automat push-down.

Ca şi un automat finit, un automat push-down funcţionează în paşi discreţi; un pas de funcţionare comportă:

1. Dispozitivul de comadă citeşte simbolul de pe banda de intrare din dreptul dispozitivului de citire şi mută banda cu o poziţie spre stânga.

2. În funcţie de starea internă, de simbolul citit şi de simbolul din vârful memoriei push-down dispozitivul de comandă efectuează operaţiile:
   (a) Trece într-o nouă stare;
   (b) Scrie în memoria push-down un anumit cuvânt peste alfabetul memoriei push-down; în particular, acesta poate să fie cuvântul vid, ceea ce are ca efect ştergerea simbolului din vârful memoriei push-down.

Funcţionarea unui APD se termină în general după ce s-a citit ultimul simbol al cuvântului scris pe banda de intrare dar este posibil ca el să efectueze un anumit număr de paşi, citind de fiecare dată de pe bandă cuvântul vid λ. De asemenea, este posibil ca în timpul funcţionării, deci înainte de a ajunge la ultimul simbol, automatul să se blocheze. De exemplu, automatul ajunge într-o configuraţie (stare, simbol pe bandă, simbol în vârful memoriei push-down) inadmisibilă sau se goleşte memoria push-down dar nu s-a epuizat cuvântul de pe bandă, etc.

Matematic, un APD se defineşte astfel:

Definiţie 3.9 *Un automat push-down este un sistem*
$APD = (\Sigma, I, Z, f, s_0, z_0)$
unde:

Σ *este mulţimea de stări (finită şi nevidă);*
I *este alfabetul de intare;*
Z *este alfabetul memoriei push-down;*
$f : \Sigma \times (I \cup \{\lambda\}) \times Z \longrightarrow \mathcal{P}(\Sigma \times Z^*)$ *este funcţia de evoluţie;*
$s_0 \in \Sigma$ *este starea iniţială;*
$z_0 \in Z$ *este simbolul iniţial al memoriei push-down.*

Un APD are în general o funcţioanre nedeterministă, $card\ f(s,i,z) \geq 1$; mulţimea automatelor push-down deterministe formează o clasă specială.

În termenii funcţiei de evoluţie, un pas de evoluţie comportă citirea simbolului i de pe bandă, citirea simbolului z din vârful memoriei push-down, apoi, în funcţie de starea internă s şi de aceste două simboluri, automatul trece într-o nouă stare s' şi scrie în memoria push-down un cuvânt $q \in Z^*$ astfel încât $(s', q) \in f(s, i, z)$. În cazul în care $f(s, i, z) = \emptyset$ evoluţia este oprită; este situaţia în care automatul se blochează.

O *stare* a automatului (sau configuraţie) este un sistem $\delta = (s, p, q)$ unde $s \in S$ este starea internă, $p \in I^*$ este subcuvântul de pe banda de intrare rămas de citit (inclusiv simbolul din dreptul dispozitivului de citire), iar $q \in Z^*$ este subcuvântul din memoria push-down.

Vom spune că un APD trece direct din starea $\delta_1 = (s_1, p_1, q_1)$ în starea $\delta_2 = (s_2, p_2, q_2)$ şi vom scrie $\delta_1 \longmapsto \delta_2$ dacă se execută un pas de evoluţie; dacă $p_1 = ip'_1, q_1 = zq'_1$ putem avea $(s_2, q) \in f(s_1, i, z)$ şi atunci $p_2 = p'_1, q_2 = qq'_1$ sau $(s_2, q) \in f(s_1, \lambda, z)$ şi atunci $p_2 = p_1, q_2 = qq'_1$.

Vom spune că automatul evoluează (fără specificaţia direct) din starea δ' în stare δ'' şi vom scrie $\delta' \overset{*}{\longmapsto} \delta''$ dacă:

(1) $\delta' = \delta''$;

(2) $\exists \delta_1, \ldots, \delta_n$ astfel încât $\delta' = \delta_1 \longmapsto \delta_2 \longmapsto \ldots \longmapsto \delta_{n-1} \longmapsto \delta_n = \delta''$.

Limbajul recunoscut de un APD se poate defini în două moduri:

(1) Limbajul recunoscut de un APD *cu golirea memoriei push-down*, este, prin definiţie

$$L(APD) = \{p | p \in I^*\ (s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, \lambda)\}.$$

Aceata înseamnă că, pornind din starea internă s_0 şi având în vârful memoriei push-down simbolul z_0, cu ajutorul cuvântului p de pe banda de intrare, automatul poate să evolueze astfel încât să golească memoria push-down după citirea cuvântului. Menţionăm că golirea memoriei push-down nu trebuie neaparat să coincidă cu citirea ultimului simbol al lui p; este posibil ca automatul să mai efectueze câţiva paşi citind de pe bandă simbolul λ.

(2) Limbajul recunoscut de un APD *cu stări finale*; în definiţia automatului se adaugă o submulţime Σ_f a lui Σ numită mulţimea de stări finale. Prin definiţie, limbajul recunoscut de un APD cu stări finale este:

$$L(APD) = \{p | p \in I^*\ (s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q), s \in \Sigma_f, q \in Z^*\}.$$

Prin urmare, este necesar ca după citirea lui p, eventual după încă câţiva paşi, APD să ajungă într-o stare finală. Vom vedea că cele două definiţii sunt echivalente.

Limbaje recunoscute de automate push-down cu golirea memoriei. Vom arăta că familia limbajelor recunoscute de APD cu stări finale coincide cu familia limbajelor independente de context. În felul acesta, APD constituie mecanisme analitice de definire a limbajelor de tipul 2.

Teorema 3.11 *Un limbaj este independent de context dacă şi numai dacă este recunoscut de un automat push–down cu golirea memoriei push–down.*

Demonstraţie. Partea I $E \in \mathcal{L}_2 \Rightarrow E = L(APD)$.

Fie $G = (V_N, V_T, S, P)$ o gramatică de tipul 2 în forma normală Greibach care generează limbajul E. Construim un automat pushdown astfel:
$APD = (\{s\}, V_T, V_N, f, s, S)$, funcţia de evoluţie fiind definită de:

$$A \to ip \in P \Rightarrow (s, p) \in f(s, i, A),$$

altfel $\emptyset$.

Fie $p \in L(G),\ p = i_1 \dots i_n,\ S \overset{*}{\underset{G}{\Rightarrow}}\ p$. Această derivare trebuie să aibă forma (extrem stângă):

$$(A)\ \ S \Rightarrow i_1 X_1 u_1 \Rightarrow i_1 i_2 X_2 u_2 u_1 \Rightarrow i_1 i_2 i_3 X_3 u_3 u_2 u_1 \Rightarrow \dots \Rightarrow i_1 \dots i_n,$$

unde $u_1, u_2, u_3, \dots \in V_N^* = Z^*$.

Observaţie. Aparent, partea $u_s u_{s-1} \dots u_1$ se măreşte cu fiecare derivare directă. În realitate, unele din cuvintele u_j sunt vide, şi anume atunci când se aplică o regulă de forma $X \to i$; în particular, în ultimele derivări directe se aplică numai reguli de această formă.

Avem

$$\begin{aligned}
&S \to i_1 X_1 u_1 \Rightarrow (s, X_1 u_1) \in f(s, i_1, S),\\
&X_1 \to i_2 X_2 u_2 \Rightarrow (s, X_2 u_2) \in f(s, i_2, X_1),\\
&X_2 \to i_3 X_3 u_3 \Rightarrow (s, X_3 u_3) \in f(s, i_3, X_2),\\
&\dots .
\end{aligned}$$

Prin urmare automatul poate să aibă următoarea evoluţie:

$$(s, i_1 i_2 i_3 i_4 \dots i_n, S) \longmapsto (s, i_2 i_3 i_4 \dots i_n, X_1 u_1) \longmapsto$$

$$\longmapsto (s, i_3 i_4 \dots i_n, X_2 u_2 u_1) \longmapsto (s, i_4 \dots i_n, X_3 u_3 u_2 u_1) \longmapsto \dots .$$

Dacă comparăm această evoluţie cu derivarea (A) putem observa că pe banda de intrare avem la fiecare pas partea complementară a cuvântului (faţă de derivare) iar în memoria push-down se reproduce partea de neterminale din formele propoziţionale

ale derivării. Cum în derivare se ajunge la $i_1 \dots i_n$, în evoluţie se va ajunge la (s, λ, λ).

Deci $p \in L(APD)$ şi $L(G) \subseteq L(APD)$.

Fie acum $p \in L(APD)$; va trebui să arătăm că $p \in L(G)$, deci că $S \underset{G}{\overset{*}{\Rightarrow}} p$. Vom arăta o implicaţie ceva mai generală, şi anume, pentru orice $u \in V_N^*$, avem

$$(s, p, u) \overset{*}{\longmapsto} (s, \lambda, \lambda) \Rightarrow u \underset{G}{\overset{*}{\Rightarrow}} p.$$

În particular, dacă $u = S$ obţinem implicaţia dorită.

Procedăm prin inducţie asupra lungimii lui p.

Dacă $|p| = 1$, atunci $p = i, u = X$ iar evoluţia va avea un singur pas $(s, i, X) \longmapsto (s, \lambda, \lambda)$, deci $(s, \lambda) \in f(s, i, X)$ şi $X \to i \in P$. Putem scrie $u = X \underset{G}{\Rightarrow} i = p$.

Presupunem că implicaţia este adevărată pentru un cuvânt $|p| = l$ şi considerăm un p astfel încât $|p| = l + 1$. Fie i şi X primele simboluri din p şi u, deci $p = ip'$ şi $u = Xu'$. În evoluţia $(s, p, u) \overset{*}{\longmapsto} (s, \lambda, \lambda)$ punem în evidenţă prima evoluţie directă

$$(s, p, u) = (s, ip', Xu') \longmapsto (s, p', vu') \overset{*}{\longmapsto} (s, \lambda, \lambda).$$

Din definiţia evoluţiei directe rezultă că $(s, v) \in f(s, i, X)$ deci $X \to iv \in P$. Pe de altă parte din ipoteza inductivă rezultă că $vu' \underset{G}{\overset{*}{\Rightarrow}} p'$. Avem

$$u = Xu' \underset{G}{\Rightarrow} ivu' \underset{G}{\overset{*}{\Rightarrow}} ip' = p,$$

ceea ce demonstrează implicaţia.

Prin urmare $p \in L(G)$ şi $L(APD) \subseteq L(G)$, de unde $L(G) = L(APD)$.□

Partea II. $E = L(APD)$ $E \in \mathcal{L}_2$

Fie $APD = (\Sigma, I, Z, f, s_0, z_0)$. Construim G de forma $G = (V_N, V_T, S, P)$ unde $V_N = \{s_0\} \cup \{(s, z, s') | s, s' \in \Sigma, z \in Z\}$, $V_T = I$, $S = s_0$, iar regulile de generare le definim astfel:

(1) $s_0 \to (s_0, z_0, s)$, $\forall s \in \Sigma$;

(2) Dacă $(s_1, z_1 \dots z_m) \in f(s, i, z)$ vom introduce în P reguli de forma

$$(s, z, s') \to i(s_1, z_1, s_2)(s_2, z_2, s_3) \dots (s_m, z_m, s'),$$

unde $s', s_2, \dots, s_m \in \Sigma$;

(3) Dacă $(s', \lambda) \in f(s, i, z)$ vom introduce în P reguli de forma

$$(s, z, s') \to i,$$

unde $s' \in \Sigma$.

Să observăm că gramatica astfel construită este independentă de context, şi anume în forma normală Greibach.

Fie $p \in L(APD)$, deci $(s_0, p, z_0) \overset{*}{\longmapsto} (s', \lambda, \lambda)$; trebuie să arătăm că $s_0 \underset{G}{\overset{*}{\Rightarrow}} p$.

Vom arăta implicaţia ceva mai generală

$$(s, p, z) \overset{*}{\longmapsto} (s', \lambda, \lambda) \quad \Rightarrow \quad (s, z, s') \underset{G}{\overset{*}{\Rightarrow}} p.$$

În particular pentru $s = s_0, z = z_0$ rezultă $(s_0, z_0, s') \underset{G}{\overset{*}{\Rightarrow}} p$ şi putem scrie $s_0 \Rightarrow (s_0, z_0, s') \underset{G}{\overset{*}{\Rightarrow}} p$, adică $p \in L(G)$.

Procedăm prin inducţie asupra lungimii evoluţiei l.

Dacă $l = 1$ atunci $(s, p, z) \longmapsto (s', \lambda, \lambda)$, deci $p = i$ şi $(s', \lambda) \in f(s, i, z)$ şi $(s, z, s') \to i$ este o regulă, adică putem scrie $(s, z, s') \Rightarrow i = p$.

Presupunem că implicaţia este adevărată pentru evoluţii de lungime oarecare l şi considerăm o evoluţie de lungime $l + 1$; punem în evidenţă prima evoluţie directă

$$(s, p, z) = (s, i_1 p', z) \longmapsto (s_1, p', z_1 \dots z_m) \overset{*}{\longmapsto} (s', \lambda, \lambda).$$

Descompunem cuvântul p' în forma $p' = p_1 \dots p_m$ astfel încât

$$\begin{array}{lll} (s_1, p_1, z_1) & \overset{*}{\longmapsto} & (s_2, \lambda, \lambda), \\ (s_2, p_2, z_2) & \overset{*}{\longmapsto} & (s_3, \lambda, \lambda), \\ & \dots & \\ (s_m, p_m, z_m) & \overset{*}{\longmapsto} & (s', \lambda, \lambda). \end{array}$$

Observaţie. Putem pune în evidenţă felul în care se defineşte cuvântul p_1 urmărind evoluţia lui APD;

$$\begin{array}{ccccc} (s_1, i_1 i_2 \dots i_n, z_1 z_2 \dots z_m) & \longmapsto & (s_1', i_2 i_3 \dots i_n, q z_2 \dots z_m) & \longmapsto & \\ (a) & & (b) & & \\ \dots & \longmapsto & (s_2, i_{j_1} \dots i_n, z_2 \dots z_m) & & \\ & & (c) & & \end{array}$$

La primul pas (situaţia a) automatul este în starea s_1, pe bandă este i_1 iar în memoria push-down este z_1. După efectuarea unui pas, automatul trece în starea s_1', mută banda cu o poziţie spre stânga, extrage pe z_1 şi scrie în memoria push-down un cuvânt q (situaţia b). Se poate observa că z_2 a "coborât"; cum ştim că memoria push-down se goleşte ($p \in L(APD)$), trebuie ca la un moment dat z_2 să ajungă în vârful stivei (situaţia c). În acest moment partea din p citită va fi p_1 iar starea în care a ajuns automatul o notăm cu s_2. Este clar că dacă pe bandă am avea scris numai p_1 am avea evoluţia $(s_1, p_1, z_1) \longmapsto (s_2, \lambda, \lambda)$.

