1 INTRODUCERE - ORGANIZAREA UNUI COMPILATOR 2
1.1 Analiza lexicala 4
1.2 Analiza sintactică 4
1.3 Analiza semantică 5
1.4 Generarea de cod intermediar 5
1.5 Optimizarea codului intermediar 5
1.6 Generarea codului obiect 6
1.7 Optimizarea codului obiect 6
1.8 Gestiunea tabelei de simboluri 6
1.9 Detectarea erorilor 6
2 ELEMENTE DE TEORIA LIMBAJELOR FORMALE 7
2.1 Gramatici 7
2.1.1 Ierarhia Chomsky 8
2.1.1.1 Exerciţii 9
2.1.2 Verificarea limbajului generat de către o gramatică 10
2.1.2.1 Exerciţii 11
2.1.3 Transformări asupra gramaticilor independente de context 11
2.1.3.1 Eliminarea ambiguităţii 12
2.1.3.2 Eliminarea λ- producţiilor 16
2.1.3.3 Eliminarea recursivităţii stânga 16
2.1.3.4 Factorizare stânga 18
2.1.3.5 Eliminarea simbolilor neterminali neutilizaţi 19
2.1.3.6 Substituţia de începuturi(corner substitution) 20
2.1.4 Construcţii dependente de context 22
2.1.4.1 Exerciţii 23
2.1.5 Proprietăţi ale limbajelor independente de context 24
2.1.5.1 Exerciţii 25
2.2 Mulţimi regulate, expresii regulate. 25
2.3 Acceptoare 28
2.3.1 Automate finite 29
2.3.1.1 Construcţia unui automat finit nedeterminist care acceptă limbajul descris de o expresie regulată dată 30
2.3.1.2 Conversia unui automat finit nedeterminist (AFN) într-un automat finit determinist(AFD) 34
2.3.1.3 Construcţia unui automat finit determinist care acceptă limbajul descris de o expresie regulată dată 37
2.3.1.4 Simularea unui automat finit determinist 43
2.3.1.5 Simularea unui automat finit nedeterminist 44
2.3.1.6 Probleme de implementare pentru automatele finite deterministe şi nedeterministe 45
2.3.1.7 Minimizarea numărului de stări pentru AFD 46

2.3.2 Automate cu stivă (pushdown) ... 49
2.3.2.1 Automate cu stivă cu acceptare prin stivă goală ... 52
2.3.2.2 Relaţia între automate cu stivă şi limbajele independente de context ... 53
2.3.3 Maşina Turing ... 57
2.3.3.1 Calcule realizate de Maşina Turing ... 60
2.3.3.2 Compunerea maşinilor Turing ... 63
2.3.3.3 Extensii pentru maşina Turing ... 67
2.3.3.4 Automate liniar mărginite ... 73
2.3.3.5 Relaţia între maşina Turing şi gramatici ... 74
2.3.3.6 Elemente de calculabilitate ... 77
2.3.3.6.1 Maşina Turing Universală ... 77
2.3.3.7 . Maşina Turing cu limită de timp ... 81

3 2. ANALIZA LEXICALĂ ... 83

3.1 Interfaţa analizorului lexical ... 87

3.2 Un exemplu elementar de analizor lexical ... 89

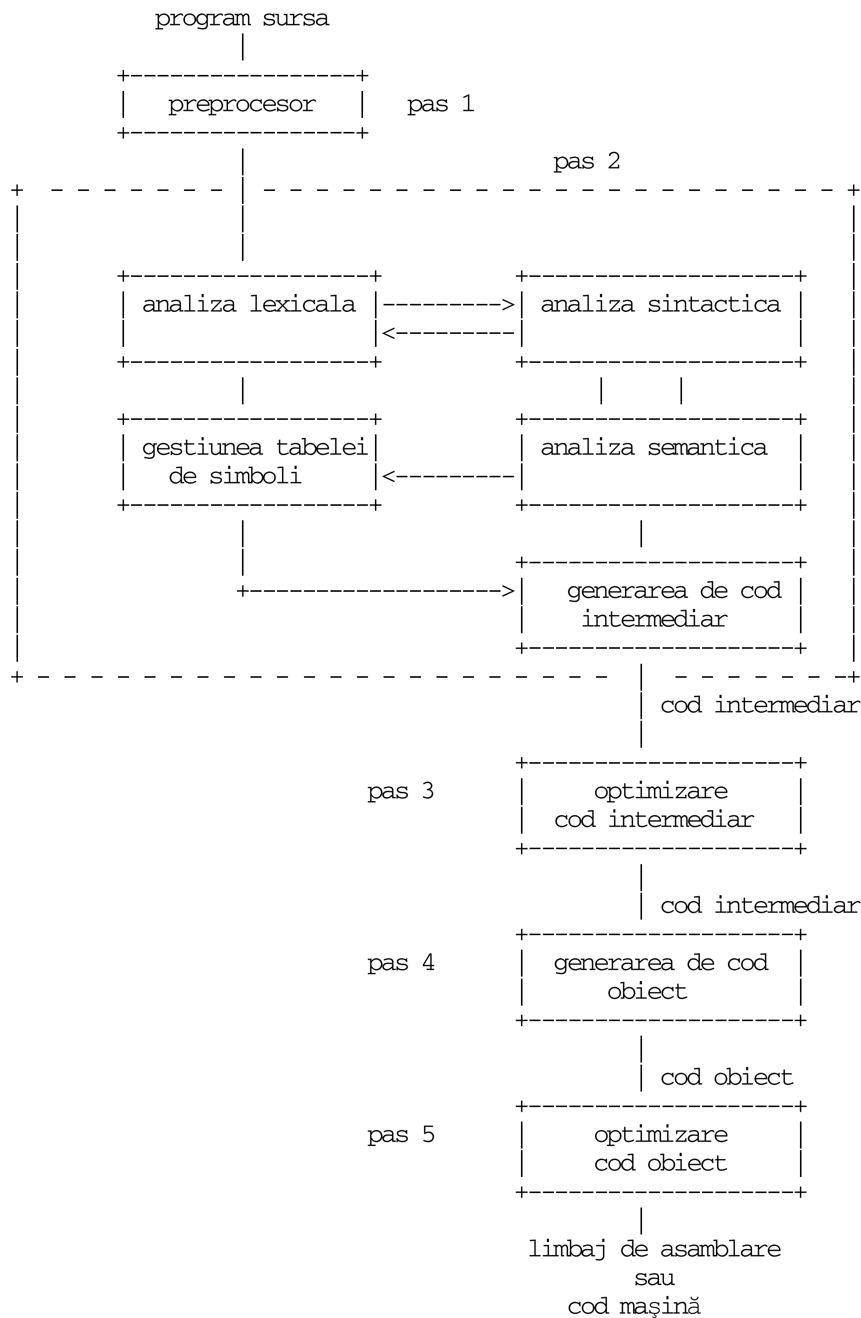
4 ANALIZA SINTACTICĂ ... 99

4.1 Analiza sintactica top - down ... 103
4.1.1 Analiza sintactica predictiva (descendent recursiva) ... 104
4.1.1.1 Gramatici LL(1) ... 109
4.1.1.2 Tratarea erorilor în analiza predictivă ... 115
4.1.2 Analiza sintactica bottom-up ... 117
4.1.2.1 Analiza sintactica de tip deplaseaza şi reduce ... 117
4.1.2.2 Implementarea analizei sintactice bottom-up deplasează şi reduce ... 117
4.1.2.3 Analiza sintactica de tip LR(k) ... 121
4.1.2.3.1 Analiza SLR ... 129
4.1.2.3.2 Analiza canonica LR ... 136
4.1.2.3.3 Analiza sintactică LALR ... 142

1 Introducere - organizarea unui compilator

Un compilator este un program complex care realizează traducerea unui program sursă într-un program obiect. De obicei programul sursă este scris într-un limbaj de nivel superior celui în care este scris programul obiect.

Structura generală pentru un compilator este:



Preprocesorul realizează activităţi de tip macro substituţie, eliminarea comentariilor, etc.

Al doilea pas reprezintă de fapt componenta principală a compilatorului (celelalte componente ar putea să lipsească). Efectul execuţiei acestui pas constă din verificarea corectitudinii formale a textului programului şi traducerea acestui text într-o formă intermediară.

Următorii trei paşi realizează prelucrări asupra programului în cod intermediar în scopul îmbunătăţirii performanţelor acestuia (optimizarea) şi generării programului obiect.

Pasul cel mai complex dintr-un compilator rămâne pasul 2 în care se realizează cele mai importante operaţii fără de care nu poate avea loc procesul de compilare. Într-un compilator real cele cinci componente care îl formează: analiza lexicală, analiza sintactică, analiza semantică, gestiunea tabelei de simboli şi generarea de cod nu sunt neapărat identificabile sub forma unor proceduri ori funcţii distincte ci

sunt realizate printr-un ansamblu de funcții care cooperează. În cele ce urmează aceste componente vor fi descrise separat pentru simplificarea expunerii.

1.1 Analiza lexicala

Această fază a compilatorului realizează traducerea textului programului într-o forma mai uşor de prelucrat de către celelalte componente. Analizorul lexical consideră textul primit la intrare ca fiind format din unități lexicale pe care le recunoaşte producând atomi lexicali. Un atom lexical poate să fie de exemplu, un cuvânt cheie al limbajului (for, while, etc) dar şi un număr sau un nume. Nu există o corespondență biunivocă între şirurile de intrare şi atomii lexicali. Adică, dacă pentru atomul lexical corespunzător cuvântului cheie **while** există un singur şir de intrare, pentru atomul lexical corespunzător unui număr întreg pot să existe foarte multe şiruri de intrare. Una dintre deciziile ce trebuie luate la începutul proiectării unui compilator constă din stabilirea atomilor lexicali. De exemplu, se pune problema dacă să existe câte un atom lexical pentru fiecare operator de comparație (<, <=, >, >=) sau să existe un unic atom lexical - corespunzător operației de comparație. În primul caz generarea de cod poate să fie mai simplă. Pe de altă parte existența unui număr mare de atomi lexicali poate complica în mod exagerat analiza sintactică. În general, operatorii care au aceeaşi prioritate şi asociativitate pot să fie grupați împreună.

Rolul unui analizor lexical este de a traduce şirurile de intrare în atomi lexicali. Un atom lexical este reprezentat printr-un cod numeric care specifica clasa acestuia şi o serie de atribute care sunt specifice fiecărei clase. Astfel, poate să existe clasa operatorilor relaționali pentru care un atribut trebuie să se specifice tipul concret al operatorului. Tipul atomului lexical este necesar pentru analiza sintactică în timp ce valoarea atributului este semnificativă pentru analiza semantică şi generarea de cod. Pentru un atom lexical de tip număr atributele vor descrie tipul numărului şi valoarea acestuia.

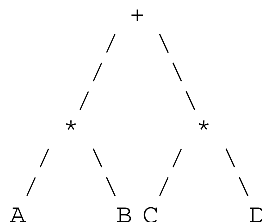
Un analizor lexical apare în general ca o funcție care interacționează cu restul compilatorului printr-o interfață simplă : ori de câte ori analizorul sintactic are nevoie de un nou atom lexical va apela analizorul lexical care îi va da atomul lexical următor.

1.2 Analiza sintactică

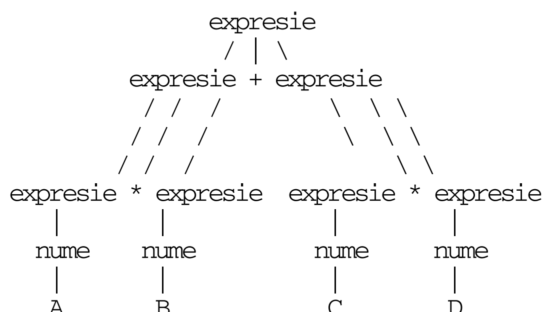
Analiza sintactică descompune textul programului sursa în componentele sale "gramaticale", construind un arbore care reflectă această structură. Să considerăm de exemplu expresia :

A * B + C * D

Această expresie poate să fie descrisă de următorul tip de arbore numit arbore sintactic:



În acest arbore au fost evidențiate relațiile (din punctul de vedere al modului de evaluare) între componentele expresiei. Dacă se doreşte însă să se evidențieze structura expresiei din punctul de vedere al unităților sintactice din care este formată, atunci se va utiliza pentru reprezentarea expresiei un arbore de derivare (parse tree). Pentru exemplul considerat un arbore de derivare ar putea să fie de forma:



Orice analizor sintactic realizează traducerea unui şir de atomi lexicali într-un astfel de arbore de derivare care descrie relaţia ierarhică între o descriere generală a propoziţiei analizate (rădăcina arborelui) şi şirul de atomi lexicali din care este format (frunzele). Un analizor sintactic poate să construiască efectiv o structură de date de tip arbore (cu pointeri şi înregistrări) sau poate să sintetizeze informaţiile din care se poate face construcţia acestuia.

1.3 Analiza semantică

Această fază este de obicei incorporată în faza de analiză sintactică. Rolul acestei faze constă din verificarea din punct de vedere semantic a structurilor recunoscute drept corecte din punct de vedere sintactic. Majoritatea verificărilor realizate de către această fază se referă la tipurile construcţiilor. De asemenea această fază completează arborele de derivare cu o serie de informaţii necesare generării de cod.

1.4 Generarea de cod intermediar

Nici aceasta fază nu este întotdeauna separată de analiza sintactică, există compilatoare care generează cod chiar în timpul analizei sintactice. Generarea de cod se face prin parcurgerea arborelui de derivare. Forma intermediară care se generează reprezintă codul obiect pentru o maşină virtuală. Utilizarea formelor intermediare se justifica prin câteva argumente. În primul rând anumite optimizări nu se pot face în timpul analizei sintactice şi sunt dificil de realizat pe un text de program într-un limbaj de programare de nivel foarte scăzut. De exemplu scoaterea expresiilor constante în afara ciclurilor. Alt motiv este utilizarea aceleaşi faze de optimizare şi generare de cod maşină pentru diferite limbaje de programare, respectiv utilizarea aceleaşi faze de analiza sintactică, semantică şi generare de cod intermediar pentru a implementa acelaşi compilator (limbaj) pe maşini diferite. Cu alte cuvinte pentru a realiza implementări portabile pentru compilatoare. De asemenea utilizarea unei maşini virtuale permite realizarea mai simplă de compilatoare incrementale sau interpretoare performante. Acest tip de translatoare execută direct (interpretează) codul intermediar fără a mai trece prin fazele următoare - editare de legături, încărcare, etc., dar şi fără a recunoaşte de fiecare dată o instrucţiune care a fost deja tratată, cum s-ar întâmpla dacă interpretarea s-ar face la nivel de limbaj sursă.

Dezavantajul utilizării unei forme intermediare constă în mod evident din mărirea timpului necesar pentru execuţia unei compilări.

1.5 Optimizarea codului intermediar

Această fază identifică optimizările posibile asupra codului în limbaj intermediar. De exemplu pentru o secvenţă ca:

```
for(....){
   a[i * c] = b[i * d] + e + f;
   ...
}
```

Se observă că o parte din calcule se pot efectua o singură dată înainte de ciclul for, rezultatele respective se pot memora în variabile temporare, etc. Desigur astfel de optimizări pot să fie făcute şi de către programator. Este de preferat însă să se păstreze claritatea programului, iar acest tip de transformări să fie realizate de către compilator.

1.6 Generarea codului obiect

Această fază depinde de maşina pentru care compilatorul trebuie să genereze cod şi care poate să fie diferită de maşina pe care se execută compilatorul (cazul cross compilatoarelor). Această fază depinde de arhitectura maşinii ţintă. Ideea este că pentru fiecare instrucţiune în limbaj intermediar se va alege o secvenţă echivalentă de instrucţiuni obiect.

1.7 Optimizarea codului obiect

Şi această fază depinde de maşina pentru care se generează cod. Şi anume se identifică secvenţe de cod maşină care pot să fie înlocuite cu instrucţiuni mai rapide.

1.8 Gestiunea tabelei de simboluri

În tabela de simboluri se înregistrează identificatorii utilizaţi în program şi informaţii asupra acestora. Aceste informaţii pot să se refere la tip, domeniu de valabilitate; dacă este vorba de identificatori de tip nume de funcţie la aceste informaţii se adaugă şi signatura funcţiei (numărul şi tipul argumentelor, modul de transfer şi eventual tipul rezultatului). În general o tabelă de simboluri este o structură de date care conţine câte o înregistrare pentru fiecare identificator având câmpuri pentru atributele posibile. Introducerea simbolurilor în tabela se face de către analizorul lexical. Atributele acestora sunt completate în tabela de către analizoarele sintactic şi semantic.

1.9 Detectarea erorilor

Fiecare fază a unui compilator poate să identifice prezenţa unei erori specifice. De exemplu, în analiza lexicală întâlnirea unui şir care nu corespunde unui atom lexical; în analiza sintactica se identifică erori legate de structura instrucţiunilor. Ca de exemplu pentru şirul:

```
int real alfa;
```

fiecare atom lexical în parte este corect dar nu formează împreună o propoziţie corectă din punct de vedere sintactic. În faza de analiză semantică se verifică dacă construcţiile corecte din punct de vedere sintactic sunt corecte şi din punct de vedere semantic. De exemplu dacă la nivelul sintaxei poate să apară ca fiind corecta o expresie de forma: **nume + nume**, fără nici o restricţie asupra tipului identificatorilor corespunzători, este rolul analizei semantice să identifice ca eronată o expresie în care primul nume este al unui vector iar al doilea nume este al unei proceduri.

Problema cea mai dificilă legată de tratarea erorilor constă din modul în care se continuă analiza după identificarea unei erori, pentru că un compilator care se opreşte la prima eroare întâlnită nu este prea comod de utilizat. Excepţie face modul de abordare utilizat în primele versiuni ale compilatorului pentru TURBO Pascal pentru care la întâlnirea unei erori se revine în regim de editare pentru corectarea acesteia.

2 Elemente de teoria limbajelor formale

Fie T o mulţime de simboluri denumita alfabet. Orice submulţime a mulţimii T* reprezintă un limbaj asupra alfabetului T. Elementele limbajului se numesc propoziţii. Dacă limbajul este finit atunci el poate să fie definit prin enumerare. De exemplu considerând alfabetul B = {0, 1} atunci L = {01, 10, 101} este un limbaj. Mulţimea cuvintelor din limbajul natural este şi el un limbaj pentru care se poate pune problema enumerării tuturor cuvintelor, chiar dacă lista care ar rezulta este imensă, deci este un limbaj reprezentabil prin enumerare. Dar cazul interesant este cel în care limbajul este infinit. Să considerăm de exemplu limbajul "şirurilor formate din 0 şi 1 a căror lungime este divizibila cu 3". Evident este vorba de un limbaj infinit. Textul prin care am specificat limbajul constituie o reprezentare finită a limbajului. Nu este singura soluţie posibilă de reprezentare finită. De exemplu dacă notam cu L limbajul respectiv atunci:

```
L = { w ∈ {0,1}* | |w| mod 3 = 0}
```

este un alt mod de a specifica acelaşi limbaj.

Se pune problema dacă dându-se un limbaj oarecare este posibilă întotdeauna construirea unei reprezentări finite. Să considerăm că o astfel de reprezentare finită se realizează utilizând simboli dintr-un alfabet finit A. Se poate demonstra că mulţimea A^* este infinit numărabilă (se poate construi o bijecţie f : $N \rightarrow A^*$). Deci există o mulţime infinit numărabilă de reprezentări finite. Numărul de limbaje ce se pot construi utilizând simboli dintr-un alfabet dat T, este $2^{|T^*|}$ deci mulţimea limbajelor este infinit nenumărabila. Rezultă deci că ar trebui să reprezentăm un număr infinit nenumărabil de obiecte având la dispoziţie numai un număr infinit numărabil de reprezentări. Din acest motiv nu orice limbaj va putea să fie reprezentabil într-un mod finit. Nu putem să oferim un exemplu de limbaj pentru care nu avem o reprezentare finită pentru că exemplul ar fi tocmai o reprezentare finită a limbajului respectiv. Spre norocul nostru, nu suntem interesaţi de toate limbajele ci numai de o clasă mai mică a limbajelor infinite cu reprezentări finite.

În general există doua mecanisme distincte de definire finită a limbajelor: prin generare sau prin recunoaştere. În primul caz este vorba de un "dispozitiv" care ştie să genereze toate propoziţiile din limbaj (şi numai pe acestea) astfel încât alegând orice propoziţie din limbaj într-un interval finit de timp dispozitivul va ajunge să genereze propoziţia respectivă. În al doilea caz este vorba de un "dispozitiv" care ştie să recunoască (să accepte ca fiind corecte) propoziţiile limbajului dat.

2.1 Gramatici

O gramatică reprezintă cel mai important exemplu de generator de limbaje. Prin definiţie o gramatică este G = (N, T, P, S) unde :

- N este o mulţime finită de simboli numită mulţimea simbolilor neterminali;
- T este o mulţime finită de simboli numită mulţimea simbolilor terminali, ($T \cap N = \emptyset$);
- P este o submulţime finită din $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$; numită mulţimea producţiilor gramaticii. Un element $(\alpha, \beta) \in P$ este notat cu $\alpha \rightarrow \beta$ şi se numeşte producţie.
- $S \in N$ este un simbol special numit simbol de start al gramaticii G.

În cele ce urmează vom utiliza o serie de notaţii devenite "clasice". Şi anume :

- literele mici de la începutul alfabetului latin (a,b,c,...) reprezintă elemente din T (simboli terminali);
- literele mici de la sfârşitul alfabetului latin (u, v, x,...) reprezintă elemente din T* (şiruri de simboli terminali);

- literele mari de la începutul alfabetului latin (A, B, C,...) reprezintă elemente din N (simboli neterminali);
- literele mari de la sfârşitul alfabetului latin (U, V, X,...) reprezintă elemente din $N \cup T$ (simboli terminali sau neterminali);
- literele alfabetului grecesc (α, β, ...) reprezintă şiruri din $(N \cup T)^*$ (şiruri de simboli terminali şi neterminali).

O formă propoziţională pentru o gramatică G se defineşte recursiv în modul următor:

(1) S este o formă propoziţională;
(2) dacă $\alpha\beta\delta$ este o forma propoziţională şi există o producţie $\beta \rightarrow \gamma$ atunci $\alpha\gamma\delta$ este o formă propoziţională.

O formă propoziţională care conţine numai simboli terminali se numeşte propoziţie generată de G. Notăm cu L(G) mulţimea tuturor propoziţiilor generate de G altfel spus L(G) este limbajul generat de gramatica G.

Se observă că o gramatică este o reprezentare finită (toate elementele sale sunt finite) pentru un limbaj care poate să fie infinit. Conform observaţiei făcute la începutul acestui capitol nu orice limbaj are o reprezentare finită, cu alte cuvinte nu pentru orice limbaj există o gramatică care să îl reprezinte.

Două gramatici G şi G' sunt echivalente dacă şi numai dacă L(G) = L(G').

Asupra formelor propoziţionale se defineşte o relaţie numită relaţie de derivare $\Rightarrow$ în modul următor. Fie α şi β doua forme propoziţionale, $\alpha \Rightarrow \beta$ dacă şi numai dacă există w1, w2 şi $\gamma \rightarrow \delta \in P$ astfel încât $\alpha = w1\ \gamma\ w2$ şi $\beta = w1\ \delta\ w2$.

Relaţia $\Rightarrow$ poate să fie extinsă obţinându-se derivarea în k paşi. şi anume $\alpha \Rightarrow^k \beta$ dacă există $\alpha_0, \alpha_1, \ldots, \alpha_k$ forme propoziţionale astfel încât $\alpha = \alpha_0$, $\alpha_{i-1} \Rightarrow \alpha_i$,$1 \le i \le k$ şi $\alpha_k = \beta$.

Închiderea tranzitivă a relaţiei $\Rightarrow$ se notează cu $\Rightarrow^+$. Închiderea tranzitivă şi reflexivă a relaţiei $\Rightarrow$ se notează cu $\Rightarrow^*$.

Să considerăm de exemplu gramatica G = ({A,S}, {0,1}, P, S) unde P = {$S \rightarrow 1A1$, $S \rightarrow 0S0$, $1A \rightarrow 11A1$, $A \rightarrow \lambda$} (cu λ s-a notat şirul vid de simboli). O derivare posibilă este:

$$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001A100 \Rightarrow 0011A1100 \Rightarrow 00111100$$

deci 00111100 este o propoziţie în L(G).

În general $L(G) = \{ w \mid w \in T^+, S \Rightarrow^+ w\}$.