Analog $p_2, \ldots, p_m$.

Din definiţia derivării directe $(s, i_1p', z) \longmapsto (s_1, p', z_1 \ldots z_m)$ avem $(s_1, z_1 \ldots z_m) \in f(s, i_1, z)$ iar în P va exista regula

$$(s, z, s') \to i_1(s_1, z_1, s_2)(s_2, z_2, s_3) \ldots (s_m, z_m, s')$$

unde luăm stările $s_2, \ldots, s_m$ cele rezultate la descompunerea lui p'. Pe de altă parte, din ipoteza inductivă, avem

$$\begin{aligned}
&(s_1, z_1, s_2) \overset{*}{\Rightarrow} p_1,\\
&(s_2, z_2, s_3) \overset{*}{\Rightarrow} p_2,\\
&\ldots\\
&(s_m, z_m, s') \overset{*}{\Rightarrow} p_m.
\end{aligned}$$

Putem scrie derivarea

$$(s, z, s') \Rightarrow i_1(s_1, z_1, s_2)(s_2, z_2, s_3) \ldots (s_m, z_m, s') \overset{*}{\Rightarrow} i_1p_1 \ldots p_m = i_1p' = p.$$

După cum am văzut, rezultă mai departe $p \in L(G)$ şi $L(APD) \subseteq L(G)$.

Pentru a demonstra incluziunea inversă, vom arăta mai întâi implicaţia $(s, z, s') \overset{*}{\Rightarrow} p(s_1, z_1s_2) \ldots (s_m, z_m, s')$ implică $(s, p, z) \overset{*}{\longmapsto} (s_1, \lambda, z_1 \ldots z_m)$.

Procedăm prin inducţie asupra lungimii derivării l.

Dacă $l = 1$ atunci $p = i$ şi se aplică regula

$$(s, z, s') \to i(s_1, z_1, s_2) \ldots (s_m, z_m, s')$$

deci $(s_1, z_1 \ldots z_m) \in f(s, i, z)$ şi $(s, i, z) \longmapsto (s_1, \lambda, z_1 \ldots z_m)$.

Presupunem că implicaţia este adevărată pentru l oarecare şi considerăm o derivare de lungime $l + 1$. Fie $p = p'i$ şi punem în evidenţă ultimul pas.

$(s, z, s') \overset{*}{\Rightarrow} p'(s'_{j-1}, z'_{j-1}, s'_j)(s_j, z_j, s_{j+1}) \ldots (s_m, z_m, s')$

$\Rightarrow p'i(s_1, z_1, s_2) \ldots (s_{j-1}, z_{j-1}, s_j)(s_j, z_j, s_{j+1}) \ldots (s_m, z_m, s')$,

unde $s'_j = s_j$; la ultimul pas s-a aplicat regula

$$(s'_{j-1}, z'_{j-1}, s_j) \to i(s_1, z_1, s_2) \ldots (s_{j-1}, z_{j-1}, s_j).$$

Rezultă $(s_1, z_1 \ldots z_{j-1}) \in f(s'_{j-1}, i, z'_{j-1})$ şi putem scrie evoluţia $(s'_{j-1}, i, z'_{j-1}) \longmapsto (s_1, \lambda, z_1 \ldots z_{j-1})$.

Pe de altă parte, conform ipotezei inductive, avem

$$(s, p', z) \overset{*}{\longmapsto} (s'_{j-1}, \lambda, z'_{j-1}z_j \ldots z_m)$$

Prin urmare

$$(s, p, z) = (s, p'i, z) \overset{*}{\longmapsto} (s'_{j-1}, i, z'_{j-1}z_j \ldots z_m) \longmapsto (s_1, \lambda, z_1 \ldots z_m)$$

şi implicaţia este demonstrată.

Fie acum $p \in L(G)$, deci $s_0 \overset{*}{\underset{G}{\Rightarrow}} p$. Ţinând seama de forma regulilor din G, în această derivare se va aplica prima dată o regulă de forma (1), apoi regula de forma (2) iar la şfârşit reguli de forma (3). La aplicarea regulilor (2) putem rescrie la fiecare pas simbolul neterminal cel mai din stânga, deci să obţinem o derivare extrem stângă. Să observăm că în acest caz structura formelor propoziţionale intermediare este cea menţionată, $p(s_1, z_1, s_2)(s_2, z_2, s_3) \dots (s_m, z_m, s')$.

Prin urmare, derivarea va avea forma

$$s_0 \Rightarrow (s_0, z_0, s') \overset{*}{\Rightarrow} p(s_1, z_1, s_2) \dots (s_m, z_m, s') \overset{*}{\Rightarrow} p.$$

Trebuie să avem regulile $(s_j, z_j, s_{j+1}) \to \lambda, j = 1, \dots, m,\ s_{m+1} = s'$ şi putem scrie

$$(s_0, p, z_0) \overset{*}{\longmapsto} (s_1, \lambda, z_1 \dots z_m) \longmapsto (s_2, \lambda, z_2 \dots z_m) \longmapsto \dots \longmapsto (s', \lambda, \lambda)$$

adică $p \in L(APD)$ şi $L(G) \subseteq L(APD)$.□

Automate push–down cu stări finale. Vom nota un automat push-down cu stări finale cu APD_f.

Teorema 3.12 *Un limbaj este recunoscut de un automat push–down dacă şi numai dacă este recunoscut de un automat push–down cu stări finale.*

Demonstraţie. Partea I $E = l(APD) \Rightarrow E \in L(APD_f)$.

Dacă $APD = (\Sigma, I, Z, f, s_0, z_0)$ construim un automat push–down cu stări finale astfel

$$APD_f = (\Sigma \cup \{s_0', s_f\}, I, Z \cup \{z_0'\}, f', s_0', z_0')$$

unde mulţimea de stări finale este $\{s\}$ iar funcţia de evoluţie este definită de:

$f'(s, i, z) = f(s, i, z), s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;

$f'(s_0', \lambda, z_0') = (s_0, z_0 z_0')$;

$f(s, \lambda, z_0') = (s_f, \lambda), s \in \Sigma$;

în rest $\emptyset$.

Fie $p \in L(APD)$; atunci $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, \lambda)$. Evident că aceiaşi evoluţie o poate avea şi automatul push–down cu stări finale. Putem scrie în APD_f evoluţia

$$(APD_f): \quad (s_0', p, z_0') \overset{*}{\longmapsto} (s_0, \lambda, z_0') \longmapsto (s, \lambda, \lambda),$$

deci $p \in L(APD_f)$ şi $L(APD) \subseteq L(APD_f)$.

Invers, fie $p \in L(APD_f)$, atunci (în APD_f)

$$(s_0', p, z_0') \longmapsto (s, p, z_0 z_0') \overset{*}{\longmapsto} (s_f, \lambda, q).$$

Ultimul pas trebuie să fie de forma $(s, \lambda, z_0') \longmapsto (s_f, \lambda, \lambda)$ pentru că nu există altă valoare a lui f care să ne ducă într-o stare finală. Deci (în APD_f)

$$(s_0', p, z_0 z_0') \overset{*}{\longmapsto} (s, \lambda, z_0') \longmapsto (s_f, \lambda, \lambda)$$

şi putem scrie în APD evoluţia $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, \lambda)$, adică $p \in L(APD)$ şi $L(APD_f) \subseteq L(APD)$.□

Partea II $E = L(APD_f \Rightarrow E \in L(APD)$.

Fie $APD_f = (\Sigma, I, Z, f, s_0, z_0, \Sigma_f)$ un automat push–down cu stări finale (mulţimea stărilor finale este Σ_f) şi construim un APD astfel

$$APD = (\Sigma \cup \{s_0', s'\}, I, Z \cup \{z_0'\}, f', s_0', z_0')$$

unde

$f'(s, i, z) = f(s, i, z), s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;

$f(s_0', \lambda, z_0') = (s, z_0 z_0')$;

$f(s, \lambda, z) = (s', \lambda), s \in \Sigma_f \cup \{s'\}, z \in Z \cup \{z_0'\}$;

în rest $\emptyset$.

Fie $p \in L(APD_f)$, atunci $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q), s \in \Sigma_f$. Este evident că în APD avem evoluţia $(s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q)$. Putem scrie

$$APD: \quad (s_0', p, z_0') \longmapsto (s_0, p, z_0 z_0') \overset{*}{\longmapsto} (s, \lambda, q z_0') \overset{*}{\longmapsto} (s', \lambda, \lambda),$$

deci $p \in L(APD)$ şi $L(APD_f) \subseteq L(APD)$.

Invers, fie $p \in L(APD)$. Avem

$$APD: \quad (s_0', p, z_0') \overset{*}{\longmapsto} (s_0, p, z_0 z_0') \overset{*}{\longmapsto} (s, \lambda, \lambda).$$

Simbolul z_0' nu poate fi şters decât cu o regulă de forma $f(s, \lambda, z) = (s', \lambda)$, $s \in \Sigma_f \cup \{s'\}$, deci APD trebuie să ajungă într-o stare $s \in \Sigma_f$, apoi să rămână în s'.

$$APD: \quad (s_0, p, z_0 z_0') \overset{*}{\longmapsto} (s, \lambda, q z_0'), s \in \Sigma_f.$$

Putem scrie

$$APD_f: \quad (s_0, p, z_0) \overset{*}{\longmapsto} (s, \lambda, q), s \in \Sigma_f$$

şi deci $p \in L(APD_f)$, adică $L(APD) \subseteq L(APD_f)$.□

3.6 Automate push–down deterministe

Funcţionarea unui APD este în general nedeterministă, $card\ f(s, i, z) \geq 1$. Pentru ca un APD să aibă o funcţionare deterministă nu ete suficient să impunem condiţia $card\ f(s, i, z) = 1$, deoarece dacă pentru un anumit $s \in \Sigma$ şi $z \in Z$ avem $f(s, \lambda, z) \neq \emptyset$ şi $f(s, i, z) \neq \emptyset$, putem face un pas citind λ sau citind i.

Definiţie 3.10 *Un automat push–down este determinist dacă*

(1) $card\ f(s, i, z) \geq 1, s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;

(2) dacă $f(s, \lambda, z) \neq \emptyset$, *atunci* $f(s, i, z) = \emptyset, \forall i \in I$.

Un limbaj recunoscut de un APD determinist îl vom numi limbaj independent de context (sau de tipul doi) determinist. Familia limbajelor independente de context deterministe este inclusă (strict) în familia limbajelor de tipul 2 (după cum vom vedea).

Un APD se poate bloca în următoarel două situaţii

1. Automatul ajunge în starea s, în vârful memoriei push–down se află simbolul z, pe banda de intrare urmează i şi $f(s,i,z) = f(s,\lambda,z) = \emptyset$;

2. Intră într-un ciclu infinit citind de pe bandă λ; de exemplu $f(s,\lambda,z) = (s,z)$ şi $f(s,i,z) = \emptyset$ pentru o anumită pereche (s,z).

Definiţie 3.11 *Un APD determinist este* neblocabil *dacă pentru orice cuvânt $p \in I^*$ există o evoluţie de forma $(s_0,p,z_0) \overset{*}{\longmapsto} (s,\lambda,q)$.*

Într-un APD determinist neblocabil orice cuvânt peste I poate fi citit. Evident, de aici nu rezultă că orice cuvânt este recunoscut de APD.

Lema 3.6 *Un APD determinist cu stări finale este echivalent cu un APD determinist cu stări finale neblocabil (relativ la prima situaţie de blocare).*

*Demonstraţie.*Fie $APD_f = (\Sigma, I, Z, f, s_0, z_0, \Sigma_f)$. Construim $APD'_f = (\Sigma \cup \{s'_0, s'\}, I, Z \cup \{z'_0\}, f', s'_0, z'_0, \Sigma_f)$ unde:
(1) $f'(s,i,z) = f(s,i,z)$ dacă $f(s,i,z) \neq \emptyset, s \in \Sigma, i \in I \cup \{\lambda\}, z \in Z$;
(2) $f'(s,i,z) = (s',z)$ dacă $f(s,i,z) = f(s,\lambda,z) = \emptyset, s \in \Sigma, i \in I, z \in Z$;
(3) $f'(s',i,z) = (s',z),\ i \in I, z \in Z$;
(4) $f'(s',\lambda,z'_0) = (s_0, z_0 z'_0)$.
Avem

$$p \in L(APD_f) \Leftrightarrow (s_0,p,z_0) \overset{APD_f}{\models} (s,\lambda,q), s \in \Sigma_f \Leftrightarrow$$

$$(s'_0,p,z'_0) \overset{APD'_f}{\models} (s_0,p,z_0z'_0) \overset{APD'_f}{\models} (s,\lambda,qz'_0), s \in \Sigma_f \Leftrightarrow p \in L(APD'_f).$$

Deci $L(APD_f) = L(APD'_f$.□

Observaţie. Într-o situaţie de blocare (s,ip,zq) şi $f(s,i,z) = f(s,\lambda,z) = \emptyset$ putem scrie

$$(s,ip,zq) \overset{APD'_f}{\models} (s',p,zq) \overset{APD'_f}{\models} \ldots \overset{APD'_f}{\models} (s',\lambda,zq).$$

Teorema 3.13 *Orice limbaj independent de context determinist este recunoscut de un APD determinist neblocabil.*

Demonstraţie. Fiind dat un APD determinist vom construi un APD determinist neblocabil echivalent. Conform lemei anterioare putem presupune că nu are loc prima situaţie de blocare.

Dacă are loc a doua situaţie de blocare, putem avea două cazuri:

1. conţinutul memoriei push–down se măreşte nelimitat;

2. lungimea cuvintelor scrise în memoria push–down nu depăşeşte un anumit număr.

Fie $card(\Sigma) = n$, $card(Z) = k$, $l = max\{|q|, q \in Z^* \mid (s', q) \in f(s, i, z)\}$.