2.1.1 Ierarhia Chomsky.

Noam Chomski este lingvist şi lucrează în domeniul limbajelor naturale. Ierarhia care îi poarta numele a rezultat dintr-o încercare a acestuia de a formaliza limbajele naturale. Gramaticile sunt clasificate conform complexităţii producţiilor în următoarea ierarhie :

- gramatici de tip 0 (fără restricţii) - au producţiile de forma:

  $\alpha \rightarrow \beta$ cu $\alpha \in (N \cup T)^* N (N \cup T)^*$, $\beta \in (N \cup T)^*$

- gramatici de tip 1 (dependente de context) - au producţiile de forma :

  $\alpha A \beta \rightarrow \alpha \gamma \beta$, $\alpha, \beta \in (N \cup T)^*$, $A \in N$, $\gamma \in (N \cup T)^+$

sau de forma

$S \to \lambda$

În al doilea caz S nu apare în membrul drept al nici unei producţii. Se utilizează termenul de dependenţă de context deoarece producţia $\alpha\ A\ \beta \to \alpha\ \gamma\ \beta$ poate să fie interpretată sub forma - dacă simbolul neterminal A apare între α şi β atunci poate să fie înlocuit cu γ.

- gramatici de tip 2 (independente de context) - au producţiile de forma :

$A \to \alpha,\ A \in N,\ \alpha \in (N \cup T)^*.$

Denumirea de independent de context apare în contrast cu gramaticile de tipul 1 (dependente de context)

- gramatici de tip 3 (regulate la dreapta) au producţii de forma:

$A \to aB$ cu $A \in N$, $B \in (N \cup \{\lambda\})$ si $a \in T^+$.

Corespunzător gramaticilor, despre limbajele generate de acestea se poate spune respectiv că sunt regulate, independente de context, dependente de context sau de tipul zero. Se poate arata că un limbaj ce poate să fie generat de o gramatică regulată poate să fie generat şi de către o gramatică independentă de context. Un limbaj independent de context poate să fie generat şi de o gramatică dependentă de context iar un limbaj dependent de context poate să fie generat şi de o gramatică de tipul zero. Deoarece cu cât o gramatică este mai restrictivă ea reprezintă un mecanism mai simplu, suntem întotdeauna interesaţi de cea mai restrictivă gramatică care reprezintă un limbaj dat.

Să considerăm câteva exemple:

a) G1 = ({S},{0,1},{S → 0S, S → 1S, S → λ}, S). Se observă că G1 este o gramatică regulată care generează limbajul {0,1}*

b) G2 = ({S, A},{0,1},{S → AS, S → λ, A → λ, A → 0, A → 1}, S). Se observă că G2 este o gramatică independentă de context iar limbajul generat este tot {0,1}*. Rezultă deci că un acelaşi limbaj poate să fie definit de mai multe gramatici diferite eventual chiar de tipuri diferite.

c) G3 = ({∈, T, F},{a, +, *, (,)}, P, ∈) cu P = { E → E + T, E → T, T → T * F, T → F, F → (E), F → a}. Să considerăm un exemplu de derivare în această gramatică :

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a.$

Se observă că gramatica G3 este o gramatică independentă de context şi este o gramatica care descrie limbajul expresiilor aritmetice cu paranteze care se pot forma cu operandul a şi cu operatorii + şi *.

În cele ce urmează pentru simplificarea notaţiilor dacă pentru un neterminal există mai multe producţii : A → w1, A → w2, ... A → wk le vom reprezenta sub o formă mai compactă: A → w1 | w2 | ... | wk.

De asemenea pentru specificarea unei gramatici nu vom mai preciza în general decât mulţimea producţiilor sale, celelalte elemente rezultând în mod banal din aceasta.

2.1.1.1 Exerciţii

Să se construiască gramaticile care generează limbajul:

1. şirurilor formate din simboli a şi b având un număr egal de a şi b.
2. $\{a^n b^n \mid n \geq 1\}$
3. $\{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$
4. $\{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\}$
5. şirurilor formate cu simboli a şi b care nu conţin subşirul abb
6. şirurilor formate cu simboli a şi b având lungimea divizibilă cu 3
7. $\{a^i b^j \mid i \neq j, i, j > 0\}$
8. $\{a^m b^n \mid n < m \text{ sau } n > 2m, n, m \geq 1\}$
9. şirurilor formate dintr-un număr par de simboli a şi un număr impar de simboli b
10. şirurilor formate din simboli a, b şi c, pentru care toţi simboli a apar înainte de toţi simboli b iar toţi simboli b apar înainte de toţi simboli c
11. $\{a^n b^n c^n \mid n >_ 1\}$
12. $\{xcx^R \mid x \in \{a,b\}^*\}$, $\{xx^R \mid x \in \{a,b\}^*\}$, $\{x = x^R \mid x \in \{a,b\}^*\}$
13. $\{xx \mid x \in \{a,b\}^*\}$
14. $\{a^n b^n c^n \mid n \geq 1\}$
15. listelor de elemente care pot să nu conţină nici un element, respectiv trebuie să conţină cel puţin un element, construirea listei este asociativă dreapta respectiv stânga (vor rezulta 4 variante)

2.1.2 Verificarea limbajului generat de către o gramatică

În toate exemplele considerate până acum s-a făcut "ghicirea" gramaticii care generează un limbaj dat sau a limbajului generat de către o gramatică dată. Se pune însă problema cum se poate demonstra corectitudinea rezultatului unei astfel de ghiciri. Să considerăm de exemplu gramatica:

$$G = (\{S\}, \{(,)\}, \{S \rightarrow (S)S \mid \lambda\}, S)$$

Această gramatica generează toate şirurile de paranteze bine închise (echilibrate). Dorim însă să demonstrăm această afirmaţie. De fapt aici trebuie să demonstrăm egalitatea a două mulţimi: mulţimea reprezentată de limbajul generat de G şi mulţimea şirurilor de paranteze bine formate. Deci demonstraţia presupune demonstrarea dublei incluziuni. Adică trebuie să demonstrăm că orice şir derivat din S satisface condiţia enunţată şi apoi trebuie să demonstrăm incluziunea în sens invers. Dându-se un şir de paranteze bine închise trebuie să arătăm că acest şir poate să fie derivat din S. Pentru prima parte a demonstraţiei vom utiliza inducţia asupra numărului de paşi în derivare. Considerăm că şirul vid care se obţine într-un pas din S este un şir de paranteze bine închise. Să presupunem că pentru toate derivările realizate în mai puţin de n paşi se obţin şiruri de paranteze bine închise şi să considerăm o derivare de exact n paşi. O astfel de derivare poate să arate ca :

$$S \Rightarrow (S)S \Rightarrow^* (x)S \Rightarrow^* (x)y$$

unde x şi y sunt şiruri de terminale derivate din S în mai puţin de n paşi, adică sunt şiruri de paranteze bine închise. Rezultă că şirul (x)y este un şir de paranteze bine închise. Cu alte cuvinte orice şir derivat din S este "corect".

Să considerăm acum şi includerea în sens invers. De data asta demonstraţia se face prin inducţie asupra lungimii şirului. Pentru primul pas observăm că şirul vid este un şir derivabil din S. Să presupunem acum ca orice şir cu mai puţin de 2n simboli este derivabil din S. Să considerăm un şir w de paranteze bine închise având lungimea de 2n, cu n mai mare sau egal cu 1. Sigur şirul începe cu o paranteză deschisă. Fie (x) cel mai scurt prefix format din paranteze bine închise. Se observă că w = (x)y, unde x şi y sunt şiruri

de paranteze bine închise de lungime mai mică decât 2n. În acest caz x şi y pot să fie derivate din S. Rezultă că există o derivare:

$S \Rightarrow (S)S \Rightarrow^* (x)y$

în care pentru obţinerea şirurilor x şi respectiv y s-au utilizat mai puţin de 2n paşi şi deci w este un şir derivabil din S. Desigur o astfel de demonstraţie este practic imposibil de realizat pentru un limbaj "adevărat". În general se pot face însă demonstraţii "pe porţiuni".

2.1.2.1 Exerciţii

1. Fie gramatica G : S → AA, A → AAA, a, A → bA, Ab, să se arate ca limbajul L(G) este limbajul tuturor şirurilor formate din simboli a având un număr par de simboli.
2. Fie gramatica G : S → aB | bA, A → a | aS | bAA, B → b | bS | aBB să se arate ca L(G) este setul tuturor şirurilor din $\{a,b\}^+$ care au un număr egal de apariţii pentru a şi pentru b.

2.1.3 Transformări asupra gramaticilor independente de context

Din punctul de vedere al procesului de compilare, gramaticile sunt utilizate pentru faza de analiză sintactică, pentru care se utilizează gramatici independente de context. Există o serie de metode de analiza sintactică, bine puse la punct atât din punct de vedere teoretic cât şi practic. Fiecare dintre aceste metode impune însă o serie de restricţii asupra gramaticilor utilizate. În general atunci când se construieşte o gramatică se pleacă de la forma generală a structurilor pe care aceasta trebuie să le descrie şi nu de la metoda de analiza sintactica ce va fi utilizată. În acest mod se obţine o gramatică ce poate să fie "citită" uşor de către proiectant. Pentru a satisface însă condiţiile impuse de către metodele de analiza sintactică sau de către generarea de cod, se realizează transformări asupra gramaticilor. Aceste transformări trebuie să păstreze neschimbat limbajul generat. În cele ce urmează vom prezenta câteva transformări tipice asupra gramaticilor independente de context. Pentru a explica semnificaţia acestor transformări în contextul analizei sintactice vom prezenta întâi noţiunea de arbore de derivare.

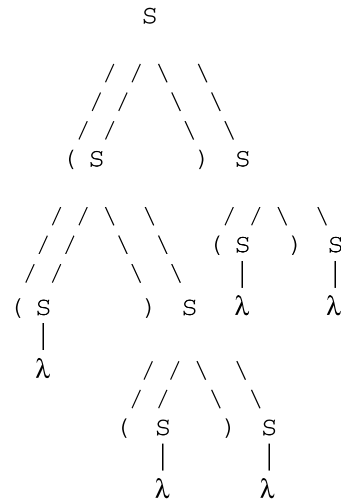
Un arbore de derivare este o reprezentare grafică pentru o secvenţă de derivări (de aplicări ale relaţiei ⇒ între formele propoziţionale). Într-un arbore de derivare nu se mai poate identifica ordinea în care s-a făcut substituţia simbolilor neterminali. Fiecare nod interior arborelui, reprezintă un neterminal. Descendenţii unui nod etichetat cu un neterminal A sunt etichetaţi de la stânga la dreapta prin simbolii care formează partea dreaptă a unei producţii care are în partea stânga neterminalul A. Parcurgând de la stânga la dreapta frunzele unui astfel de arbore se obţine o formă propoziţională. Să considerăm de exemplu din nou gramatica şirurilor de paranteze bine formate:

$G = (\{S\}, \{(,)\}, \{S \rightarrow (S)S \mid \lambda\}, S)$

Fie următoarea secvenţă de derivări:

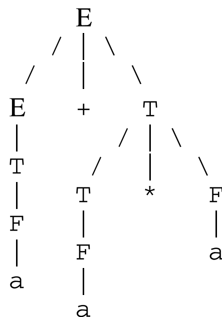
$$S \Rightarrow (S)S \Rightarrow ((S)S)S \Rightarrow (()S)S \Rightarrow$$
$$(()(S)S)S \Rightarrow (()()S)S \Rightarrow$$
$$(()())S \Rightarrow (()())(S)S \Rightarrow^+ (()())()$$

Se obţine arborele de derivare:



Arborele de derivare este construit de către analizorul sintactic. Aceasta construcţie se poate face pornind de la rădăcină aplicând succesiv producţii - în acest caz se spune că analiza sintactică este top-down (descendentă). Dar, se poate porni şi invers de la şirul de atomi lexicali (frunze) identificându-se simbolii neterminali din care se poate obţine un şir de atomi lexicali. În acest caz se spune că analiza sintactică este de tip bottom-up (ascendentă).

Deoarece arborele de derivare descrie relaţia ierarhica între entităţile sintactice (neterminale) şi atomii lexicali (terminale) se poate asocia o interpretare în termeni de evaluare a entităţilor sintactice. Astfel, considerând de exemplu gramatica expresiilor aritmetice pentru şirul a + a * a se obţine arborele derivare :



în care se poate observa că a fost evidenţiat faptul că operaţia de înmulţire este prioritară faţă de operaţia de adunare (aspect semantic).