Cazul 1. Există în total un număr nk de perechi de forma (s, z). Dacă în evoluţia lui APD s-ar succede numai configuraţii cu perechi de forma (s, z) distincte atunci lungimea cuvântului din memoria push–down ar creşte la maximum nkl simboluri. Prin urmare, dacă $(s', \lambda, \alpha') \vdash^* (s", \lambda, \alpha")$ şi $|\alpha"| - |\alpha'| > nkl$, atunci în această evoluţie trebuie să existe două configuraţii cu aceiaşi stare s şi cu acelaşi simbol z în vârful memoriei push–down. Deci

$$(s', \lambda, \alpha') \vdash^* (s, \lambda, zr) \vdash^* (s, \lambda, zqr) \vdash^* (s", \lambda, \alpha"),$$

de unde urmează că

$$(s', \lambda, \alpha') \vdash^* (s, \lambda, zr) \vdash^* (s, \lambda, q^m r), \forall m \in N.$$

Cazul II Lungimea cuvântului scris în memoria push–down nu depăşeşte nkl, căci dacă $|\alpha"| - |\alpha'| > nkl$, ar rezulta că în memoria push–down am avea $zq^m, \forall m \in N$ şi lungimea cuvântului nu ar fi finită.□

Teorema 3.14 *Dacă E este un limbaj independent de context determinist atunci limbajul complementar $I^* \setminus E$ este de asemenea independent de context determinist.*

Demonstraţie. Fie $APD_f = (\Sigma, I, Z, f, s_0, z_0, \Sigma_f)$ automatul push–down determinist care recunoaşte limbajul E. Construim un automat push–down determinist care va recunoaşte limbajul $I^* \setminus E$ în modul următor

$$APD'_f = (\Sigma \times \{\sigma_1, \sigma_2, \sigma_3\}, I, Z, f', s'_0, z_0, \Sigma'_f)$$

unde

$$s'_0 = \begin{cases} (s_0, \sigma_1) & pentru\ s_0 \in \Sigma_f, \\ (s_0, \sigma_2) & pentru\ s_0 \notin \Sigma_f, \end{cases}$$

mulţimea de stări finale este $\Sigma'_f = \{(s, \sigma_3) | s \in \Sigma\}$ iar funcţia de evoluţie este definită de

(1) dacă $f(s,i,z)=(s',q)$ atunci
$$f'((s,\sigma_1),i,z)=((s',k'),q),$$
$$f'((s,\sigma_2),i,z)=((s,\sigma_3),q),$$
$$f'((s,\sigma_3),i,z)=((s',k'),q);$$
(2) dacă $f(s,\lambda,z)=(s',q)$ atunci
$$f'((s,\sigma_1),\lambda,z)=((s',k'),q),$$
$$f'((s,\sigma_3),\lambda,z)=((s',k'),q);$$

unde $k'=1$ dacă $s'\in\Sigma_f$ şi $k'=2$ dacă $s'\notin\Sigma_f$.

Fie $p\in L(APD'_f)$, atunci $((s_0,\sigma_k),p,z_0)\overset{*}{\longmapsto}((s,\sigma_3),\lambda,q), k=1,2$. În mod necesar, ultima configuraţie trebuie să fie precedată de o configuraţie de forma $((s',\sigma_2),\lambda,q)$, deci

$$(A)\ ((s_0,\sigma_k),p,z_0)\overset{*}{\longmapsto}((s',\sigma_2),\lambda,q)\longmapsto((s,\sigma_3),\lambda,q)$$

de unde rezultă că $(s_0,p,z_0)\overset{*}{\longmapsto}(s',\lambda,q)$ şi $s'\notin\Sigma_f$. Deci $p\in I^*\setminus E$ şi $L(APD'_f)\subseteq I^*\setminus E$.

Invers, fie $p\in I^*\setminus E$, atunci în APD'_f avem evoluţia $(s_0,p,z_0)\overset{*}{\longmapsto}(s,\lambda,q), s\notin\Sigma_f$ şi putem scrie evoluţia (A). Prin urmare $p\in L(APD'_f)$, adică $I^*\setminus E\subseteq L(APD'_f)$.□

Consecinţă. Putem enunţa proprietatea de mai sus astfel: Familia limbajelor independente de context deterministe este închisă la complementariere. Cum familia limbajelor independente de context nu este închisă la complementariere şi cum ea coincide cu familia limbajelor recunoscute de automatele push–down nedeterministe putem mai departe obţine următorul rezultat:

APD nedeterministe nu sunt echivalente cu APD deterministe.

3.7 Probleme propuse

1. Să se arate că $L(G)=\{(ab)^na|n\geq 0\}$ unde G are producţiile $S\to SbS,\ S\to a$. Să se construiască o gramatică echivalentă cu G care să fie neambiguă.

2. Eliminaţi redenumirile din gramatica G_E ce generează expresiile aritmetice simple.

3. Aduceţi la forma normală Chomsky gramaticile ce au regulile:

   (a) $S\to TbT, T\to TaT|ca$;

   (b) $S\to aAC, A\to aB|bAB, B\to b, C\to c$;

   (c) $S\to A.A, A\to 0A|1A|\ldots 9A|\lambda$.

4. Găsiţi forma normală Greibach pentru gramaticile:

(a) $A_1 \to A_2A_3,\ A_2 \to A_1A_2|1,\ A_3 \to A_1A_3|0$.

(b) G_E care generează expresiile aritmetice simple.

5. Construiţi un automat push–down pentru recunoaşterea limbajului:

(a) $L = \{w|w \in \{a,b\}^*$, numărul literelor a în w este egal cu numărul literelor b în w $\}$;

(b) $L = \{w|w \in \{a,b\}^*, w = \tilde{w}\}$;

(c) $L = \{w|w \in \{(,)\}^*, w$ este un cuvânt în care fiecare paranteză deschisă are o pereche, paranteză închisă $\}$.

6. Folosind lema de pompare să se arate că următoarele limbaje nu sunt independente de context:

(a) $L = \{a^ib^jc^k|i < j < k\}$;

(b) $L = \{a^ib^j|j = i^2\}$;

(c) $L = \{a^nb^nc^n|n \geq 1\}$;

(d) $L = \{a^i|i$ prim $\}$;

(e) $L = \{a^ib^ic^j|j \geq i\}$;

Capitolul 4

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 supe- rioară. 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).

4.1 Algoritmi TOP-DOWN

4.1.1 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 cartea lui D. Gries, *Compiler construction for digital compters.*

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:

$$x_0 \to x_{11} \dots x_{1n_1} | x_{21} \dots x_{2n_2} | \dots | x_{p1} \dots x_{pn_p}$$

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 $x_{1j} \overset{*}{\Rightarrow} p_j,\ j = 1, \dots n_1$.

Simbolul x_0 îl activează pe x_{11} şi îi cere să găsească derivarea $x_{11} \overset{*}{\Rightarrow} p_1$; 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 $x_{12} \overset{*}{\Rightarrow} p_2$, 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 $x_{1j} \overset{*}{\Rightarrow} p_j$, 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 $x_{13} \stackrel{*}{\Rightarrow} i_7 i_8 \ldots$. 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 4.1.

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).

4.1.2 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**.

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

Să considerăm urmatorul cuvânt de analizat

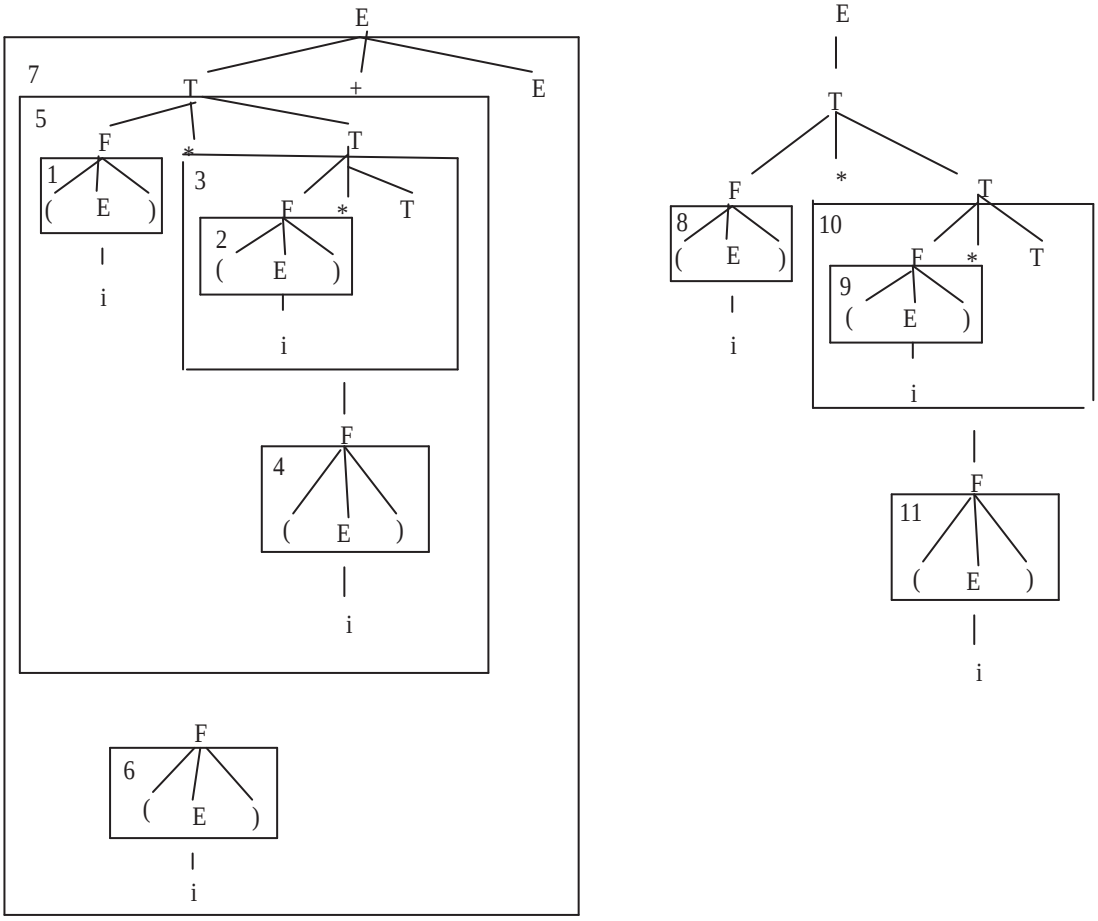

Figura 4.1: Arborele sintactic pentru $p = i * i$

```
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 $D \rightarrow d; X$. Din cuvântul iniţial trebuie generată în continuare secvenţa $d; i; i; i; EndProgram$ 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 $X \rightarrow \lambda$, ş.a.m.d.

Dacă se consideră gramatica ce generează expresii aritmetice simple, în forma considerată la algoritmul general top-down (4.2.9), 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:

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

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

Figura 4.2: Gramatica pentru generarea blocurilor de instrucţiuni.

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

Figura 4.3: Program sursa de analizat sintactic

4.1.3 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 4.2) 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 4.3.

Lista de unităti 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.

Un program pentru analiza sintactică top-down fără reveniri corespunzător gramaticii din figura 4.2 este reprezentat parţial în figura 4.4. 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 4.3 va produce o secvenţă de apeluri recursive ca în figura 4.5.

Un pic de algebră

Vom da în cele ce urmează o definiţie riguroasă a gramaticilor $LL(k)$ (k simboluri citite în avans) vom da în cele ce urmează , 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

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

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

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:

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

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.

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

```
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;
 }
};
...
```

Figura 4.4: Cod C corespunzător analizorului sintactic

```
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 4.5: Execuţia analizei lexicale pentru textul sursa

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 }
$Prim(a) = \{a\}; \forall a \in V_T$
$Prim(X) = Urm(X) = \Phi; \forall X \in V_N$
Pas 2: **Repeat**
Pentru fiecare regulă $X \to Y_1 Y_2 \dots Y_k$ **Do**
For$(i = 1; i \leq k; i++)$
(2.1)**if** $(i = 1$ *sau* $Y_1 Y_2 \dots Y_i \in NULL^+)$
$Prim(X) = Prim(X) \cup Prim(Y_i)$;
(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

$$Prim(X\alpha) = \begin{cases} Prim(X) & dac'a\ X \notin NULL(G) \\ Prim(X) \cup Prim(\alpha) & dac'aX \in NULL(G) \end{cases}$$

Dacă se consideră gramatica ce generează expresii aritmetice 4.1.2, 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 simbolurilor 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.

4.2 Algoritmi BOTTOM-UP

4.2.1 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 4.2.9 şi forma propoziţională $p = T * F + a * (E + T)$. Arborele de derivare este prezentat în figura 4.6. Se poate observa că frazele simple sunt $T * F, a, E + T$.

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 4.7):

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

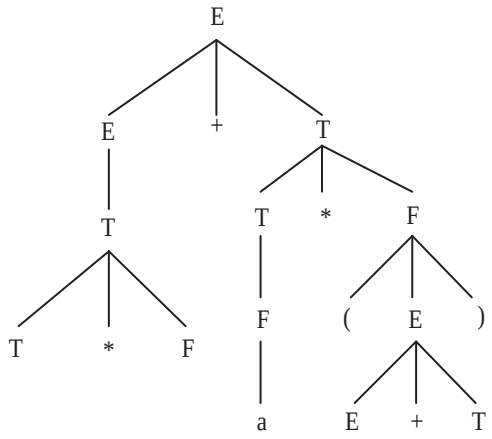

Figura 4.6: Arborele de derivare corespunzător derivării lui $p = T*F+a*(E+T)$

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

Figura 4.7: Algoritmul bottom-up

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.

4.2.2 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		$\pm$	$\succ$	$\prec$

4.2.3 Proprietăţi ale gramaticilor cu precedenţă simplă

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

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

Demonstraţia acestei proprietăţi se efectuează considerând toate structurile de arbori de derivare posibile într-o situaţie dată. Vom exemplifica această idee pentru cuvânul $p = a_1 \dots a_7$. 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 4.8.

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 $f_p = a_l \dots a_k$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 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

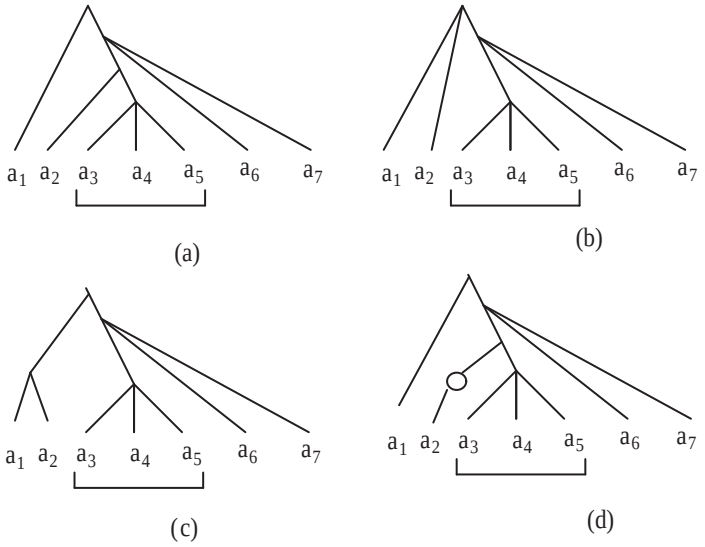

Figura 4.8: Arbori de derivare posibili

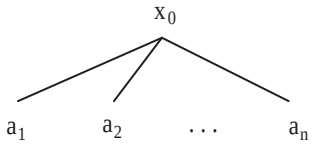

Figura 4.9: Arborele de derivare pentru $ni = 1$

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 4.9 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, $p = rf_ps$. 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.

4.2.4 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 4.10.

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 4.10. 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 $x \notin f_p$, $y \in f_p$.

4.2.5 Studiu de caz

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

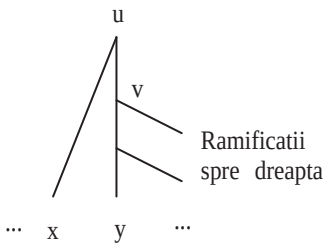

Figura 4.10: Arborele de derivare corespunzator cazului $x \prec y$

	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 4.11: Matricea de precedenţă pentru G_E

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

Relaţiile First, Last şi închiderile acestora sunt:

$$\begin{array}{lll} E\ F\ \{E, T\} & E\ F^+\ \{E, T, F, (, a\} & E\ F^*\ \{E, T, F, (, a\} \\ T\ F\ \{T, F\} & T\ F^+\ \{T, F, (, a\} & T\ F^*\ \{T, F, (, a\} \\ F\ F\ \{(, a\} & F\ F^+\ \{(, a\} & F\ F^*\ \{F, (, a\} \end{array}$$

$$\begin{array}{lll} E\ L\ \{T\} & E\ L^+\ \{T, F, a,)\} & E\ F^*\ \{E, T, F, a,)\} \\ T\ L\ \{F\} & T\ L^+\ \{F, a,)\} & T\ F^*\ \{T, F, a,)\} \\ F\ L\ \{a,)\} & F\ L^+\ \{a,)\} & F\ F^*\ \{F, a,)\} \end{array}$$

Conform teoremei de calcul a relaţiilor de precedenţă vom obţine matricea de precedenţă (vezi figura 4.11) asociată simbolurilor gramaticii G_E.

$$G:\begin{cases} A \to A0 \mid B1 \\ B \to 1 \end{cases}$$

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

Figura 4.12: 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 4.13: Reconstrucţia derivării cuvântului $p = aaabaababbbabb$

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 4.12, matricea de precedenţă asociată este cea alăturată.

Reconstituirea derivării cuvântului $p = aaabaababbbabb$ este prezentată în tabelul 4.13, unde fiecare linie corespunde formei propoziţionale, în care fraza simplă stângă este subliniată, precedată de regula identificată.

4.2.6 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, $i = i + 1$;

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 4.14.

Exemplu. Tabelul 4.15 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 mai mare. Acest fapt este ilustat în figura 4.16.

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.

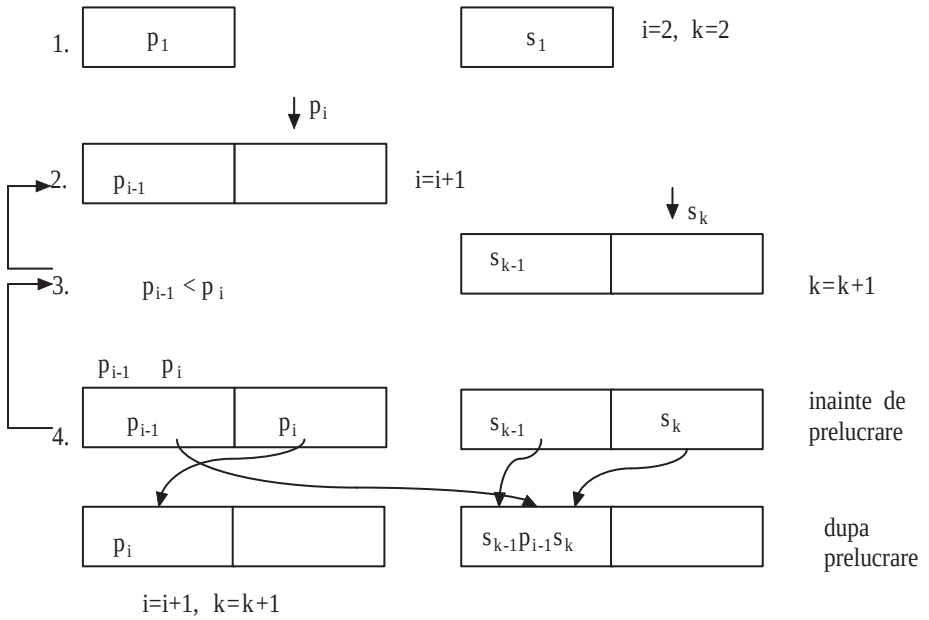

Figura 4.14: Prelucrările efectuate asupra celor două stive

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

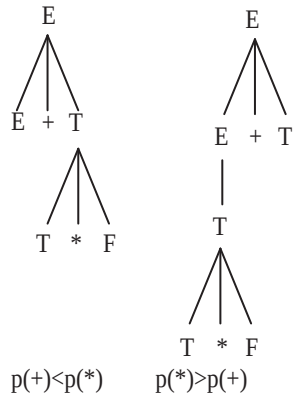

Figura 4.16: Ierarhia operatorilor

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.

4.2.7 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 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:

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

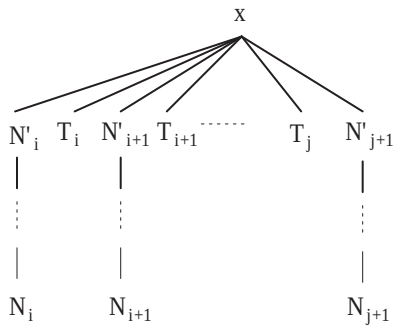

Figura 4.17: Structura subarborelui

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, $\ldots((a\ldots.$

Definiţia 2. Fie $p = N_1T_1N_2T_2\ldots T_nN_{n+1}$ o formă propoziţională într-o gramatică operatorială. Vom spune că f este o *frază simplă* a lui p dacă $f = N_iT_iN_{i+1}T_{i+1}\ldots T_jN_{j+1}$ este un subcuvânt al lui p care conţine *cel puţin un simbol terminal* şi în plus:

(1) $\exists x \to N_i'T_iN_{i+1}'T_{i+1}\ldots T_jN_{j+1}' \in \mathcal{P}$, şi $N_k' \stackrel{*}{\Rightarrow} N_k\ \forall k = i, \ldots, j$.

(2) $\mathcal{A}_{x,f} \subset \mathcal{A}_{x_0,p}$.

Structura de arbore corespunzătoare unei fraze simple este prezentată în figura 4.17.

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 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.

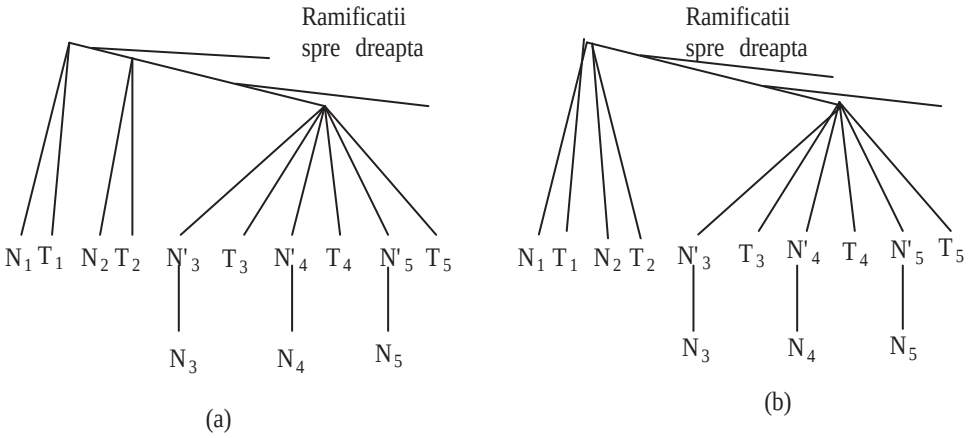

Figura 4.18: Structurile posibile de arbori

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

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

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 4.18).

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 $f_p = N_kT_kN_{k+1}T_{k+1} \dots T_lN_{l+1}, l \neq j, k \neq i$ 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 4.19 şi unicitatea acestuia este evidentă. Presupunem acum că proprietatea

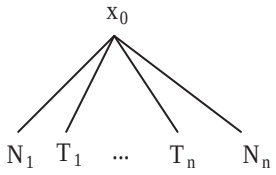

Figura 4.19: Arborele de derivare pentru $ni = 1$

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, $p = rf_ps$. 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.

4.2.8 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

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

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 4.20.

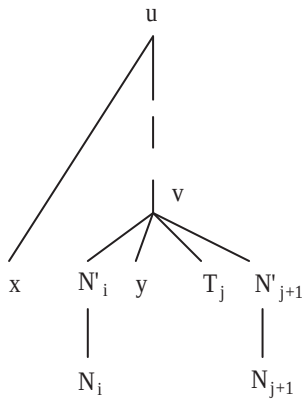

Figura 4.20: Structura posibilă de arbore

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.

4.2.9 Studiu de caz

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

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

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

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

	$+$	$*$	$($	$)$	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 4.21: Matricea de precedenţă operatorială pentru G_E

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

Figura 4.22: Reconstrucţia derivării cuvântului $p = a + (a + a) * a * a$

Conform teoremei de calcul a relaţiilor de precedenţă vom obţine matricea de precedenţă (vezi figura 4.21) 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.

Reconstituirea derivării cuvântului $p = a + (a + a) * a * a$ este prezentată în tabelul 4.22, unde fiecare linie corespunde formei propoziţionale, în care fraza simplă stângă este subliniată, precedată de regula identificată.

Capitolul 5

Sinteza Programelor

5.1 Forme interne ale programelor

5.1.1 Tabelele compilatorului

Compilatoarele prelucrează un număr important de informaţii, variabile de diverse tipuri, date organizate în anumite structuri standard sau definite de utilizator, etc. De cele mai multe ori aceste informaţii sunt stocate în tabele şi memorate pe suporturile de informaţii ale sistemului. Organizarea şi prelucrarea tabelelor constituie o problemă distinctă, cu infuenţă importantă asupra performanţelor unui compilator, mai ales dacă acestea sunt memorate pe un suport extern. În cele ce urmează vom prezenta pe scurt principalele tabele utilizate în mod obişnuit în cadrul unui compilator şi modul de organizare a informaţiilor stocate în aceste tabele. Menţionăm că structura acestor tabele este definită de proiectantul de compilator şi că prin urmare există un anumit grad de subiectivitate în modul de organizare al acestora.

- *Tabelul de variabile.* Conţine variabilele utilizate într-un program şi informaţii asupra lor. Menţionăm că variabilele conţin datele prelucrate de program, în conformitate cu tipul acestora, conţinut care se modifică în mod dinamic pe parcursul evoluţiei programului. Informaţiile din fiecare linie a tabelului pot fi:

  - numele variabilei; aşa cum s-a precizat la analiza lexicală, numele unei variabile este de obicei un şir de caractere, litere sau cifre, primul caracter trebuind să fi o literă.
  - tipul variabilei; relativ la tipurile de variabile utilizate în programare trebuie să facem menţiunea că acestea depind în mod esenţial de domeniul de aplicabilitate al limbajului. De exemplu, pentru un limbaj de tip matematic (Pascal), tipul unei variabile este caracterizat în general de trei atribute: *structura* (scalar, tablou), *calitatea* (intreg, flotant,

logic), *dimensiunea* (simplă precizie, dublă precizie, etc). În principiu, fiecare variabilă posedă toate aceste trei caracteristici, putând exista şi anumite restricţii, de exemplu, variabilele întregi pot avea dimensiunea numai simplă precizie sau dublă precizie.

- adresa; adresa de început a zonei de memorie rezervată variabilei;
- Alte informaţii: sunt informaţii de tip semantic, utilizate la depanarea programului; de exemplu, dacă variabila a fost sau nu iniţializată (adică dacă înainte de prima utilizare i s-a atribuit valoare), dacă a fost utilizată sau nu, etc.

Pentru uniformitatea prelucrării tabelului se poate prevedea o anumită lungime fixă pentru elementele tabelului, chiar dacă spaţiul nu este folosit în întregime. De exemplu, în cazul variabilelor de tip tablou, se pot utiliza două sau mai multe linii pentru a stoca valorile maxime ale indicilor (aceste dimensiuni vor fi utilizate atât la rezervarea zonei de memorie cât şi la calculul deplasamentului unui element.

Observaţie asupra funcţiilor. Denumirile funcţiilor sunt deseori utilizate ca variabile care conţin valoarea pe care funcţia respectivă o returnează. Este natural ca în acest caz numele funcţiei să fie considerat variabilă şi să fie stocat în acest tabel împreună cu atributele respective; de exemplu, unul din aceste atribute trebuie să fie contorul de amplasare, (CA), adică adresa primei instrucţii executabile a rutinei corespunzătoare funcţiei.

- *Tabelul de etichete.* Conţine etichetele utilizate într-un program şi adresele primelor instrucţii ale secvenţelor de program corespunzătoare. În esenţă, etichetele nu diferă de variabilele obişnuite, dar există anumite particularităţi; de exemplu, numele unei etichete trebuie să fie constituit dintr-un identificator urmat de un delimitator special (două puncte, etc.) sau numai din caractere numerice (cazul limbajului BASIC) iar valoarea unei etichete este întodeauna o valoare de adresă. De asemenea, în cazul etichetelor trebuie facută analiza compatibilităţii dintre etichetele definite şi etichetele referite.

- *Tabelul de variabile temporare.* Conţine variabile ale compilatorului necesare păstrării unor rezultate intermediare. Având în vedere că rezultatele intermediare, la fel ca şi variabilele definite de utilizator, pot avea tipuri diferite, se pot constitui zone separate în care se vor memora rezultate de un anumit tip bine precizat. În acest caz, variabilele temporare pot avea o structură liniară iar adresa unei astfel de variabile se determină cu o formulă de tipul

$$adr = adr_0 + c \times i$$

unde adr_0 este adresa de început a zonei, c este lungimea elementelor iar i numărul de ordine al variabilei temporare. Se poate adopta un sistem de alocare dinamică a variabilelor temporare în scopul economisirii memoriei.

- *Tabelul de constante*. Conţine constantele de diverse tipuri utilizate într-un program. În general este un tabel de mai mici dimensiuni, de aceea nu comportă probleme deosebite de organizare şi prelucrare.

5.1.2 Cvadruple şi triplete

Cvadruplele şi tripletele sunt forme intermediare ale programelor în care fiecare operaţie elementară împreună cu operanzii ei se stochează într-o secvenţă de patru sau trei elemente.