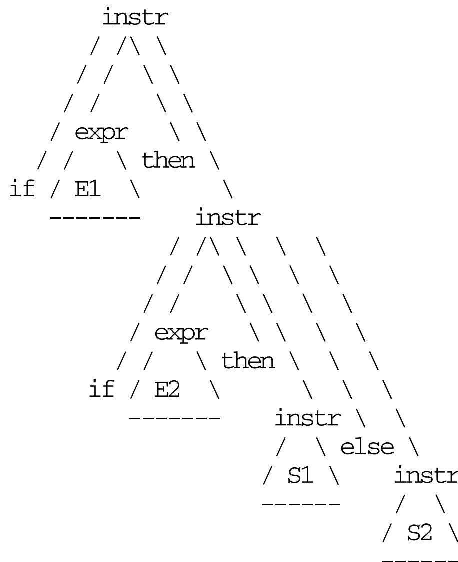
2.1.3.1 Eliminarea ambiguităţii

O gramatică care produce mai mulţi arbori de derivare pentru aceeaşi propoziţie este o gramatică ambiguă. Deoarece există tehnici de analiză sintactică care lucrează numai cu gramatici neambigue vom încerca să construim gramatici care generează acelaşi limbaj şi care sunt neambigue. Să considerăm de exemplu următoarea gramatică :

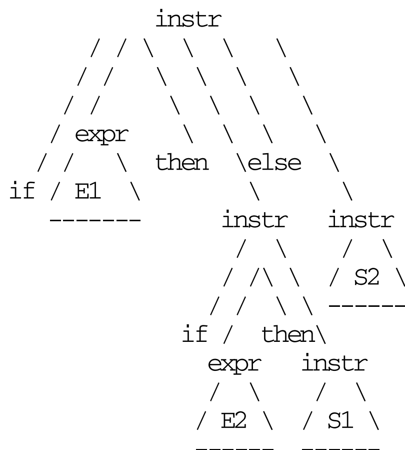
```
instr → if expresie then instr |
if expresie then instr else instr |
alte_instr
```

Să construim arborele de derivare pentru propoziţia :

```
if E1 then if E2 then S1 else S2
```



Pentru această propoziţie mai există însă un arbore de derivare.

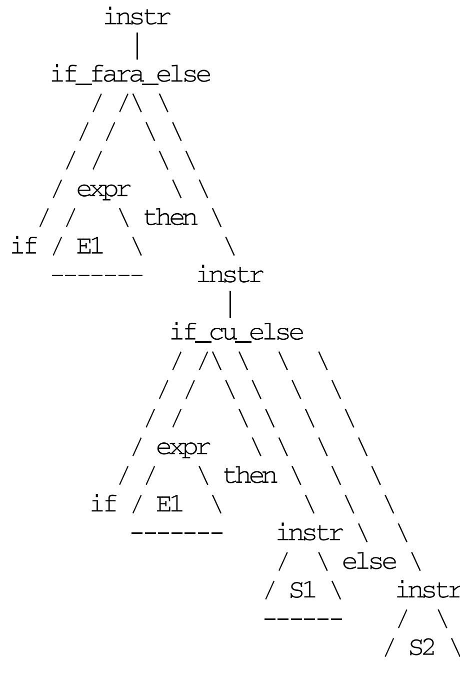


În toate limbajele de programare care acceptă construcţii de tip *if then else* se consideră cu sens prima derivare în care fiecare clauza *else* este atribuită instrucţiunii *if* cea mai interioară. Rezultă deci condiţia pe care trebuie să o satisfacă o instrucţiune *if*. Instrucţiunea cuprinsă între *then* şi *else* trebuie să nu fie o instrucţiune *if* sau să fie o instrucţiune *if* cu clauza *else*. Rezultă următoarea gramatică obţinută prin transformarea gramaticii anterioare:

```
instr → if_cu_else| if_fara_else
if_cu_else → if expresie then if_cu_else else if_cu_else |
             alte_instr
if_fara_else → if expresie then instr |
               if expresie then if_cu_else else if_fara_else
```

Se observă că această gramatică generează acelaşi limbaj cu gramatica anterioară dar acceptă o derivare unică pentru propoziţia :

```
if E1 then if E2 then S1 else S2
```



Se numeşte producţie ambiguă o producţie care are în partea dreaptă mai multe apariţii ale aceluiaşi simbol neterminal. Existenţa unei producţii ambigue nu implică faptul că gramatica este ambiguă. Să considerăm gramatica G = ({S, A}, {a, b, c }, {S → aAbAc, A → a | b}). Se observă că această gramatică nu este ambiguă, gramatica generând limbajul {aabac, aabbc, abac, abbc}

Să considerăm de exemplu şi gramatica pentru expresii aritmetice:

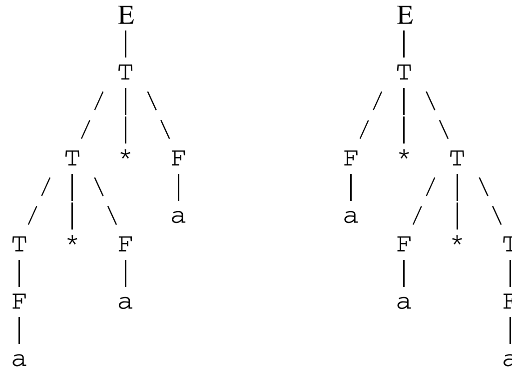
```
G = ({E}, {a, +, *}, {E → E + E | E * E | a}, E)
```

Gramatica G este o gramatică ambiguă (se poate verifica uşor utilizând de exemplu şirul a + a * a). În gramatica G nu au fost evidenţiate relaţiile de precedenţă dintre operatori. Aceasta gramatica poate să fie transformata într-o gramatica neambiguă în care operaţia de înmulţire este mai prioritară decât cea de adunare în doua moduri:

```
1. E → E + T | T
   T → T * F | F
   F → a

2. E → T + E | T
   T → F * T | F
   F → a
```

Fie şirul a * a * a. Pentru cele doua gramatici se obţin arborii de derivare respectivi:



Se observă că primul arbore evidenţiază asociativitatea stânga a operatorului * în timp ce al doilea arbore evidenţiază asociativitatea dreapta. În funcţie de definiţia limbajului este de preferat prima variantă sau a doua.

În cazul general dacă pentru un neterminal A există producţiile:

$$A \rightarrow A\ B\ A \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \quad , B \in N,\ \alpha_1, \ldots \alpha_n \in T^*$$

acestea pot să fie înlocuite cu:

$$A \rightarrow A'\ B\ A \mid A'$$
$$A' \rightarrow A \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Producţia A' → A poate să fie eliminată (există şi A → A') şi se obţine:

$$A \rightarrow A'\ B\ A \mid A'$$

$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

Dacă se construieşte arborele de derivare se observă că în acest caz se utilizează asociativitatea dreaptă. Pentru a se descrie asociativitatea stânga se utilizează transformarea:

$$A \rightarrow A\ B\ A' \mid A'$$
$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n.$$

Trebuie să remarcam însă faptul că există limbaje pentru care nu se pot construi gramatici neambigue. Un astfel de limbaj se numeşte inerent ambiguu. De exemplu limbajul :

$$L = \{ a^i b^j c^k d^l \mid i = k \text{ sau } j = l,\ i, j, k, l \geq 0\}$$

este inerent ambiguu. O gramatică care descrie acest limbaj va trebui probabil să considere că L este de fapt reuniunea a două limbaje:

$$L = \{ a^n b^j c^n d^l \mid n, j, l \geq 0\} \text{ şi } L = \{ a^i b^n c^k d^n \mid i, n, j, k \geq 0\}$$

Un şir de forma $a^p b^p c^p d^p$ va face parte din ambele limbaje şi deci probabil că va avea doi arbori de derivare. Exemplul anterior nu constituie însă o demonstraţie, o demonstraţie "adevărată" depăşeşte ca dificultate cadrul textului curent.

2.1.3.2 Eliminarea λ- producţiilor

Se spune ca o gramatică este λ- free dacă satisface una dintre următoarele condiţii:

a. Nu există nici o producţie care să aibă în partea dreapta şirul vid
b. Există o singura producţie care are în partea dreapta şirul vid şi anume o producţie de forma $S \rightarrow \lambda$. Simbolul S nu apare în acest caz în partea dreaptă a nici unei producţii.

Algoritmul de transformare pleacă de la gramatica G = (N, T, P, S) şi construieşte gramatica G' = (N', T, P', S') care satisface condiţiile :

(i) L(G) = L(G')
(îi) G' este λ- free.

Descrierea algoritmului este :

```
i = 0
Ni = {A | A → λ ∈ P}
repeta
    i = i + 1
    Ni = { A | A → α ∈ P, a ∈ N*i-1} ∪ Ni-1
până Ni = Ni-1
Ne = Ni
dacă S ∈ Ne
    N' = N   {S'}
    P' = {S' → λ, S' → S}
altfel
    N' = N
    S' = S
    P' = ∅
□
pentru fiecare p ∈ P executa
    dacă p este de forma : A → a0 B1 a1 ... Bk ak, k ≥ 0,
          Bi ∈ Ne, 1 ≤ i ≤ k, aj ∈((N \ Ne) ∪ T)*, 0 ≤ j ≤ k
          P' = P' ∪ ({A → a0 X1 a1 ... Xk ak | Xi ∈ {Bi, λ}} \{A → λ})

    altfel
           P' = P' ∪ {p}
    □
□
```

Fie de exemplu gramatica $S \rightarrow aSbS \mid bSaS \mid \lambda$. Aplicând algoritmul anterior se obţine:

$S' \rightarrow S$, $S \rightarrow aSbS \mid aSb \mid abS \mid ab \mid bSaS \mid bSa \mid baS \mid ba$

2.1.3.3 Eliminarea recursivităţii stânga

O gramatică este recursivă stânga dacă există un neterminal A astfel încât există o derivare $A \Rightarrow^{+} A\beta$ pentru $\beta \in (T \cup N)^*$. O analiză sintactică descendentă deterministă nu poate să opereze cu o astfel de gramatică, deci este necesară o transformare. Să considerăm întâi cazul cel mai simplu pentru care în gramatică există producţii de forma $A \rightarrow A\beta \mid \alpha$. În acest caz limbajul generat este de forma $\alpha\beta^n$ cu $n \geq 0$. Acelaşi limbaj poate să fie generat de către gramatica: $A \rightarrow \alpha A'$, $A' \rightarrow \beta A' \mid \lambda$.

Să considerăm de exemplu gramatica expresiilor aritmetice :

$E \rightarrow E + T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow (E) \mid a$

Se observă că pentru un şir de forma a + a * a, examinând numai primul simbol terminal(a) nu este clar cu care dintre producţiile pentru E trebuie să se înceapă derivarea. Aplicând ideea anterioară se obţine :

$E \rightarrow T E',\ E' \rightarrow +TE' \mid \lambda,\ T \rightarrow FT',\ T' \rightarrow *FT' \mid \lambda,\ F \rightarrow (E) \mid a$

În acest caz derivarea va începe sigur prin aplicarea producţiei $E \rightarrow TE'$ şi se obţine derivarea $E \Rightarrow TE' \Rightarrow FT'E'$. În acest moment se vede că pentru F trebuie să se aplice producţia $F \rightarrow a$. Deci se obţine $E \Rightarrow^+ aT'E'$. Urmează simbolul terminal + datorită căruia pentru T' se va aplica producţia $T' \rightarrow \lambda$, etc.

În general dacă pentru un neterminal A există producţiile :

$A \rightarrow A\beta_1 \mid A\beta_2 \mid ... \mid A\beta_m \mid \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

unde γ_i nu începe cu A, $1 \le i \le n$, se pot înlocui aceste producţii cu :

$A \rightarrow \alpha_1A' \mid \alpha_2A' \mid ... \mid \alpha_nA'$
$A' \rightarrow \beta_1A' \mid \beta_2A' \mid ... \mid \beta_mA' \mid \lambda$

Această construcţie elimină recursivitatea stângă imediată. Să considerăm însă gramatica:

$A_1 \rightarrow A_2\ a \mid b$
$A_2 \rightarrow A_3\ c \mid d$
$A_3 \rightarrow A_1\ e \mid f$

cu A_1 simbolul de start al gramaticii. Se observă că este posibilă următoarea derivare:

$A_1 \Rightarrow A_2a \Rightarrow A_3\ ca \Rightarrow A_1\ eca$

deci gramatica este recursivă stânga. Se observă că dacă am considerat o ordine a simbolilor, toate producţiile mai puţin ultima, respectă regula "un simbol neterminal este înlocuit de un şir care începe cu alt simbol neterminal cu un număr de ordine mai mare". Existenţa unei producţii care nu respectă condiţia conduce la apariţia recursivităţii stânga.

Dacă gramatica nu permite derivări de tipul $A \Rightarrow^+ A$ (fără cicluri) şi nu conţine λ - producţii poate să fie transformată în vederea eliminării recursivităţii stânga utilizând următorul algoritm, obţinându-se o gramatică echivalentă fără recursivitate stânga.

```
Se aranjează neterminalele în ordinea A1, ..., An
pentru i = 1 până la n  executa
pentru j = 1 până la i - 1 executa
    înlocuieşte fiecare producţie de forma Ai → Aj β cu
    producţiile Ai → α1 β | α2β | ... | αk β
    unde  Aj  →  α1|α2| ... |αk  sunt  toate  producţiile
    pentru Aj
□
elimină recursivitatea stânga între producţiile Ai
□
```

Să considerăm pasul i. Producţiile $A_k \rightarrow A_l\beta$ care au mai rămas (pentru care $k < i$), au $l > k$. În aceasta iteraţie prin variaţia lui j se ajunge ca pentru orice producţie de forma $A_i \rightarrow A_m \beta$, $m \geq i$. Dacă se elimină recursivitatea directă rămâne $m > i$. Să considerăm de exemplu din nou gramatica :

$A_1 \rightarrow A_2a \mid b, A_2 \rightarrow A_2c \mid A_1d \mid e$

Considerăm pentru neterminale ordinea : A_1, A_2. Pentru A_1 ($i = 1$) nu există recursivitate stânga directă deci nu se face nici o modificare. Pentru $i = 2$, producţia $A_2 \rightarrow A_1d$ se inlocuieşte cu $A_2 \rightarrow A_2ad \mid bd$. Rezultă deci că $A_2 \rightarrow A_2c \mid A_2ad \mid bd \mid e$.

Eliminând recursivitatea stânga se obţine :

$A_1 \rightarrow A_2a \mid b, A_2 \rightarrow bdA_2' \mid eA_2', A_2' \rightarrow cA_2' \mid adA_2' \mid\lambda$

2.1.3.4 Factorizare stânga

Acest tip de transformare este util pentru producerea unei gramatici potrivite pentru analiza sintactică descendentă de tip determinist. Ideea este că dacă nu este clar care dintre producţiile alternative poate să fie aplicată pentru un neterminal se va amâna luarea unei decizii până când s-a parcurs suficient din şirul de intrare pentru a se putea lua o decizie. Să considerăm de exemplu producţiile :

$S \rightarrow A\ b\ S \mid A$
$A \rightarrow B\ c\ A \mid B$
$B \rightarrow a \mid dSd$

Să presupunem că încercăm să construim şirul derivărilor pentru a b a c a pornind de la simbolul de start al gramaticii. Din recunoaşterea simbolului a la începutul şirului nu se poate încă trage concluzia care dintre cele doua producţii corespunzătoare neterminalului S trebuie să fie luata în considerare (abia la întâlnirea caracterului b pe şirul de intrare se poate face o alegere corectă). În general pentru producţia $A \rightarrow \alpha\beta1 \mid \alpha\ \beta2$ dacă se recunoaşte la intrare un şir nevid derivat din α nu se poate ştii dacă trebuie aleasă prima sau a doua producţie. Corespunzător este utilă transformarea: $A \rightarrow \alpha\ A'$, $A' \rightarrow \beta1 \mid \beta2$.

Algoritmul de factorizare funcţionează în modul următor. Pentru fiecare neterminal A se caută cel mai lung prefix α comun pentru două sau mai multe dintre producţiile corespunzătoare neterminalului A. Dacă $\alpha \neq \lambda$ atunci se înlocuiesc producţiile de forma $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \delta$ (unde δ reprezintă alternativele care nu încep cu α) cu :

$A \rightarrow \alpha A' \mid \delta$
$A' \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

A' este un nou neterminal. Se aplică în mod repetat această transformare până când nu mai există două alternative producţii cu un prefix comun pentru acelaşi simbol neterminal.

Reluând exemplul considerat se obţine :

$S \rightarrow AX$
$X \rightarrow bS \mid \lambda$
$A \rightarrow BY$
$Y \rightarrow cA \mid \lambda$
$B \rightarrow a \mid dSd$

Deci în analiza şirului a b a la întâlnirea simbolului b pentru neterminalul Y se va utiliza producţia $Y \rightarrow \lambda$, în acest mod rezultă şirul de derivări :

$S \Rightarrow AX \Rightarrow BYX \Rightarrow aYX \Rightarrow ...$

2.1.3.5 Eliminarea simbolilor neterminali neutilizaţi

Un simbol neterminal neutilizat este un simbol care:

- nu poate să apară într-o formă propoziţională, adică într-un şir derivat din simbolul de start al gramaticii (simbol inaccesibil)
- din care nu poate deriva un şir format numai din simboli terminali (simbol nefinalizabil)
- care apare în formele propoziţionale numai datorită sau împreună cu simboli neterminali ce satisfac una dintre condiţiile anterioare.

Pornind de la o gramatică G = (N, T, P, S) se poate obţine o gramatică fără simboli nefinalizaţi şi care satisface următoarele condiţii:

(i) L(G) = L(G')
(îi) $\forall A \in N, A \Rightarrow^{+} w, w \in T^{*}$.

utilizând următorul algoritm :

```
N0 = ∅
i = 0
repeta
    i = i + 1
    Ni = { A | A → α ∈ P si α ∈ (Ni-1 ∪ T)* } ∪ Ni-1
până Ni = Ni-1
N' = Ni
P' ⊆ P conţine numai producţiile din P care au în partea stânga simboli din N' si în
partea dreapta simboli din N' si T.
```

Prin inducţie asupra numărului de paşi se poate demonstra corectitudinea algoritmului. Să considerăm ca exemplu o gramatică având producţiile : $P = \{S \rightarrow A, A \rightarrow a, B \rightarrow b, B \rightarrow a\}$, se observă că B este un simbol neterminal inaccesibil. Aparent condiţia de inaccesibilitate pentru un neterminal constă din ne apariţia în partea dreaptă a unei producţii. Dacă însă considerăm o gramatică având producţiile: $\{S \rightarrow A, A \rightarrow a, B \rightarrow cC, C \rightarrow bB\}$ se observă că este necesară o altă condiţie.

Pornind de la o gramatică G = (N, T, P, S) se poate construi o gramatică fără simboli inaccesibili G' = (N', T, P', S) care să satisfacă condiţiile:

(i) L(G) = L(G')

(îi) $\forall A \in N', \exists w \in (N \cup T)^*, S \Rightarrow w$ şi A apare în w

utilizând următorul algoritm.

```
N0 = {S}
i = 0
repeta
    i = i + 1;
    Ni = {A | A  apare  în  partea dreapta  a  unei  producţii
    pentru un neterminal din Ni-1} ∪ Ni-1
până Ni = Ni-1
N' = Ni
P' conţine numai producţii care corespund neterminalelor din
N'  si  care conţin în partea  dreapta  simboli  neterminali numai din N'
```

Prin inducţie asupra numărului de paşi se poate determina corectitudinea algoritmului. Utilizând algoritmii pentru eliminarea simbolilor nefinalizaţi şi cel pentru eliminarea simbolilor inaccesibili se obţine o gramatică care nu conţine simboli neutilizaţi. Ordinea în care se aplică aceşti algoritmi nu este indiferentă. Să considerăm de exemplu gramatica cu producţiile:

$S \rightarrow a \mid A, A \rightarrow AB, B \rightarrow b.$

Dacă se aplică întâi algoritmul pentru eliminarea simbolilor nefinalizaţi, rămân producţiile $S \rightarrow a$ şi $B \rightarrow b$. Prin eliminarea simbolilor inaccesibili rămâne numai producţia $S \rightarrow a$. Dacă însă se aplică întâi algoritmul pentru eliminarea simbolilor inaccesibili şi apoi cel pentru eliminarea simbolilor nefinalizaţi vor rămâne până la sfârşit producţiile $S \rightarrow a$ şi $B \rightarrow b$, adică nu se obţine o gramatică fără simboli neutilizaţi. Rezultatul obţinut nu este întâmplător în sensul că nu se poate găsii un exemplu pentru care ordinea corectă de aplicare a celor doi algoritmi să fie inversă. Ideea este că prin eliminarea unui simbol neterminal neaccesibil nu pot să apară simboli neterminali nefinalizaţi, în timp ce prin eliminarea unui simbol neterminal nefinalizat pot să apară simboli neterminali inaccesibili deoarece anumiţi simboli neterminali puteau să fie accesibili numai prin intermediul simbolului neterminal respectiv.

2.1.3.6 Substituţia de începuturi(corner substitution)

Anumite metode de analiză sintactică impun ca partea dreaptă a fiecărei producţii care nu este şirul vid să înceapă cu un terminal. Să considerăm de exemplu gramatica având producţiile:

```
lista → a(număr) lista | *elem lista | a
elem  → a(număr) | *elem
```

în acest caz dacă pe şirul de intrare se găseşte terminalul *a* nu este clar care dintre producţiile pentru neterminalul *lista* trebuie să fie utilizată.

Dacă factorizăm producţiile neterminalului *lista*:

```
lista → aX | *elem lista
X     → (număr) lista | λ
elem  → a(număr) | *elem
```

Se observă ca în acest caz în funcție de simbolul terminal curent se poate decide în mod determinist care este producția următoare ce trebuie să fie aplicată pentru derivare (construirea arborelui de derivare).

O gramatică independentă de context pentru care este îndeplinită condiția ca partea dreaptă a oricărei producții începe cu un terminal sau este şirul vid se numeşte gramatică de tip Q. O formă particulară de gramatica de tip Q este forma normală Greibach. În acest caz nu există λ--producții cu excepția cel mult a unei λ--producții corespunzătoare simbolului de start al gramaticii. În cazul în care aceasta producție există simbolul de start al gramaticii nu apare în partea dreaptă a nici unei producții. În forma normală Greibach producțiile sunt de forma $A \rightarrow a\alpha$ cu $a \in T$ şi $\alpha \in N^*$. Să presupunem că o gramatică are producțiile:

$$P \rightarrow a_1 \alpha_1 \mid a_2 \alpha_2 \mid ... \mid a_n \alpha_n \mid \lambda$$

unde $a_i \in T$, $i \neq j \Rightarrow a_i \neq a_j$, $1 \leq i, j \leq n$. O procedură care recunoaşte şirurile derivate din P este de forma:

```
p(){
    switch (caracter_următor){
        case a1 : avans(); a1  /* tratare a1 */
        case a2 : avans(); a2
        ...
        case an : avans(); an
    default:  /* ? - producție */
    }
}
```

Pe baza transformărilor anterioare se poate elabora un algoritm pentru construirea unei gramatici în forma normală Greibach pentru o gramatică independentă de context care nu este recursivă stânga.