Cvadruple

Ne vom referi în continuare numai la operaţii binare, pentru celelalte operaţii putându-se proiecta structuri similare. Considerăm deci o secvenţă de text de forma A *op* B, unde A şi B sunt operanzi (variabile, constante, variabile temporare, etc.) iar *op* este un operator binar. Cvadrupul corespunzător acestei secvenţe va fi

$$\overline{op},\ \overline{A},\ \overline{B},\ t_i$$

unde $\overline{op}$, $\overline{A}$, $\overline{B}$ sunt codurile unităţilor lexicale respective iar t_i este o variabilă temporară. Semnificaţia acestui cvadruplu este următoarea: se execută operaţia definită de operatorul *op* între operanzii A şi B, în această ordine, iar rezultatul este memorat la t_i. În continuare vom folosi aceiaşi notaţie pentru un operand şi pentru codul operandului, posibilitatea de confuzie fiind redusă, deci nu vom mai utiliza notaţii de forma $\overline{A}$, etc.

Exemplu. Cvadruplele corespunzătoare expresiei aritmetice $a - b + (c + d)/e$ sunt

(1) $-,\ a,\ b,\ t_1$
(2) $+,\ c,\ d,\ t_2$
(3) $/,\ t_2,\ e,\ t_3$
(4) $+,\ t_1, t_3,\ t_4$

În cazul altor operaţii, chiar de n-aritate diferită de doi, se stabilesc structurile specifice ale cvadruplelor; pentru operaţii cu mai mult de doi operanzi, se pot utiliza două sau chiar mai multe cvadruple. Un exemplu de cvadruple pentru structurile uzuale ale limbajelor de programare este prezentat în tabelul 5.1.

Observaţie asupra instrucţiei **if**. Presupunem că structura instrucţiei este

if (expresie logica) then instrucţiune_ bloc;

Text Sursă	Cvadruple
a+b	+,a,b,t, idem -,*,/
a=b	=,a,b,
goto i	BR, i, ,
if (a==b) then	BE, a,b, n

Figura 5.1: Exemplu de codificare prin cvadruple

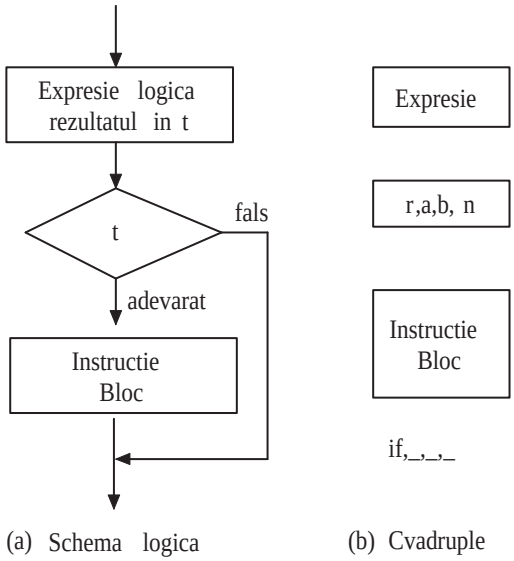

Figura 5.2: Instrucţiunea If logic

Semantica acestei instrucţii este prezentată în figura 5.2. Menţionăm că expresie logică va avea forma particulară $E_1 r E_2$, unde r este un operator relaţional.

În figură (a) prezintă schema logică a instrucţiei iar (b) cvadruplele corespunzătoare. Partea de cod corespunzătoare expresiei logice va stoca valoarea (logică) a expresiei într-o variabilă temporară t; această valoare va fi utilizată în instrucţia de salt, în conformitate cu tipul de salt prevăzut în programul sursă ($<$, $=$, etc.). În consecinţă, cvadrupul de salt r va trebui să prevadă această variabilă temporară, plus numărul de ordine n al primului cvadruplu care urmează după cvadruplele corespunzătoare instrucţiei sau blocului. De exemplu, acest cvadruplu poate avea următoarea formă $r,\ a,\ b,\ n$ unde n este numărul cvadruplului la care trebuie să se facă saltul.

Triplete

Este o altă posibilitate de format intermediar, asemănător cu cvadruplele, singura deosebire fiind accea că nu se prevede explicit o variabilă temporară pentru stocarea rezultatului. Prin urmare, tripletul corespunzător unei operaţii binare de forma $A\ op\ B$ va fi $op,\ A,\ B$. În cazul în care ca operand al unui triplet trebuie specificat rezultatul unui triplet anterior, se va scrie numărul de ordine al tripletului respectiv.

Exemplu. Pentru secvenţa de program $xx = a - b + (c + d)/e$ şirul de triplete generat va fi

(1) $-,\ a,\ b$
(2) $+,\ c,\ d$
(3) $/,\ (2),\ e$
(4) $+,\ (1),\ (3)$
(5) $=, xx, (4)$

Notaţia poloneză

Notaţia poloneză a fost introdusă iniţial pentru a se putea evalua expresii aritmetice printr-o singură parcurgere secvenţială. Din punctul de vedere al proiectării compilatoarelor, această notaţie poate fi utilizată ca un format intermediar al programelor, în mod special pentru limbajele de programare algoritmice, Fortran, Pascal. În continuare vom ilustra ideea de evaluare secvenţială a expresiilor aritmetice printr-un exemplu.

Considerăm expresia aritmetică $a*b+(c-d*e)*f$. Ideea şirului polonez constă în scrierea operanzilor şi a operatorilor acestei expresii (fără a utiliza paranteze) într-un şir cu următoarea proprietate: se parcurge şirul secvenţial de la stânga la dreapta şi fiecare operator întâlnit provoacă efectuarea operaţiei respective între cei doi operanzi din faţă, în ordinea apariţiei lor. Pentru exemplul considerat, un

şir care satisface această condiţie este

$$ab * cde * -f * +$$

Vom numi acest şir notaţia poloneză corespunzătoare expresiei, sau *notaţia poloneză inversă*. Evaluarea se poate face utilizând o stivă cu următoarea prelucrare: se parcurge secvenţial şirul, operanzii se introduc în stivă fără nici un control, iar fiecare operator citit provoacă efectuarea operaţiei între operanzii aflaţi în penultimul şi ultimul element al stivei, în această ordine şi memorarea rezultatului în penultimul element. Starea dinamică a stivei pentru exemplul considerat este următoarea:

(1)	a	a*b	a*b	a*b	a*b	a*b+(c-d*e)*f
(2)	b	c	c	c-d*e	(c-d*e)*f	
(3)		d	d*e	f		
(4)		e				

Conţinutul stivei a fost rescris atunci când un operator provoacă efectuarea unei operaţii. Se poate observa că şirul considerat de noi (notaţia poloneză a expresiei) satisface condiţia de evaluare secvenţială. De fapt, această condiţie este esenţială şi ea poate fi luată chiar ca o definiţie a şirului polonez.

Fie V un alfabet şi B o mulţime de operatori binari.

Definiţia 1. Vom numi şir polonez orice cuvânt peste alfabetul $V \cup B$ obţinut recursiv cu ajutorul regulilor:

(1) dacă $a \in V$ atunci a este şir polonez;

(2) dacă p şi q sunt şiruri poloneze şi $b \in B$, atunci pqb este şir polonez.

Să observăm că utilizând aceste două reguli, putem să obţinem succesiv toate şirurile poloneze având operanzii V şi operatorii binari B. De exemplu, dacă $V = \{a, b, c\}$ şi $B = \{+, *\}$, atunci şirurile poloneze vor fi $a, b, c, ab+, ac+, bc+, ab*$, $\ldots, aab++, aac++, \ldots$.

În cele ce urmează vom da o altă definiţie, tot constructivă, folosind gramaticile generative Chomsky. Este necesar să precizăm expresiile aritmetice pentru care introducem aceste şiruri.

Definiţia 2. Fie gramatica $G_{exp} = (\{E, T, F\}, \{a, +, *\}, E, \mathcal{P})$ unde regulile de rescriere sunt

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

Observaţie. Terminalul a desemnează operanzii, deci a este de fapt o notaţie simbolică pentru variabile de forma $a, b, c, etc.$ Este uşor de văzut că această gramatică generează expresii aritmetice fără paranteze, în care singurele operaţii sunt $+$ şi $*$. Utilizarea a trei neterminale E, T, F se face numai pentru a preciza ponderea operaţiilor, creşterea făcându-se odată cu adâncimea, deci $p(+) < p(*)$.

Acelaşi limbaj al expresiilor aritmetice fără paranteze poate fi generat cu o gramatică cu un singur neterminal, fie acesta A, şi având regulile

$$A \to A + A | A * A | a$$

Dar în acest caz trebuie introduse alte convenţii pentru a preciza ordinea operaţiilor; de exemplu, pentru expresia $a+b*c$, în acest ultim caz, nimic nu arată că operaţia $*$ trebuie făcută înainte de $+$. Trebuie să precizăm că prin considerarea a numai doi operanzi nu se impune o constrângere asupra noţiunii de expresie aritmetică, foarte uşor putem generaliza această gramatică la un număr oarecare de operatori binari, de exemplu

$$\begin{gathered} A_1 \to A_1 op_1 A_2 | A_2 \\ A_2 \to A_2 op_2 A_3 | A_3 \\ \text{etc.} \end{gathered}$$

De asemenea, putem considera operatori de n-aritate oarecare precum şi reguli de rescriere în concordanţă cu sintaxa pe care dorim să o satisfacă aceştia. Utilizarea unor paranteze pentru a indica o anumită ordine de efectuare a operaţiilor, iarăşi nu diferă esenţial de cazul expresiilor fără paranteze. Putem folosi algoritmul de modificare dinamică a ponderilor descris la gramatici operatoriale: parcurgem expresia de la stânga la dreapta şi ataşăm fiecărui operator ponderea sa; în momentul în care întâlnim o paranteză deschisă, mărim toate ponderile cu o constantă mai mare decât cea mai mare pondere iar la întâlnirea unei paranteze închise, scădem această constantă. Apoi eliminăm toate parantezele. Este evident că în acest fel operaţiile din interiorul parantezelor vor avea prioritatae mai mare, mergându-se spre parantezele din interior.

Definiţia 3. Fie $G_p = (V_N, V_T, x_0, \mathcal{P})$, unde $V_N = \{A\}$, $V_T = \{a, +, *\}$, $x_0 = A$ iar regulile sunt

$$A \to AA + | AA * | a$$

Orice cuvânt $p \in L(G_p)$ va fi numit şir polonez. Este natural ca aceste şiruri poloneze să fie puse în legătură cu expresiile aritmetice fără paranteze prezentate mai sus. Gramatică G_p nu stabileşte o ordine intrinsecă de efectuare a operaţiilor, întocmai ca şi gramatica cu un singur neterminal de generare a expresiilor. Convenim însă ca această ordine să fie cea *secvenţială*.

Dacă $p \in L(G_p)$ atunci convenim să notăm cu $aes(p)$ cuvântul obţinut din p cu ajutorul algoritmului de evaluare secvenţială.

Următoarea teoremă stabileşte legătura între cele două entităţi.

Teoremă. Pentru orice cuvânt $e \in L(G_{exp})$ există un $p_e \in L(G_p)$ astfel încât $aes(p_e) = e$.

Demonstraţie. Procedăm prin inducţie asupra numărului n de operatori din e. Dacă $n = 1$ atunci $e = a+b$ sau $e = a*b$ şi şirul polonez care satisface condiţia din teoremă va fi $p_e = ab+$ sau $p_e = ab*$. Să presupunem că proprietatea este

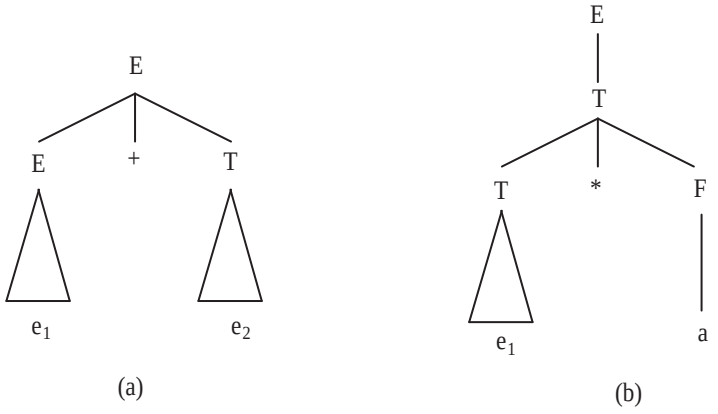

Figura 5.3: Structurile posibile ale arborelui de derivare

adevărată pentru o expresie care conţine un număr de operatori mai mic sau egal cu n şi considerăm o expresie e cu n operatori. Arborele de derivare al acesteia va avea una din formele prezentate în figura 5.3.

În cazul (b) putem considera $e_2 = a$. În ambele cazuri expresiile e_1 şi e_2 au un număr de operanzi inferior lui n şi în conformitate cu ipoteza inductivă avem $aes(p_{e_1}) = e_1$ şi $aes(p_{e_2}) = e_2$. Luăm $p_e = p_{e_1}p_{e_2}+$ pentru cazul (a) şi $p_e = p_{e_1}p_{e_2}*$ pentru cazul (b). Este evident că $aes(p_e) = e$.

5.2 Generarea formatului intermediar

În această secţiune vom analiza problema generării formatului intermediar pentru limbaje algoritmice de tip Fortran, Pascal, Basic. Formele intermediare avute în vedere sunt cvadruplele şi notaţia poloneză, într-un singur caz ne vom referi şi la triplete, având în vedere că principial generarea acestora nu diferă esenţial de cea a cvadruplelor.

Schema generală a algoritmilor este următoarea.

Fie G gramatica (de tipul 2) care generează limbajul de programare. Prima fază constă în stabilirea unor elemente semantice în concordanţă cu structura limbajului; astfel, fiecărui simbol $x \in V_G$ i se atribuie un anumit *sens semantic*, notat $S(x)$, sens care conţine o anumită informaţie depinzând de simbolul respectiv, de contextul în care acesta apare, etc. De asemenea, fiecărei reguli de generare i se ataşează o anu- mită *rutină semantică*. Operaţiile obişnuite care se efectuează în aceste rutine sunt: analize specifice, atribuirea unui sens semantic simbolului din stânga, generarea unui cvadruplu sau a unei secvenţe de şir polonez, etc. Faza a doua este faza de generare propriuzisă şi ea comportă în principiu următoarele operaţii: Se efectuează analiza sintactică a tex-

tului sursă cu un algoritm de analiză pe care proiectantul de compilator a decis să îl utilizeze. Indiferent de algoritmul ales, analiza poate fi concepută în două etape, mai întâi construcţia arborelui de derivare şi apoi reducerea lui succesivă prin eliminarea unor subarbori. În mod natural aceşti subarbori sunt cei corespunzători frazelor simple stângi aşa încât acţiunile care se realizează cu acest prilej sunt cele corespunzătoare rutinelor semantice din prima fază. Menţionăm că frontierele subarborilor care se reduc vor constitui, prin definiţie, unităţile sintactice ale limbajului. În momentul în care se ajunge la simbolul de start (s-a efectuat întreaga reducere) într-un fişier de ieşire se obţine programul în format intermediar, cvadruple, şir polonez, etc. Se poate observa că analiza sintactică joacă în acest proces un rol central. În cele ce urmează vom presupune că analiza sintactică se efectuează cu un algoritm de tip bottom-up şi deci subarborele care se reduce are drept frontieră fraza simplă stângă.