Se defineşte întâi o relație de ordine asupra neterminalelor unei gramatici. şi anume A < B dacă există o producție $A \rightarrow B\ \alpha$. Considerând această relație se observă că se poate construi o relație parțială de ordine care poate să fie extinsa la o relație liniară de ordine. Dacă gramatica pentru care se construieşte această relație nu este recursivă dreapta atunci producțiile celui "mai mare" neterminal încep sigur cu simboli terminali. Rezultă algoritmul:

```
Se construieşte relația de ordine asupra N astfel încât
    N =    {A1, A2, ..., An} si A1 < A2 < ... < An.
pentru i = n - 1 până la 1 pas = -1 executa
    fiecare  producție de  forma Ai → Aj α cu i <  j   se înlocuieşte cu Ai → β1 α | ... |
         βm α unde Aj →  β1 |  ...|  βm  (se observa ca  β1,  ...,  βm  încep  cu terminale)
□
pentru fiecare  producție  de forma p = A →a  X1  ...  Xk executâ
    pentru i = 1 până la  k execută
        dacă Xi ∈ T
            N = N ∪ {Xi'}
            P = P ∪ { Xi' → Xi}
        □
    □
    P = P \ {p} ∪ {A → aX1'X2' ... Xk'}
□
```

Prin inducție asupra numărului de paşi se poate demonstra că algoritmul realizează transformarea dorită. De exemplu pentru gramatica expresiilor aritmetice în forma fără λ--producții:

```
E  → T E' | T ,
E' → +TE' | +T,
T  → FT'  | F,
T' → *FT' | *F
F  → (E)  | a
```

Relația de ordine liniară poate să fie E' < E < T' < T < F. Se porneşte de la F, pentru care toate producțiile încep cu un terminal. Rezultă modificarea producțiilor pentru T:

```
T → (E) T' | a T' | (E) | a.
```

Urmează T' care nu necesită transformări. Pentru E se obține:

```
E → (E)T'E' | a T' E' | (E)E' | aE' | (E)T' | aT' | (E) | a
```

De asemenea E' nu se modifică. Rezultă următoarea gramatică în forma normală Greibach:

```
E  → (EAT'E' | a T' E' | (EAE' | aE' | (EAT' | aT' | (EA | a
E' → +TE' | +T
T  → (EA T' | a T' | (EA | a.
T' → *FT' | *F
F  → (EA | a
A  → )
```

2.1.4 Construcții dependente de context

Anumite construcții specifice limbajelor de programare nu pot să fie descrise prin limbaje independente de context. Să considerăm câteva exemple :

1. Fie limbajul L1 = $\{wcw| \ w \in \{0,1\}^*\}$. Acest limbaj realizează abstractizarea condiției ca un identificator să fie declarat înainte de a fi utilizat. L1 nu poate să fie generat de o gramatică independentă de context. Din acest motiv condiția ca un identificator să fie definit înainte de a fi utilizat nu pot să fie verificată prin analiză sintactică, deci va fi verificată de către analiza semantică.
2. Fie limbajul L2 = $\{a^nb^mc^nd^m \mid n > 1, m > 1\}$. Acest limbaj realizează abstractizarea corespondenței ca număr între numărul de parametrii cu care au fost declarate procedurile şi numărul de parametrii cu care se utilizează acestea. Nici acest limbaj nu pot fi descris de o gramatică independentă de context. Pe de altă parte limbajul L2' = $\{a^nb^n \mid n > 1\}$, poate să fie descris de gramatica : S → a S b | a b. Deci şi aici după o primă recunoaştere făcută în analiza sintactică, partea de dependență de context va fi rezolvată de către faza de analiză semantică.

Observație

În general interpretând gramaticile ca “algoritmi” de generare de şiruri cărora le corespund ca echivalenți algoritmi de recunoaştere de şiruri, se constată că pentru cazul limbajelor regulate este necesar ca pentru recunoaştere să se facă o numărare până la o valoare finită (care poate însă să fie oricât de mare). Astfel că limbajul LR = $\{a^nb^n \mid 0 \leq n \leq 20000000000000\}$ este un limbaj regulat. Numărul de *a*-uri întâlnite poate să fie memorat în neterminalele. De altfel, LR este un limbaj finit şi deci automat este regulat (producțiile gramatici pot să reprezinte de fapt lista şirurilor care formează limbajul). Limbajul

LR1 = {w ∈ {0,1}* | |w| mod 3 = 0} este infinit şi regulat. Se observă că pentru recunoaşterea şirurilor este necesar să se numere câte două elemente. În schimb limbajul LI = {$a^n b^n$ | 0 ≤ n} nu este regulat, algoritmul pe care s-ar baza recunoaşterea şirurilor nu are o limită. Tratarea unei astfel de situaţii se face în mod natural utilizând un contor infinit sau o stivă.

2.1.4.1 Exerciţii

1. Fie gramatica:

   A → BC | a. B → CA | Ab, C → AB | cC | b

   Să se construiască gramatica echivalenta nerecursiva stânga

2. Să se elimine simbolii neutilizaţi pentru gramatica:

   S → A | B, A → aB | bS | b, B → AB | BA, C → AS|b

3. Să se construiască gramatica echivalenta fără λ-producţii pentru gramatica:

   S → ABC, A → BB | λ, B → CC | λ, C → AA | b

4. Să se construiască un algoritm care verifică dacă pentru o gramatică dată G, L(G) este mulţimea vidă. Să se demonstreze că algoritmul este corect

2.1.5 Proprietăți ale limbajelor independente de context

Limbajele formale sunt mulțimi - putem să aplicăm asupra lor operații caracteristice mulțimilor : reuniune, intersecție, diferență dar şi operații specifice cum este de exemplu concatenarea. Dacă L1 şi L2 sunt două limbaje, concatenarea lor este un limbaj L = L1L2 astfel încât:

$$L = \{ w \mid w = xy, x \in L1, y \in L2\}$$

O altă operație specifică ce se poate aplica asupra unui limbaj formal L este "Kleen star". Notația utilizată pentru rezultat este L^* şi reprezintă mulțimea şirurilor obținute prin concatenarea a zero sau mai multe şiruri din L (concatenarea a zero şiruri este şirul vid, concatenarea unui singur şir este el însuşi etc.). Se va utiliza şi notația L^+ pentru LL^*.

Dacă aplicând o operație asupra oricăror doua limbaje independente de context obținem un limbaj independent de context vom spune că mulțimea limbajelor independente de context este închisă sub operația respectivă. Se poate demonstra că mulțimea limbajelor independente de context este închisă sub operațiile: reuniune, concatenare şi Kleen star.

Să demonstrăm de exemplu închiderea sub operația de reuniune. Fie doua gramatici G1 = (N1, T1, P1, S1), G2 = (N2, T2, P2, S2). Putem întotdeauna să presupunem că $N1 \cap N2 = \emptyset$ eventual prin redenumirea simbolilor neterminali din una dintre gramatici). Gramatica corespunzătoare limbajului reuniune a limbajelor generate de către cele doua gramatici se poate construi în modul următor:

$$G = (N, T, P, S), N = N1 \cup N2 \cup \{S\}, T = T1 \cup T2, P = P1 \cup P2 \cup \{S \rightarrow S1 \mid S2\}$$

Pentru a demonstra că limbajul $L(G) = L(G1) \cup L(G2)$ trebuie să demonstrăm includerea mulțimilor în ambele sensuri. Dacă $w \in L(G)$ înseamnă că a fost obținut printr-o derivare ca $S \Rightarrow S1 \Rightarrow^* w$ sau $S \Rightarrow S2 \Rightarrow^* w$. Deci un şir derivat din S aparține limbajului $L(G1) \cup L(G2)$. Invers dacă alegem un şir $w \in L(G1) \cup L(G2)$ el este obținut dintr-o derivare din S1 sau din S2, deci se poate construi o derivare pentru acest şir şi în G. Construcții similare se pot face şi pentru operația de concatenare şi pentru operația Kleen star. De exemplu L(G1)* pentru o gramatica G1 = (N1, T1, P1, S1) poate să fie generat de către:

$$G = (N1, T1, P1 \cup \{S1 \rightarrow \lambda \mid S1\ S1\}, S1)$$

Clasa limbajelor independente de context nu este închisă la operațiile de intersecție şi complementare. Să considerăm de exemplu limbajele $L1 = \{a^nb^nc^m \mid n,m \geq 0\}$, $L2 = \{a^mb^nc^n \mid n,m \geq 0\}$.

Limbajul $L = L1 \cap L2$ este "cunoscut" ca fiind un limbaj dependent de context. Să presupunem acum că pentru un limbaj independent de context generat de către gramatica G = (N, T, P, S), limbajul complementar, adică T* - L(G) este întotdeauna independent de context. În acest caz, fie L un limbaj obținut prin intersecția a două limbaje independente de context: L1 şi L2. Se poate scrie:

$$L = L1 \cap L2 = T^* - ((T^* - L1) \cup (T^* - L2)).$$

Dacă mulțimea limbajelor independente de context este închisă la operația de complementare atunci $(T^* - L1)$ şi $(T^* - L2)$ sunt limbaje independente de context şi la fel este şi reuniunea lor şi la fel va fi şi complementarea limbajului obținut prin reuniune. Adică ar rezulta că $L1 \cap L2$ este un limbaj independent de context ceea ce deja ştim că nu este adevărat.

După cum s-a constatat acelaşi limbaj poate să fie descris de mai multe gramatici, eventual de tipuri diferite. Este utilă găsirea unui criteriu prin care să se indice tipul restrictiv al gramaticilor care pot să descrie un limbaj dat, într-o manieră mai precisă decât observația de la pagina 24. De exemplu să considerăm limbajul:

$\{ a^{n * n} | n > 1 \}$

Se pune problema dacă există o gramatica independentă de context care să-l genereze. Următorul rezultat ne dă un criteriu cu ajutorul căruia putem să demonstrăm că o gramatică nu este independentă de context.

Lema de pompare Fie L un limbaj independent de context. Atunci, există o constanta k care este caracteristică limbajului astfel încât dacă $z \in L$ şi $|z| \geq k$, atunci z poate să fie scris sub forma z = uvwxy cu $vx \neq \lambda$, $|vwx| \leq k$, $|w| \geq 1$ şi pentru orice i, $uv^iwx^iy \in L$.

Să utilizăm rezultatul anterior pentru a studia limbajul:

$L = \{ a^{n * n} | n > 1 \}$

să presupunem că L este independent de context. Înseamnă că există un număr k (asociat limbajului) astfel încât dacă $n * n \geq k$ atunci $a^{n \times n} = uvwxy$ astfel încât v şi x nu sunt amândouă şiruri vide şi $|vwx| \leq k$. Să presupunem ca n este chiar k (k * k > k). Rezultă ca $uv^2wx^2y \in L$. Deoarece $|vwx| \leq k$ rezultă $1 \leq |vx| < k$ deci lungimea şirului iniţial $k \times k < |uv^2wx^2y| < k \times k + k$. După k x k următorul pătrat perfect este (k + 1) x (k + 1) care este mai mare deci k x k + k, adică $|uv^2wx^2y|$ nu poate să fie un pătrat perfect, deci L nu poate să fie generat de o gramatică independentă de context.

2.1.5.1 Exerciţii

1. Să se arate că limbajul $\{ a^nb^nc^n | n > 1 \}$ nu este independent de context
2. Să se arate că limbajul $\{ a^nb^nc^j | n > 1, j < n \}$ nu este independent de context

2.2 Mulţimi regulate, expresii regulate.

Fie T un alfabet finit. Se numeşte mulţime regulată asupra alfabetului T mulţimea construită recursiv în modul următor:

1. $\emptyset$ este o mulţime regulată asupra alfabetului T;
2. Dacă $a \in T$ atunci {a} este o mulţime regulată asupra alfabetului T;
3. Dacă P şi Q sunt mulţimi regulate atunci $P \cup Q$, PQ, P^* sunt mulţimi regulate asupra alfabetului T.
4. Nimic altceva nu este o mulţime regulată.

Se observă deci că o submulţime a mulţimii T^* este o mulţime regulată asupra alfabetului T dacă şi numai dacă este mulţimea vida, este o mulţime care conţine un singur simbol din T sau poate să fie obţinută din aceste două tipuri de mulţimi utilizând operaţiile de reuniune ($P \cup Q$), concatenare (PQ) sau închidere P*.

Descrierea mulţimilor regulate se poate face cu ajutorul expresiilor regulate. O expresie regulată este la rândul ei definită recursiv în modul următor (prin analogie cu definiţia mulţimilor regulate) :

1. λ este o expresie regulata care descrie mulţimea $\emptyset$;
2. $a \in T$ este o expresie regulată care descrie mulţimea {a};
3. dacă p,q sunt expresii regulate care descriu mulţimile P şi Q atunci :
   a. (p + q) sau (p | q) este o expresie regulată care descrie mulţimea $P \cup Q$;
   b. (pq) este o expresie regulată care descrie mulţimea PQ;
   c. (p)* este o expresie regulată care descrie mulţimea P^*.
4. nimic altceva nu este o expresie regulată.

De exemplu expresia (0 | 1)* reprezintă mulţimea {0,1}*, expresia regulată : (00 | 11)*((01 | 10)(00 | 11)*(01 | 10)(00 | 11)*)* reprezintă mulţimea care conţine toate şirurile formate din 0 şi 1 şi care conţin un număr par din fiecare simbol. În particular pp^* este notat cu p^+. Între operatorii utilizaţi în descrierea mulţimilor regulate există o serie de relaţii de precedenţă. Cea mai prioritară operaţie este operaţia de închidere (notată cu *), urmează operaţia de concatenare, cel mai puţin prioritară este operaţia + (|). De exemplu, expresia regulată (0 | (1(0)*)) poate să fie scrisa şi sub forma 0 | 10*.