5.2.1 Generarea cvadruplelor

Gramatica care generează expresiile aritmetice G_E este cunoscută; pentru a ne situa într-un context cât mai apropiat de cazul real, vom completa această gramatică cu operaţiile - şi / (diferenţă şi împărţire). Prin urmare regulile acestei gramatici vor fi

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

Sensul semantic este prestabilit numai pentru simbolul terminal a, şi anume, $S(a)$ este adresa zonei de memorie ataşată identificatorului desemnat de a. Pentru celelalte simboluri neterminale, E, T, F sensul semantic se atribuie pe parcursul efectuării analizei. Rutinele semantice corespunzătoare regulilor sunt prezentate în tabelul 5.4. Menţionăm că acţiunile prevăzute în aceste rutine sunt stabilite în conformitate cu înţelesul pe care proiectantul doreşte să îl atribuie diverselor structuri şi nu au o raţiune intrinsecă. Ca o regulă generală însă, putem să facem precizarea că în cazul cvadruplelor, variabila temporară în care se reţine rezultatul intermediar, va fi transmisă ca sens semantic simbolului din stânga.

Exemplu. Considerăm următoarea expresie aritmetică $(a+b)*c-d*(e-f)$. Arborele de derivare corespunzător este prezentat în figura 5.5.

În dreapta fiecărui nod al arborelui (nodurile sunt etichetate cu simboluri ale gramaticii), se află înscris sensul semantic corespunzător variabilei şi contextului în care aceasta se află. Procesul de generare al cvadruplelor se desfăşoară astfel: la începutul procesului, după cum se poate vedea pe arborele de generare, fraza simplă stângă este a. Se efectuează reducerea $F \to a$ şi se aplică rutina semantică corespunzătoare acestei reguli (în tabelul rutinelor poziţia (8)); singura operaţie care se face este transmiterea sensului semantic lui F. În continuare, acest sens

Regula	*Cvadruplu*	*Sens*
(1) $E_1 \to E_2 + T$	$+, S(E_2), S(T), t_i$	$S(E_1) = t_i$
(2) $E_1 \to E_2 - T$	$-, S(E_2), S(T), t_i$	$S(E_1) = t_i$
(3) $E \to T$		$S(E) = S(T)$
(4) $T_1 \to T_2 * F$	$*, S(T_2), S(F), t_i$	$S(T_1) = t_i$
(5) $T_1 \to T_2/F$	$/, S(T_2), S(F), t_i$	$S(T_1) = t_i$
(6) $T \to F$		$S(T) = S(F)$
(7) $F \to (E)$		$S(F) = S(E)$
(8) $F \to a$		$S(F) = Adr(a)$

Figura 5.4: Rutine semantice asociate regulilor

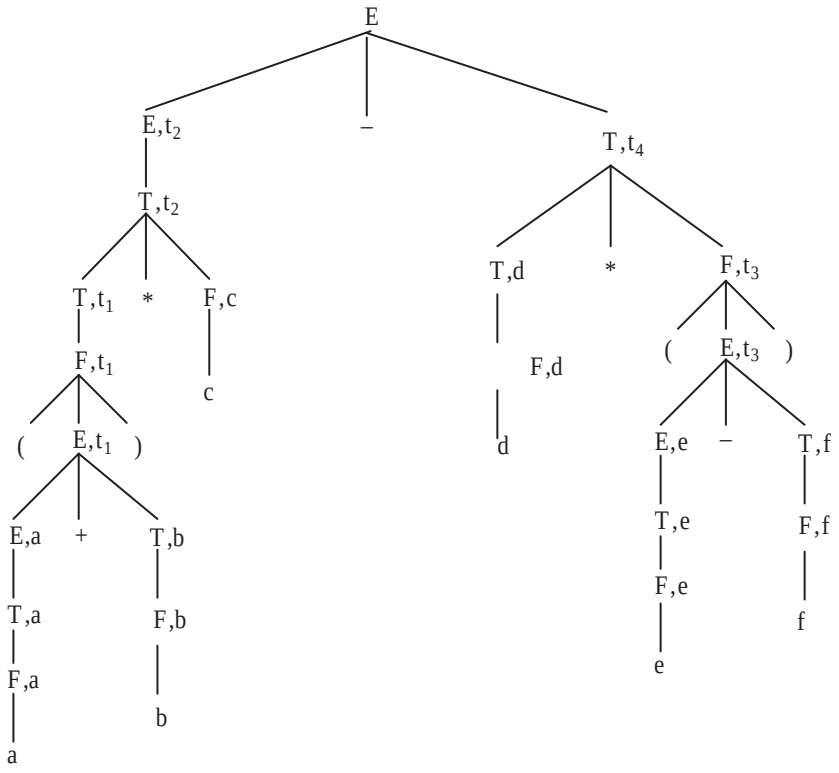

Figura 5.5: Arborele de derivare

Regula	*Cvadruplu*	*Sens*
(1) $E_1 \rightarrow E_2 + T$	$+, S(E_2), S(T)$	$S(E_1) = (n)$
(2) $E_1 \rightarrow E_2 - T$	$-, S(E_2), S(T)$	$S(E_1) = (n)$
(3) $E \rightarrow T$		$S(E) = S(T)$
(4) $T_1 \rightarrow T_2 * F$	$*, S(T_2), S(F)$	$S(T_1) = (n)$
(5) $T_1 \rightarrow T_2/F$	$/, S(T_2), S(F)$	$S(T_1) = (n)$
(6) $T \rightarrow F$		$S(T) = S(F)$
(7) $F \rightarrow (E)$		$S(F) = S(E)$
(8) $F \rightarrow a$		$S(F) = Adr(a)$

Figura 5.6: Rutine semantice pentru generarea tripletelor

se transmite până la E, apoi fraza simplă stângă este b şi se transmite sensul semantic până la T. În acest moment fraza stângă este $E + T$ iar cele două simboluri au sensurile $S(E) = a, S(T) = b$; se efectuează reducerea $E \rightarrow E + T$ şi se generează cvadruplul $+, a, b, t_1$ iar sensul semantic care se transmite lui E este t_1. Procesul continuă în acelaşi mod şi se obţine în final şirul de cvadruple:

$+, a, b, t_1$
$*, t_1, c, t_2$
$-, e, f, t_3$
$*, d, t_3, t_4$
$-, t_2, t_4, t_5$

5.2.2 Generarea tripletelor

Modificarea ceva mai importantă intervine la rutinele semantice precum şi la faptul că după generarea unui triplet, sensul semantic care se transmite va fi numărul de ordine al tripletului generat. Aceste rutine sunt prezentate în tabelul 5.6.

Generarea tripletelor se desfăşoară la fel cu cea a cvadruplelor. Pentru expresia considerată la generarea cvadruplelor se obţine următorul şir de triplete

(1) +, a, b
(2) *, (1), c
(3) -, e, f
(4) *, d, (3)
(5) -, (2), (4)

	Regula	Notaţia poloneză
(1)	$E \to E + T$	$p_i = +$
(2)	$E \to E - T$	$p_i =$
(3)	$E \to T$	
(4)	$T \to T * F$	$p_i = *$
(5)	$T \to T/F$	$p_i = /$
(6)	$T \to F$	
(7)	$F \to (E)$	
(8)	$F \to a$	$p_i = Adr(a)$

Figura 5.7: Metoda rutinelor semantice

5.2.3 Generarea şirului polonez

Tehnica rutinelor semantice

După cum s-a precizat, notaţia poloneză constituie o interesantă şi utilă formă intermediară a programelor. În paragraful precedent această notaţie s-a introdus numai pentru expresii aritmetice fără paranteze, dar această construcţie se poate face pentru majoritatea structurilor sintactice ale limbajelor de programare. Desigur că cea mai naturală tehnică de generare a notaţiei poloneze pentru o expresie este procedura generală prezentată la începutul capitolului, bazată pe rutinele semantice ataşate regulilor de generare. Spre deosebire de procedura de generare a cvadruplelor şi a tripletelor, pentru şirurile poloneze nu mai avem nevoie de sensul semantic al neterminalelor; acestea sunt utilizate în momentul apariţiei lor şi deci nu mai este necesară stocarea unor informaţii suplimentare. În continuare sunt prezentate rutinele semantice pentru generarea notaţiei poloneze corespunzătoare expresiilor aritmetice cu paranteze (vezi figura 5.7), generate de gramatica G_E. Notaţia p_i o utilizăm pentru a desemna elementul i din şirul polonez; indicele i se actualizează după fiecare prelucrare a şirului.

Exemplu. Vom aplica rutinele semantice de mai sus pentru generarea şirului polonez corespunzător expresiei aritmetice $a*b+(c-d*e)*f$. Analiza sintactică se efectuează cu un algoritm de tip bottom-up, aşa încât trasarea explicită a arborelui de derivare nu mai este necesară. Prelucrarea efectivă se poate organiza într-un tabel ca în figura 5.8; menţionăm că fraza simplă stângă este indicată prin sublinierea secvenţei corespunzătoare de text.

Tehnica parantezelor

Ideea de bază constă în aceea că se plasează între două paranteze fiecare operator împreună cu cei doi operanzi ai săi; apoi generarea notaţiei poloneze se poate face simplu, folosind o stivă de operatori.

Textul sursă	Regula aplicată	Notaţia poloneză
a * b + (c - d * e) * f	$F \to a$	a
F * b + (c - d * e) * f	$F \to b$	ab
F * F + (c - d * e) * f	$T \to T * F,\ T \to F$	ab*
T + (c - d * e) * f	$F \to c$	ab*c
T + (F - d * e) * f	$F \to d$	ab*cd
T + (F - F * e) * f	$F \to e$	ab*cde
T + (F - F * F) * f	$T \to T * F,\ T \to F$	ab*cde*
T + (F - T) * f	$E \to E - T,\ E \to T,\ T \to F$	ab*cde*-
T + (E) * f	$F \to (E)$	ab*cde*-
T + F * f	$F \to f$	ab*cde*-f
T + F * F	$T \to T * F,\ T \to F$	ab*cde*-f*
T + T	$E \to E + T,\ E \to T$	ab*cde*-f*+
E		ab*cde*-f*+

Figura 5.8: Generarea sirului polonez: tehnica analizei sintactice

Plasarea parantezelor. Se atribuie fiecărui operator câte o pondere în conformitate cu convenţia de priorităţi sau cu gramatica care generează expresiile considerate. În cazul unei expresii aritmetice fără paranteze se poate proceda în următorii paşi:

- (1) Se ataşează fiecărui operator ponderea sa.
- (2) Se parcurge expresia de la stânga la dreapta până când între doi operatori se găseşte relaţia $\geq$; de exemplu $\ldots a * b + c \ldots$ şi, cu convenţia uzuală, $* \geq +$. Se plasează între paranteze operatorul $*$ împreună cu cei doi operanzi ai săi, adică $\ldots (a * b) + c \ldots$.
- (3) Se şterge ponderea operatorului curent (cel care a fost inclus între paranteze) şi se reia procesul de la punctul (2) începând cu operatorul următor (în exemplul nostru de la +).

De exemplu, expresia $a/b * c + d - e$ va avea forma ((((a/b)*c)+d)-e).

În cazul unor expresii aritmetice cu paranteze se foloseşte un algoritm asemănător în care se modifică pasul (1) şi anume: se parcurge expresia de la stânga la dreapta ataşând fiecărui operator ponderea sa. La întâlnirea unei paranteze deschise se modifică provizoriu ponderile prin adăugarea unei constante întregi (această constantă trebuie să fie mare decât cea mai mică pondere) şi se atribuie în continuare noile ponderi. La întâlnirea unei paranteze închise, această constantă este scăzută, astfel încât operatorii din interiorul parantezelor vor avea ponderi superioare. Apoi toate parantezele se şterg. De exemplu, în expresia

$$(a + b) * (c/(d - e) - f),$$

după efectuarea acestei operaţii, presupunând că iniţial ponderile operatorilor $+, -, *, /$ sunt, respectiv, $1, 2, 3, 4$ şi constanta de majorare are valoarea 10, operatorilor li se vor atribui ponderile:

a	+	b	*	c	/	d	-	e	-	f
	11		3		14		22		12	

Generarea şirului polonez. Pornind de la expresiile aritmetice în care au fost plasate parantezele în conformitate cu prima parte, şirul polonez poate fi construit cu ajutorul algoritmului:

- Se parcurge expresia de la stânga la dreapta, parantezele deschise sunt ignorate şi se execută operaţiile
  - operanzii se scriu succesiv în şirul polonez;
  - operatorii se scriu într-o stivă.
- La citirea unei paranteze închise, operatorul din vârful stivei se scrie în şirul polonez.

Tehnica stivei de operatori

Este probabil cea mai răspândită metodă de generare a şirului polonez pentru expresii aritmetice. Se utilizează două liste liniare şi o stivă. Prima listă conţine expresia aritmetică dată iar în cealaltă se obţine succesiv şirul polonez corespunzător. Operatorii sunt plasaţi în stivă ţinându-se cont de ponderile lor, apoi sunt extraşi şi scrişi în şir conform cu un algoritm care în esenţă reproduce acţiunea rutinelor semantice. Mecanismul care realizează acest proces este prezentat în figura 5.9.

Algoritmul este următorul:

(1) Operanzii se scriu direct în şirul polonez.

(2) Operatorii şi parantezele deschise se scriu în stivă respectându-se următoarele reguli

- Parantezele deschise au pondere 0 şi se scriu fără condiţii;
- Pentru ceilalţi operatori, se compară ponderea operatorului de introdus cu ponderea operatorului din vârful stivei; dacă aceasta este mai mare, introducerea are loc; dacă este mai mică sau egală, se extrage vârful stivei (acesta trece în şirul polonez), se reia comparaţia cu operatorul din vârful stivei până la prima paranteză deschisă sau până la baza stivei, şi apoi se introduce noul operator;

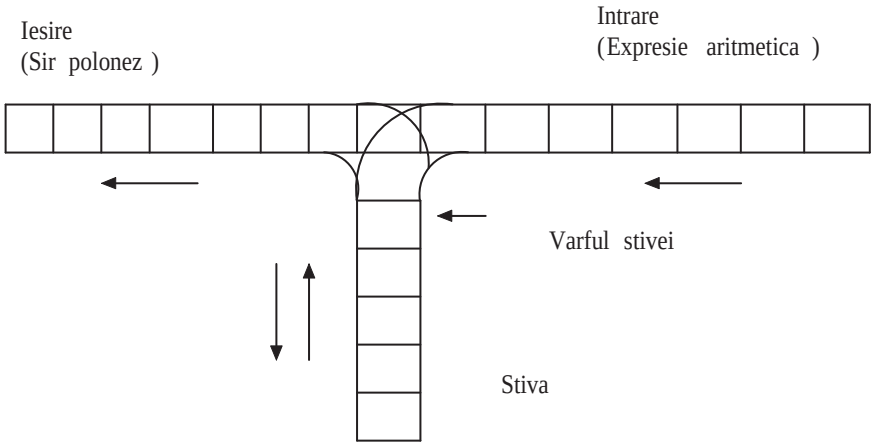

Figura 5.9: Structura dispozitivului

(3) Citirea unei paranteze închise provoacă extragerea operatorilor din stivă până la prima paranteză deschisă; aceasta este extrasă şi ea din stivă (dar paranteza extrasă nu trece în şirul polonez) iar în continuare cele două paranteze sunt ignorate.

(4) În finalul procesului, deci după ce ultimul operand a fost trecut în şirul polonez, se extrag toţi operatorii din stivă şi se scriu în şirul polonez.

Exemplu. Pentru expresia a * b + (c - d * e) * f, stare dinamică a stivei de operatori este prezentată în figura 5.10.

5.3 Generarea formatului intermediar pentru instrucţii clasice

5.3.1 Instrucţiunea de atribuire

to fill

5.3.2 Instructiunea If

to fill

5.3.3 Variabile indexate

to fill

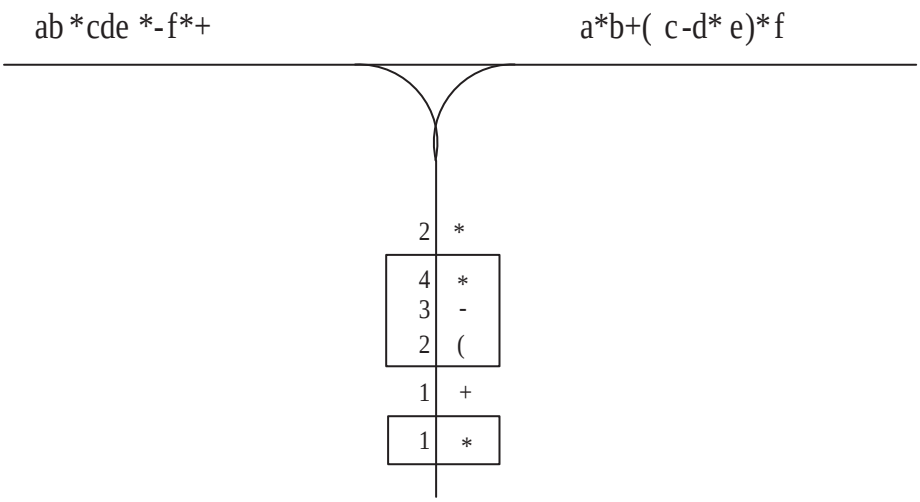

Figura 5.10: Starea dinamică a stivei

Capitolul 6

Masina Turing

6.1 Limbaje de tipul zero

Limbajele de tipul 0 sunt generate de gramatici Chomsky fără restricţii asupra regulilor de derivare. Se poate arăta că orice gramatică de tipul zero acceptă următoarea formă normală

$$S \to SA,$$
$$B \to C|i|\lambda,$$
$$DE \to FH,$$

unde $A, C, F, H \neq S$. Se observă că singura deosebire faţă de gramaticile dependente de context este aceea că putem avea şi λ–reguli.

Demonstraţia existenţei formei normale se face prin transformări succesive ale regulilor. În prealabil gramatica se completează cu neterminale noi astfel încât orice regulă $u \to v$ care nu este de ştergere să respecte condiţia de monotonie, adică $|u| \leq |v|$. Acest lucru se poate realiza după cum urmează. Fie

$$X_1 \dots X_n \to Y_1 \dots Y_m, n > m,$$

o regulă care nu respectă condiţia de monotonie. Vom pune

$$X_1 \dots X_n \to Y_1 \dots Y_m Z_{m+1} \dots Z_n,\ Z_{m+1} \to \lambda, \dots, Z_n \to \lambda.$$

Este clar că orice derivare directă $u \Rightarrow v$ în gramatica iniţială se poate obţine şi în gramatica modificată şi reciproc (neterminalele Z nu pot apărea în stânga altor relaţii). În acest fel, gramatica se va transforma astfel încât să conţină două categorii de reguli: reguli care respectă condiţia de monotonie şi λ–reguli.

În continuare reducem lungimea părţii drepte a regulilor folosind următoarea transformare. Fie $u \to v \in P, |v| > 2$; atunci $v = Y_1Y_2Y_3v'$.

1. dacă $u = X_1$ atunci regula se înlocuieşte cu

$$X_1 \to Z_1Z_2,$$
$$Z_1 \to Y_1,$$
$$Z_2 \to Y_2Y_3v'.$$

2. dacă $u = X_1X_2u'$ atunci regula se înlocuieşte cu

$$X_1X_2 \to Z_1Z_2,$$
$$Z_1 \to Y_1,$$
$$Z_2u' \to Y_2Y_3v'.$$

În ambele cazuri Z_1, Z_2 sunt neterminale noi.

Este clar că noile lungimea părţii drepte a regulilor se reduce cu o unitate pentru cazul în care există minim 3 simboluriîn dreapta. repetând procedura se poate obţine o gramatică echivalentă în care regulile au una din formele:

1. $B \to C|i|\lambda$,
2. $DE \to FH$,
3. $X \to YZ$.

Regulile (1) şi (2) satisfac condiţiile de la forma normală, iar cele de forma (3) presupunem că nu satisfac aceste condiţii, deci că $X \neq S$ sau $Y \neq S$. În general, putem avea mai multe astfel de reguli; pentru simplificare presupunem că există două reguli de forma (3):

$$(3');\ \ X_1 \to Y_1Z_1,\ \ X_2 \to Y_2Z_2.$$

Construim o gramatică echivalentă $G' = (V_N \cup \{S', \bar{X}_1, \bar{X}_2\}, V_T, S', P')$, unde P' conţine toate regulile care convin (de formele (1) şi (2)) precum şi

$$S' \to S,$$
$$S' \to S'\bar{X}_1|S'\bar{X}_2,$$
$$X_1\bar{X}_1 \to Y_1Z_1, X_2\bar{X}_2 \to Y_2Z_2.$$

De asemenea vom include în P' următoarel reguli de comutare (relativ la simbolurile nou introduse $\bar{X}_1, \bar{X}_2$):

$$\begin{matrix} A\bar{X}_1 \to \bar{X}_1A, \\ A\bar{X}_2 \to \bar{X}_2A, \end{matrix}, \forall A \in V_N.$$

Se vede că G' este o gramatică în formă normală. Este trivial să arătăm că $L(G) = L(G')$.

Fie $p \in L(G)$, deci $S \overset{*}{\underset{G}{\Rightarrow}} p$. Presupunem că în această derivare există o singură secvenţă $u \overset{*}{\Rightarrow} v$ în care se aplică reguli de forma $(3')$, adică

$$G:\ \ S \overset{*}{\Rightarrow} u \overset{*}{\Rightarrow} v \overset{*}{\Rightarrow} p.$$

Presupunem că secvenţa $u \overset{*}{\Rightarrow} v$ are forma

$$u = u_1X_1u_2X_2u_3X_2u_4 \overset{*}{\Rightarrow} u_1Y_1Z_1u_2Y_2Z_2u_2Y_2Z_3u_4 = v,$$

unde $u_1, u_2, u_3, u_4 \in V_N^*$. În G' putem scrie

$$\begin{aligned}
&S' \Rightarrow S'\bar{X}_2 \Rightarrow S'\bar{X}_2\bar{X}_2 \Rightarrow S'\bar{X}_1\bar{X}_2\bar{X}_2 \Rightarrow S\bar{X}_1\bar{X}_2\bar{X}_2 \\
&\overset{*}{\Rightarrow} u\bar{X}_1\bar{X}_2\bar{X}_2 = u_1X_1u_2X_2u_3X_2u_4\bar{X}_1\bar{X}_2\bar{X}_2 \overset{*}{\Rightarrow} u_1X_1\bar{X}_1u_2X_2\bar{X}_2u_3X_2\bar{X}_2u_4 \\
&\overset{*}{\Rightarrow} u_1Y_1Z_1u_2Y_2Z_2u_3Y_2Z_2u_4 = v \overset{*}{\Rightarrow} p
\end{aligned}$$

Prin urmare, $S' \underset{G'}{\overset{*}{\Rightarrow}} p$ şi $p \in L(G')$.

Invers, fie $p \in L(G')$, deci $S' \underset{G'}{\overset{*}{\Rightarrow}} p$. Dacă în derivarea $S' \underset{G'}{\overset{*}{\Rightarrow}} p$ apar simbolurile $\bar{X}_1, \bar{X}_2$ ele nu pot apărea dacât în urma aplicării regulilor $S' \to S'\bar{X}_1$ şi $S' \to S'\bar{X}_2$, deci la începutul derivării, apoi singura posibilitate de continuare este $S' \to S$; de asemenea, aceste simboluri nu pot fi eliminate decât cu regulile $X_1\bar{X}_1 \to Y_1Z_1$ şi $X_2\bar{X}_2 \to Y_2Z_2$, etc. Astfel derivarea noastră trebuie să aibă forma descrisă mai sus, de unde rezultă $S \underset{G}{\overset{*}{\Rightarrow}} p, p \in L(G)$.□

Relativ la puterea generativă a gramaticilor de tipul zero se poate arăta că $\mathcal{L}_1 \subset \mathcal{L}_0$ (strict). Demonstraţia se face pe o cale indirectă.

Menţionăm şi următoarea "teoremă a spaţiului de lucru". Fie G o gramatică de tipul zero. Pentru o derivare

$$D: \quad S = u_0 \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n = p \in V_T^*$$

definim

$$WS_D(p) = \underset{i=1,\ldots,n}{max} |u_i|$$

şi

$$WS(p) = \underset{D}{min}\{WS_D(p)\}.$$

Observaţie. WS este prescurtare de la *working space*.

Spunem că o gramatică de tipul zero G are spaţiul de lucru mărginit dacă există $k \in N$ astfel încât pentru $\forall p \in L(G)$ să avem $WS(p) \leq k|p|$. Să observăm că orice gramatică care nu are reguli de ştergere (excepţie $S \to \lambda$ şi atunci S nu apare în dreapta) satisface această condiţie cu $k = 1$.

Teorema 6.1 *(teorema spaţiului de lucru) dacă o gramatică G are spaţiul de lucru mărginit, atunci $L(G) \in \mathcal{L}_1$.*

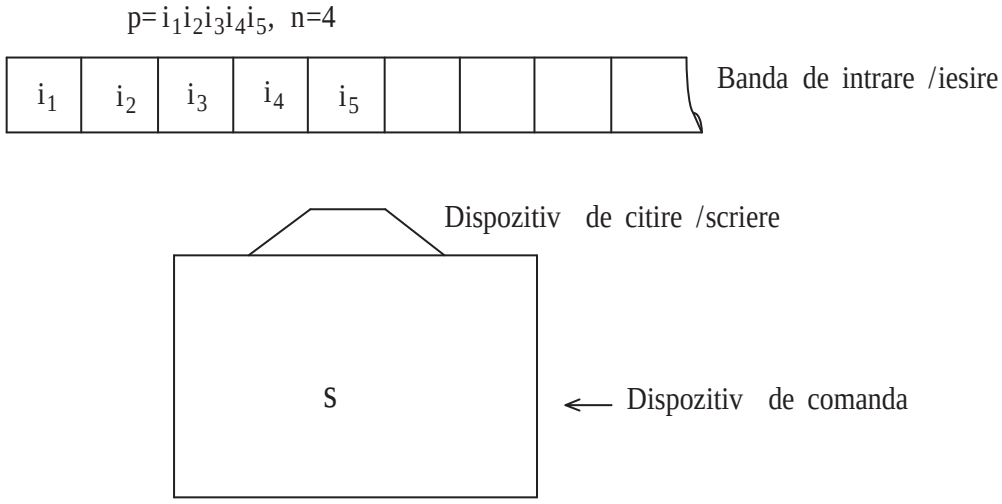

Figura 6.1: Reprezentarea schematică a maşinii Turing

6.2 Maşina Turing

Conceptul de maşină Turing. Maşina Turing este un mecanism de recunoaştere a limbajelor de tipul zero. În felul acesta, familiilor de limbaje din clasificarea Chomsky le corespund automate specifice de recunoaştere.

Familia $\mathcal{L}_3$ - automate finite,
Familia $\mathcal{L}_2$ - automate push–down,
Familia $\mathcal{L}_1$ - automate liniar mărginite,
Familia $\mathcal{L}_0$ - maşina Turing.

O maşină Turing (prescurtat MT) se compune dintr–un dispozitiv de comandă care posedă un dispozitiv de scriere–citire şi o bandă de intrare. Banda de intrare se consideră mărginită la stânga şi nemărginită la dreapta; ea conţine simboluri ale unui alfabet de intrare, începând din prima poziţie. Alfabetul are un simbol special numit blanc şi notat $\flat$ (sau spaţiu), care se consideră înregistrat în toată partea neocupată a benzii. Indicele simbolului din dreptul dispozitivului de scriere–citire îl notăm cu n şi el va constitui un element al stării maşinii Turing. Dispozitivul de comandă se află într-o anumită stare internă, element al unei mulţimi finite de stări.

Schema unei maşini Turing este dată în figura 6.1. O maşină Turing funcţionează în paşi discreţi. Un pas de funcţionare constă din:

Dispozitivul de comandă citeşte simbolul aflat în dreptul dispozitivului de scriere–citire; în funcţie de simbolul citit, maşina Turing trece într-o nouă stare internă, scrie pe bandă un nou simbol (întotdeauna diferit de blanc) în locul simbolului citit şi mută banda cu o poziţie (spre stânga sau dreapta) sau o lasă

pe loc. Convenţional, vom nota cu $+1$ o mişcare spre stânga, cu -1 spre dreapta şi cu 0 pe loc.