Expresiile regulate au o serie de proprietăţi. Fie α, β, γ, expresii regulate. Spunem ca $\alpha=\beta$ dacă şi numai dacă α şi β descriu aceleaşi mulţimi regulate. Sunt adevărate următoarele proprietăţi :

1. $\alpha \mid \beta = \beta \mid \alpha$ 2. $\alpha \mid (\beta \mid \gamma) = (\alpha \mid \beta) \mid \gamma$
3. $\alpha(\beta\gamma) = (\alpha\beta)\gamma$ 4. $\alpha(\beta \mid \gamma) = \alpha\beta \mid \alpha\gamma$
5. $(\alpha \mid \beta)\gamma = \alpha\gamma \mid \beta\gamma$ 6. $\alpha? = ?\ \alpha = \alpha$
7. $\alpha^* = \alpha \mid \alpha^*$ 8. $(\alpha^*)^* = \alpha^*$
9. $\alpha \mid \alpha = \alpha$

Utilizând expresii regulate se pot construi şi rezolva ecuaţii având ca soluţii expresii regulate. Să considerăm de exemplu ecuaţia:

$$X = aX + b$$

unde a,b şi X sunt expresii regulate. Se poate verifica uşor că $X = a * b$ este o soluţie a acestei ecuaţii. Dacă *a* este o expresie regulată care poate să genereze şirul vid atunci soluţia prezentată anterior nu este unică. Să înlocuim în ecuaţie pe *X* cu expresia regulată:

$$a^*(b + \gamma)$$

Se obţine:

$$\begin{aligned} a^*(b + \gamma) &= a[a^*(b + \gamma)] + b = \\ &= a^+ b + a^+ \gamma + b = \\ &= (a^+ + \lambda)b + a^+\gamma = a^*b + a^*\gamma \\ &= a^*(b + \gamma) \end{aligned}$$

Se numeşte punct fix al unei ecuaţii cea mai mică soluţie a ecuaţiei respective.

Propoziţie Dacă G este o gramatică regulată atunci L(G) este o mulţime regulata.

Demonstraţia acestei teoreme se face prin construcţie. Nu vom face demonstraţia dar vom considera un exemplu de construcţie. Fie gramatica G = ({A, B, C}, {0,1}, P,A) cu P = {$A \to 1A \mid 0B$, $B \to 0B \mid 1C$, $C \to 0B \mid 1A \mid \lambda$}. Se cere să se construiască expresia regulată care generează acelaşi limbaj ca şi G. Se pot scrie următoarele ecuaţii:

$$A = 1A + 0B$$
$$B = 0B + 1C$$

$C = 0B + 1A + \lambda$

Din prima ecuaţie se obţine: $A = 1^*0B$. Din a doua ecuaţie se obţine: $B = 0^*1C$. Înlocuind în A se obţine: $A = 1^*00^*1C = 1^*0+1C$. Înlocuind în a treia ecuaţie se obţine:

$C = 0^+1C + 1^+0^+1C + \lambda$
$C = (\lambda + 1^+)0^+1C + \lambda = 1^*0+1C$
$C = (1^*0^+1)*$

Rezultă:

$A = 1^*0^+1\ (1^*0^+1)^*$
$A = (1^*0^+1)^+$

Mulţimea regulată descrie şirurile de 0 şi 1 care se termină cu sub şirul 01. Evident o expresie mai simplă pentru descrierea aceluiaşi limbaj este:

(0 | 1)*01

Se observă că se poate considera că limbajul se obţine prin concatenarea a doua limbaje unul descris de expresia $(0 \mid 1)^*$ şi celălalt descris de expresia 01. Rezultă următoarea gramatică:

$S \rightarrow AB$
$A \rightarrow 0A \mid 1A \mid \lambda$
$B \rightarrow 01$

Se observă că gramatica obţinută nu este regulată. Să transformăm această gramatică prin substituţie de începuturi:

$S \rightarrow 0AB \mid 1AB \mid B$
$A \rightarrow 0A \mid 1A \mid \lambda$
$B \rightarrow 01$

Se obţine:

$S \rightarrow 0S \mid 1S \mid 01$

Utilizând factorizarea:

$S \rightarrow 0X \mid 1S$
$X \rightarrow S \mid 1$

Din nou prin înlocuire de începuturi se obţine:

$X \rightarrow 0X \mid 1S \mid 1$

Aplicând factorizarea:

$X \rightarrow 0X \mid 1Y$
$Y \rightarrow S \mid \lambda$

Rezultă gramatica descrisă de producţiile:

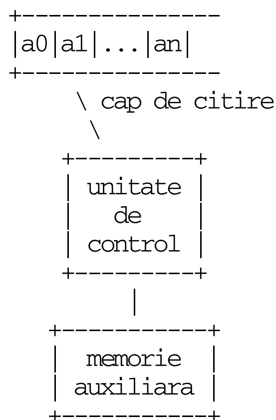
$S \to 0X \mid 1S$
$X \to 0X \mid 1Y$
$Y \to 0X \mid 1S \mid \lambda$.

Se observă că s-a ajuns la aceeaşi gramatică cu cea iniţială (desigur dacă redenumim neterminalele).

Expresiile regulate reprezintă o metodă alternativă generativă pentru definirea limbajelor regulate. Limbajele definite de gramatici regulate sunt utilizate la nivelul analizei lexicale, în majoritatea limbajelor de programare modul de scriere al numelor şi al identificatorilor pot să fie descrise de gramatici respectiv de expresii regulate. Construirea expresiilor regulate corespunzătoare atomilor lexicali reprezintă prima etapa în proiectarea unui analizor lexical.

2.3 Acceptoare

Un acceptor este un dispozitiv format dintr-o bandă de intrare, o unitate de control şi o memorie auxiliară.



Banda de intrare este formata din elemente în care se înscriu simbolii din şirul de intrare. La un moment dat capul de citire este fixat pe un simbol. Funcţionarea dispozitivului este dată de acţiunile acestuia. Într-o acţiune capul de citire se poate deplasa la stânga sau la dreapta sau poate să rămână nemişcat. În cadrul unei acţiuni unitatea de control are acces la informaţia din memoria auxiliară pe care o poate modifica. Se observă ca acest model este foarte general. Utilizând diferite restricţii asupra benzii de intrare, organizării şi accesului la memoria auxiliară se obţin diferite cazuri particulare de acceptoare.

O acţiune a acceptorului este formata din :

- deplasarea capului de citire la stânga sau la dreapta sau menţinerea capului de citire pe aceeaşi poziţie şi / sau;
- memorarea unei informaţii în memoria auxiliară şi / sau;
- modificarea stării unităţii de control.

Comportarea unui acceptor poate să fie descrisă în funcţie de configuraţiile acestuia. O configuraţie este formată din următoarele informaţii :

- starea unităţii de control;
- conţinutul benzii de intrare şi poziţia capului de citire;

- conţinutul memoriei.

Rezultă că o acţiune a acceptorului poate să fie precizată prin configuraţia iniţială (înainte de acţiune) şi cea finală (după execuţia acţiunii).

Dacă pentru o configuraţie dată sunt posibile mai multe acţiuni atunci spunem că acceptorul este nedeterminist altfel este determinist.

Un acceptor are o configuraţie iniţială în care unitatea de control este într-o stare iniţială, capul de citire este fixat pe simbolul cel mai din stânga al şirului de intrare iar memoria are un conţinut iniţial specific. Acceptorul are o configuraţie finală pentru care capul de citire este situat pe simbolul cel mai din dreapta de pe banda de intrare. Se spune că dispozitivul a *acceptat* un şir de intrare *w* dacă pornind din configuraţia iniţială ajunge în configuraţia finala. Eventual, noţiunea de acceptare poate să fie condiţionata şi de ajungerea într-o anumită stare a unităţii de control sau de un anumit conţinut al memoriei auxiliare. Evident dacă acceptorul este nedeterminist dintr-o configuraţie iniţială pot avea loc mai multe evoluţii. Dacă există între acestea una pentru care se ajunge la o configuraţie finală atunci se spune că dispozitivul a acceptat şirul de intrare.

Limbajul acceptat de către un acceptor este format din mulţimea şirurilor acceptate de către acesta.

Pentru fiecare tip de gramatica din ierarhia Chomsky există o clasa de acceptoare care definesc aceeaşi clasă de limbaje. Dintre acestea cele utilizate pentru implementarea compilatoarelor sunt automatele finite care sunt acceptoare pentru limbaje regulate şi automatele cu stivă (push-down), care sunt acceptoare pentru limbajele independente de context. Automatele finite sunt acceptoare fără memorie auxiliară iar automatele cu stivă sunt acceptoare pentru care accesul la memoria auxiliară se face conform unui mecanism de tip stivă.

2.3.1 Automate finite

Un automat finit este un obiect matematic AF = (Q, T, m, s0, F) unde :

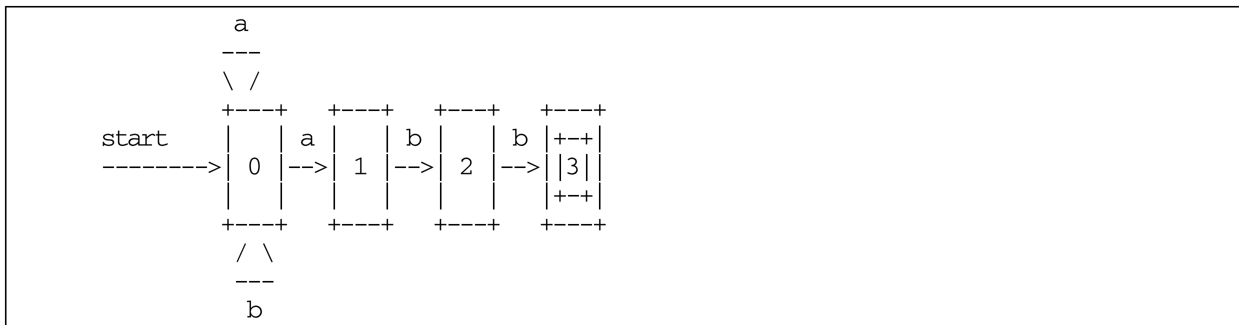
- Q este mulţimea finită a stărilor;
- T este o mulţime finită numită alfabet de intrare;
- m : Q x (T $\cup$ {λ})$\rightarrow$ P(Q) este funcţia parţială a stării următoare;
- s0 $\in$ Q este o stare numită stare de start;
- F $\subseteq$Q este o mulţime de stări numită mulţimea stărilor finale (de acceptare).

Dacă pentru q $\in$ Q, a $\in$ T m(q,a) este definită, atunci m(q,a) se numeşte tranziţie, dacă a = λ atunci m(q,a) se numeşte λ-tranziţie. Se observă ca pentru o combinaţie stare(q $\in$ Q), intrare(a $\in$ T) pot să corespundă mai multe stări următoare, deci aşa cum a fost definit, automatul este nedeterminist (AFN).

Pentru reprezentarea unui automat finit se pot utiliza două moduri de reprezentare, printr-o tabela de tranziţie sau printr-un graf. Să considerăm de exemplu automatul care accepta limbajul definit de expresia (a | b)* abb. Poate să fie descris sub forma următoarei tabele :

	simbol de intrare	
stare	a	b
0	{0, 1}	{0}
1	-	{2}
2	-	{3}
3	-	+

Se observă că în fiecare intrare este specificată mulţimea stărilor următoare în care se trece pentru starea şi simbolul corespunzător intrării respective. Acelaşi automat finit poate să fie descris de următorul graf de tranziţie :



Se observă că pentru fiecare stare există câte un nod, între două noduri există un arc etichetat cu un simbol de intrare dacă şi numai dacă se poate trece din starea reprezentată de primul nod în starea reprezentată de al doilea nod la aplicarea la intrare a simbolului cu care este etichetat arcul. De remarcat modul în care a fost specificată starea finală.