Din punct de vedere matematic, o maşină Turing este un sistem

$$MT = (\Sigma, I, f, s_0, \Sigma_f)$$

unde:

Σ este mulţimea de stări;

I este alfabetul de intrare;

$f : (\Sigma \times I)' \longrightarrow \Sigma \times (I - \{\flat\}) \times \{1, 0, -1\},\ (\Sigma \times I)' \subseteq \Sigma \times I,$
este funcţia de evoluţie;

$s_0 \in \Sigma$ este simbolul de start;

$\Sigma_f \subseteq \Sigma$ este mulţimea de stări finale.

Observaţie. Funcţia f nu este definită pe întregul produs $\Sigma \times I$; dacă maşina ajunge într-o stare s iar în dreptul dispozitivului de scriere–citire se află simbolul i şi $(s, i) \notin (\Sigma \times I)'$ spunem că maşina se blochează. Există şi alte cazuri de blocare, de exemplu situaţia în care dispozitivul de scriere–citire se află în dreptul primului simbol, iar pasul de funcţionare prevede o mişcare a benzii spre dreapta.

O stare (sau configuraţie) a unei maşini Turing este un triplet de forma $\sigma = (s, p, n)$ unde s este starea internă, p este cuvântul scris pe bandă iar n este indicele simbolului din dreptul dispozitivului de scriere–citire.

Vom spune că starea $\sigma_1 = (s_1, p_1, n_1)$ *evoluează direct* în starea $\sigma_2 = (s_2, p_1, n_2)$ şi vom scrie $\sigma_1 \longmapsto \sigma_2$ dacă se efectuează un pas de evoluţie. În termenii funcţiei de evoluţie, avem

$(1) f(s_1, i_{n_1}) = (s_2, i, +1),\ p_2 = i_1 \dots i_{n_1-1} i i_{n_1+1} \dots i_m,\ n_2 = n_1 + 1;$

$(2) f(s_1, i_{n_1}) = (s_2, i, -1),\ p_2 = i_1 \dots i_{n_1-1} i i_{n_1+1} \dots i_m,\ n_2 = n_1 - 1;$

$(3) f(s_1, i_{n_1}) = (s_2, i, 0),\ p_2 = i_1 \dots i_{n_1-1} i i_{n_1+1} \dots i_m,\ n_2 = n_1;$

$(4) f(s_1, \flat) = (s_2, i, +1),\ p_2 = p_1 i,\ n_2 = n_1 + 1;$

$(5) f(s_1, \flat) = (s_2, i, -1),\ p_2 = p_1 i,\ n_2 = n_1 - 1;$

$(6) f(s_1, \flat) = (s_2, i, 0),\ p_2 = p_1 i,\ n_2 = n_1;$

Vom spune că σ' *evoluează* (fără specificaţia direct) în σ'' şi vom nota $\sigma' \overset{*}{\longmapsto} \sigma''$ dacă $\sigma' = \sigma''$ sau dacă există $\sigma_1, \dots, \sigma_n$ astfel încât

$$\sigma' = \sigma_1 \longmapsto \sigma_2 \longmapsto \dots \longmapsto \sigma_n = \sigma''.$$

Limbajul recunoscut de o maşină Turing este prin definiţie

$$L(MT) = \{p | p \in I^*, (s_0, p, 1) \overset{*}{\longmapsto} (s, q, \epsilon), s \in \Sigma_f\}.$$

Observaţie. Este posibil ca maşina să ajungă într-o stare finală înainte de citirea integrală a lui p; analiza stării finale trebuie făcută numai după parcurgerea cuvântului.

Exemplu. Considerăm maşina turing $MT = (\Sigma, I, f, s_0, \Sigma_f)$ unde $\Sigma = \{s_0, s_1, s_2\}$, $I = \{0, 1\}$, $\Sigma_f = \{s_1\}$ iar funcţia de evoluţie este dată de

f	s_0	s_1	s_2
$\flat$	$(s_2, 1, 1)$		$(s_0, 0, -1)$
0	$(s_1, 0, 1)$		
1	$(s_0, 1, 1)$	$(s_0, 0, 1)$	

Evoluţia maşinii pentru $p = 001$ este

$$(s_0, 011, 1) \vdash\!\!\longrightarrow (s_1, 011, 2) \vdash\!\!\longrightarrow (s_0, 001, 3) \vdash\!\!\longrightarrow (s_0, 001, 4) \vdash\!\!\longrightarrow$$

$$(s_2, 0011, 5) \vdash\!\!\longrightarrow (s_0, 0011, 4) \vdash\!\!\longrightarrow (s_0, 00110, 5) \vdash\!\!\longrightarrow (s_1, 00110, 6)$$

Se poate observa că după citirea întregului cuvânt maşina poate să efectueze un număr de paşi suplimentari până la ajungerea într-o stare finală, deci este posibil ca $|p| < |q|$.

Situaţii în care o maşină Turing se blochează (în aceste situaţii cuvântul scris pe bandă nu este recunoscut):

1. MT ajunge într-o stare s, în dreptul dispozitivului de citire–scriere se află simbolul i şi $(s, i) \notin (\Sigma \times I)'$;

2. MT este în starea s, a citit simbolul din prima poziţie i şi $f(s, i) = (s', i', -1)$;

3. MT efectuează un ciclu infinit în interiorul cuvântului.

Definiţie 6.1 *Vom spune că o maşină Turing este nestaţionară dacă*

$$f : (\Sigma \times I)' \to \Sigma \times (I \setminus \{\flat\}) \times \{-1, +1\}.$$

Prin urmare, o maşină Turing nestaţionară nu lasă în nici o situaţie banda pe loc.

Lema 6.1 *Orice maşină Turing este echivalentă cu o maşină Turing nestaţionară.*

Demonstraţie. Pornind de la o MT dată construim o MT' nestaţionară astfel:
Dacă $f(s, i) = (s', i', \pm 1)$ vom pune $f'(s, i) = f(s, i)$;
Dacă $f(s, i) = (s', i', 0)$ vom pune $f'(s, i) = (s'', i', -1)$ şi $f'(s'', j) = (s', j, +1), \forall j \in I$, unde s'' este o stare nou introdusă.

Astfel, în cazul unei rămâneri pe loc a lui MT, noua maşină va face un pas spre stânga şi unul spre dreapta, fără să modifice nici starea şi nici conţinutul benzii. Evident, cele două maşini sunt echivalente.

Limbajele recunoscute de maşini Turing.

Teorema 6.2 *Un limbaj este recunoscut de o maşină Turing dacă şi numai dacă este de tipul zero.*

Demonstraţie. Partea I. $E = L(MT) \Rightarrow E \in \mathcal{L}_0$.

Fie $MT = (\Sigma, I, f, s_0, \Sigma_f)$ o maşină Turing astfel încât $E = L(MT)$. Putem presupune că MT este nestaţionară. Fie $I_\lambda = (I \setminus \{\flat\}) \cup \{\lambda\}$.

Construim o gramatică de tipul zero astfel: $G = (V_N, V_T, S, P)$ unde

$$V_N = \Sigma \cup \{(i,j) | i \in I_\lambda, j \in I\} \cup \{S, X_1, X_2\};\ V_T = I \setminus \{\flat\}.$$

Definiţia lui P:

$S \to s_0 X_1,\ \ X_1 \to (i,i)X_1,\ i \in I \setminus \{\flat\}$,

$X_1 \to X_2,\ X_2 \to (\lambda, \flat)X_2,\ X_2 \to \lambda$,

dacă $f(s,i) = (s', i', 1)$ atunci $s(i_1, i) \to (i_1, i')s',\ \forall i_1 \in I_\lambda$,

dacă $f(s,i) = (s', i', -1)$ atunci $(i_1, i_2)s(i_3, i) \to s'(i_1, i_2)(i_3, i'),\ i_1, i_3 \in I_\lambda,\ i_2 \in I$,

dacă $s \in \Sigma_f$, atunci $s(i_1, i_2) \to s i_1 s$ şi $(i_1, i_2)s \to s i_1 s$.

Fie $p \in L(MT), p = i_1 \dots i_n$. Presupunem că MT utilizează în recunoaştere m poziţii de pe bandă situate la dreapta cuvântului p. În G avem

$$S \Rightarrow s_0 X_1 \overset{*}{\Rightarrow} s_0(i_1, i_1) \dots (i_n, i_n) X_2 \overset{*}{\Rightarrow} s_0(i_1, i_1) \dots (i_n, i_n)(\lambda, \flat)^m.$$

Cuvântul p fiind recunoscut de MT avem

$$(1)\ \ (s_0, i_1 \dots i_n, 1) \overset{*}{\longmapsto} (s_f, i'_1 \dots i'_h, k),\ h \geq n.$$

Vom arăta că (1) implică existenţa unei derivări de forma

$$2) s_0(i_1, i_1) \dots (i_n, i_n)(\lambda, \flat)^m \overset{*}{\Rightarrow} (i_1, i'_1) \dots (i_{k-1}, i'_{k-1}) s_f (i_k, i'_k) \dots (i_{n+m}, i'_{n+m}),$$

unde

$$i_1, \dots, i_n \in I \setminus \{\flat\},\ i_{n+1}, \dots, i_{n+m} = \lambda,$$
$$i'_1, \dots, i'_h \in I \setminus \{\flat\},\ i'_{h+1}, \dots, i'_{n+m} = \lambda.$$

Demonstraţie prin inducţie asupra lungimii l a evoluţiei.

Pentru $l = 0$ avem

$$(s_0, i_1 \dots i_n, 1) \overset{*}{\longmapsto} (s_0, i_1 \dots i_n, 1), k = 1, h = n, i'_j = i_j.$$

Partea dreaptă a lui 2) va avea forma $s_0(i_1, i_1) \dots (i_n, i_n)(\lambda, \flat)^m$ şi deci 2) este adevărată.

Presupunem că implicaţia este adevărată pentru l oarecare şi considerăm o evoluţie de lungime $l + 1$. Punem în evidenţă ultima evoluţie directă

$$(s_0, i_1 \dots i_n, 1) \overset{*}{\longmapsto} (s, i''_1 \dots i''_g, j) \longmapsto (s_f, i'_1 \dots i'_h, k).$$

În general $h = g$ sau $h = g + 1$ în conformitate cu următoarele două cazuri (figura 6.2).

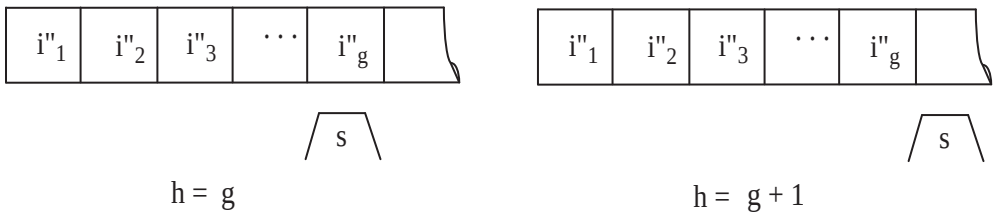

Figura 6.2: Configuraţii posibile ale maşinii Turing

Întotdeauna $k = j \pm 1$. apoi $i''_t = i'_t$, $t = 1, \ldots, g, t \neq j$, adică

$$i''_1, i''_2, \ldots, i''_{j-1}, i''_j, i''_{j+1}, \ldots, i''_g;$$
$$i'_1, i'_2, \ldots, i'_{j-1}, i'_j, i'_{j+1}, \ldots, i'_g.$$

În urma ultimei evoluţii directe vor diferi numai simbolurile i''_j, i'_j.

Din ipoteza inductivă rezultă

$$s_0(i_1, i_1) \ldots (i_n, i_n)(\lambda, \flat)^m \stackrel{*}{\Rightarrow} (i_1, i''_1) \ldots (i_{j-1}, i''_{j-1}) s (i_j, i''_j) \ldots (i_{n+m}, i''_{n+m}).$$

În conformitate cu definiţia evoluţiei directe avem

$$f(s, i''_j) = (s_f, i'_j, 1),\ k = j + 1, s(i_j, i''_j) \to (i_j, i''_j) s_f;$$
$$f(s, i''_j) = (s_f, i'_j, -1),\ k = j - 1, (i_{j-1}, i''_{j-1}) s (i_j, i''_j) \to s_f (i_j, i''_{j-1})(i_j, i'_j).$$

În ambele cazuri

$$s_0(i_1, i_1) \ldots (i_n, i_n)(\lambda, \flat)^m \stackrel{*}{\Rightarrow} (i_1, i'_1) \ldots (i_{k-1}, i'_{k-1}) s_f (i_k, i'_k) \ldots (i_{n+m}, i'_{n+m}).$$

Acum, deoarece $s_f \in \Sigma_f$, putem scrie

$$\begin{aligned} S \ &\stackrel{*}{\Rightarrow} s_0(i_1, i_1) \ldots (i_n, i_n)(\lambda, \flat)^m \\ &\stackrel{*}{\Rightarrow} (i_1, i'_1) \ldots (i_{k-1}, i'_{k-1}) s_f (i_k, i'_k) \ldots (i_{m+n}, i'_{m+n}) \\ &\Rightarrow (i_1, i'_1) \ldots (i_{k-1}, i'_{k-1}) s_f i_k s_f (i_{k+1}, i'_{k+1}) \ldots (i_{m+n}, i'_{m+n}) \\ &\Rightarrow (i_1, i'_1) \ldots (i_{k-2}, i'_{k-2}) s_f i_{k-1} s_f i_k s_f i_{k+1} s_f (i_{k+2}, i'_{k+2}) \ldots (i_{m+n}, i'_{m+n}) \\ &\stackrel{*}{\Rightarrow} s_f i_1 s_f i_2 s_f \ldots s_f i_n s_f \stackrel{*}{\Rightarrow} i_1 \ldots i_n = p. \end{aligned}$$

Prin urmare $p \in L(G)$ şi $L(MT) \subseteq L(G)$. Analog se arată şi incluziunea inversă şi deci $L(MT) = L(G)$.□

Bibliografie

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

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

3. J. E. Hopcroft şi J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Reading Mass., 1979.

4. Solomon Marcus, *Gramatici şi automate finite*, Editura Academiei, Bucureşti, 1964.

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

6. Gheorghe Orman, *Limbaje formale*, Editura tehnică, Bucureşti, 1982.

7. Gheorghe Păun, *Probleme actuale în teoria Limbajelor formale*, Editura Ştiinţifică şi Enciclopedică, Bucureşti, 1984.

8. Teodor Rus, *Mecanisme formale pentru specificarea limbajelor*, Editura Academiei, Bucureşti, 1983.

9. Arto Salomma, *Formal languages*, Academic Press, New York, 1973.

10. Dan Simovici, *Limbaje formale şi tehnici de compilare*, Editura didactică şi pedagogică, Bucureşti, 1978.

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