Se spune că un automat finit acceptă un şir de intrare dacă există o cale în graful de tranziţie între nodul care reprezintă starea de start şi un nod care reprezintă o stare finală, astfel încât şirul simbolilor care etichetează arcele care formează această cale este şirul de intrare. De exemplu şirul *aabb* este acceptat de automatul descris anterior, care va executa următoarele mişcări:

$$\begin{array}{l} \quad a \quad\; a \quad\; b \quad\; b \\ 0 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \end{array}$$

Se observa că pentru acelaşi şir de intrare există şi alte secvenţe de intrări care însă nu duc într-o stare de acceptare:

$$\begin{array}{l} \quad a \quad\; a \quad\; b \quad\; b \\ 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \rightarrow 0 \end{array}$$

Un caz particular al automatelor finite este dat de automatele finite deterministe (AFD). În cazul automatelor finite deterministe definiţia funcţiei stării următoare se modifică:

- $m : S \times T \rightarrow S$.

Se observă că sunt satisfăcute următoarele restricţii:

1. nu există λ-tranziţii;
2. pentru $\forall$ (q, a) $\in$ Q x T este definită cel mult o tranziţie.

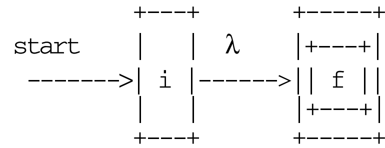
Se observă ca în acest caz, pentru un şir de intrare dat, în graful de tranziţie există cel mult o cale care porneşte din starea de start şi duce într-o stare de acceptare.

2.3.1.1 Construcţia unui automat finit nedeterminist care acceptă limbajul descris de o expresie regulată dată

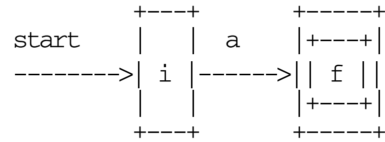
Algoritmul pe care îl vom prezenta utilizează direct structurile din definiţia expresiilor regulate. Se pleacă de la o expresie regulată *r* asupra unui alfabet T şi se obţine un automat finit nedeterminist (N) care acceptă limbajul L(r).

Pentru construirea automatului se identifică în structura expresiei **r** structurile care în mod recursiv compun expresia şi se construiesc automatele finite nedeterministe (AFN) elementare corespunzătoare.

1. Pentru expresia λ, automatul care acceptă limbajul corespunzător este :

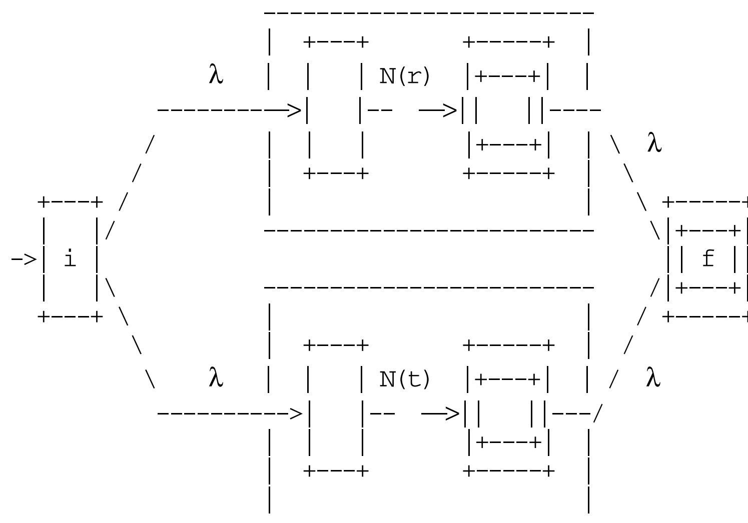


2. Pentru expresia *a*, automatul care acceptă limbajul corespunzător este :

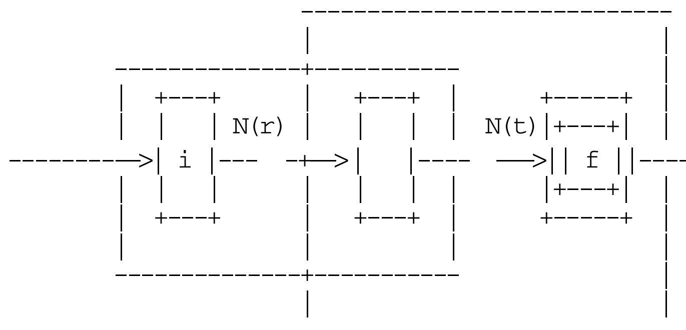


3. Fie două expresii regulate *r,t* pentru care s-au construit automatele finite nedeterministe corespunzătoare N(r) şi N(t).

a. pentru expresia regulată r|t limbajul corespunzător este acceptat de :

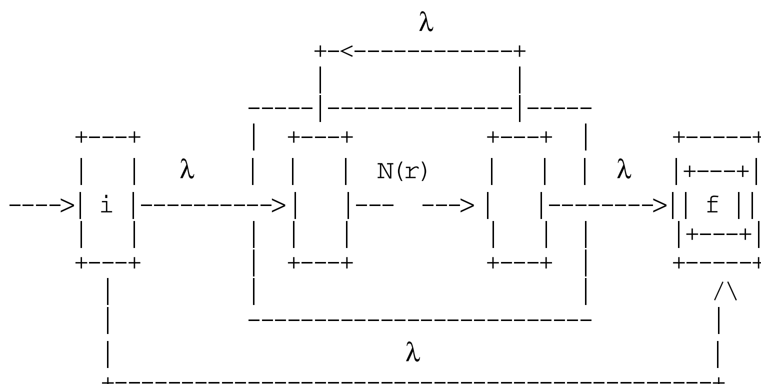


b. pentru expresia regulată *rt* limbajul corespunzător este acceptat de :



Se observă ca în acest caz starea finală a automatului corespunzător expresiei r coincide cu starea inițială a automatului corespunzător expresiei t.

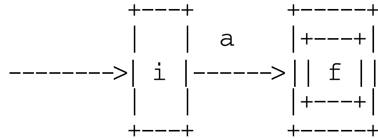
c. pentru expresia regulata *r** limbajul corespunzător este acceptat de :



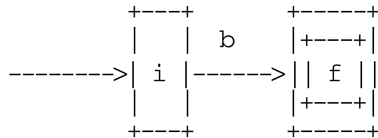
d. pentru expresia (r) limbajul corespunzător este acceptat de automatul care accepta N(r).

Se observă că pentru fiecare AFN - elementar se adaugă cel mult o stare iniţială şi o stare finală. Corespunzător pentru o expresie regulată dată automatul va avea un număr de stări egal cu maxim dublul numărului de simboli de intrare care apar în expresie. Compunerea acestor automate elementare va produce evident un automat care recunoaşte limbajul generat de expresie. De exemplu pentru expresia regulata : *(a|b)*abb* se obţine :

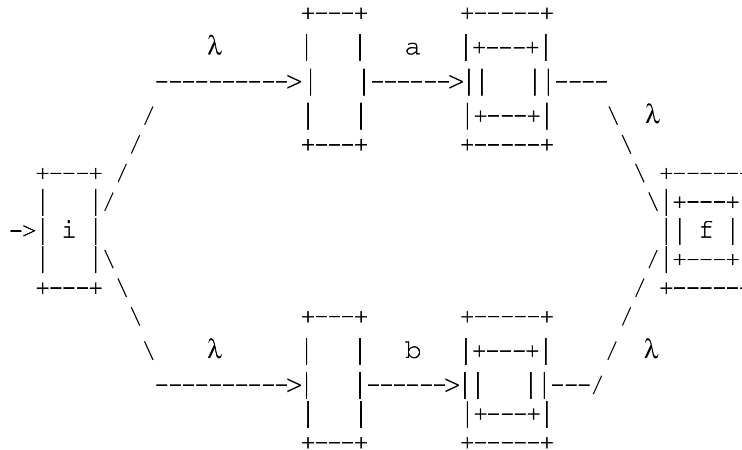
pentru fiecare *a* din expresie :



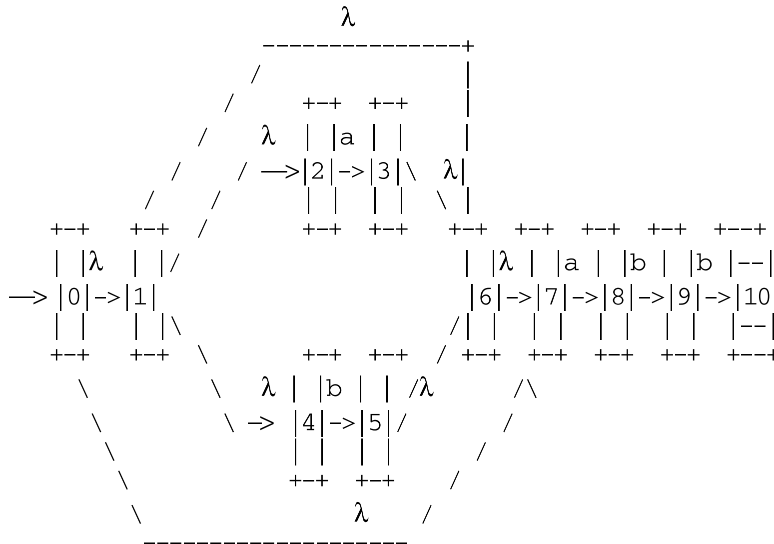
pentru fiecare b din expresie :



pentru a | b :



În final se obţine :



2.3.1.2 Conversia unui automat finit nedeterminist (AFN) într-un automat finit determinist(AFD)

Din exemplele anterioare s-a observat că un acelaşi limbaj reprezentat de expresia regulată (a | b)* abb poate să fie recunoscut de două automate diferite - unul nedeterminist şi unul determinist.

Propoziţie Pentru orice automat finit nedeterminist există un automat finit determinist care acceptă acelaşi limbaj.

Având în vedere această echivalenţă între cele două automate se pune problema, cum se poate construi un automat determinist echivalent unui automat nedeterminist dat. În cele ce urmează vom considera că automatul finit nedeterminist a fost construit pornind de la o expresie regulată pe baza construcţiei prezentate în capitolul anterior. Algoritmul care construieşte automatul determinist utilizează o metodă de construire a submulţimilor unei mulţimi date. Diferenţa între automatele deterministe şi cele nedeterministe este dată de cardinalitatea funcţiei m. Automatul determinist se va construi calculând în mod recursiv stările acestuia, simulând în paralel toate evoluţiile posibile pe care le poate parcurge automatul nedeterminist. Starea iniţială a automatului finit determinist va corespunde mulţimii stărilor din automatul nedeterminist în care se poate ajunge pornind din starea iniţiala s0 şi utilizând numai λ-tranziţii. Orice stare s' a automatului determinist corespunde unei submulţimi $Q' \subseteq Q$ a automatului nedeterminist. Pentru aceasta stare şi un simbol de intrare $a \in T$,

$$m(s',a) = \{s \in Q \mid (\exists\, q \in Q')(s = m(q,a))\}.$$

Rezultă deci că în funcţionarea AFD care simulează de fapt funcţionarea AFN se urmăresc simultan toate "căile" pe care poate evolua automatul finit nedeterminist. Algoritmul care construieşte automatul finit determinist echivalent cu un automat nedeterminist dat utilizează două funcţii : *λ-închidere* şi *mutare*. Funcţia λ-închidere : $P(Q) \rightarrow P(Q)$ determină pentru fiecare mulţime $Q' \subset Q$ setul stărilor în care se poate ajunge datorită λ-tranziţiilor. Considerând o stare *q* a automatului finit determinist (AFD), aceasta reprezintă de fapt o submulţime $Q' \subseteq Q$ a automatului nedeterminist. Notăm cu

$$\lambda\text{-închidere}(Q') = \bigcup_{s \in Q'} \lambda\text{-închidere}(\{s\})$$

unde dacă $s \in S$ este o stare care nu are λ-tranziţii atunci:

$$\lambda\text{-închidere}(\{s\}) = \{s\}$$

altfel

$$\lambda\text{-închidere}(\{s\}) = \bigcup_{s' \in m(s,\lambda)} \lambda\text{-închidere}(\{s'\}) \cup \{s\}$$

Construcţia funcţiei λ-închidere pentru o mulţime Q' se poate face utilizând următorul algoritm :

```
A = Q', B = ∅
cât timp A \ B ≠ ∅ execută
   fie t ∈ A \ B
   B = B ∪ {t}
   pentru fiecare u ∈ Q astfel încât m(t,λ) = u executa
          A = A ∪ {u}
   □
□
λ-închidere(Q') = A
```

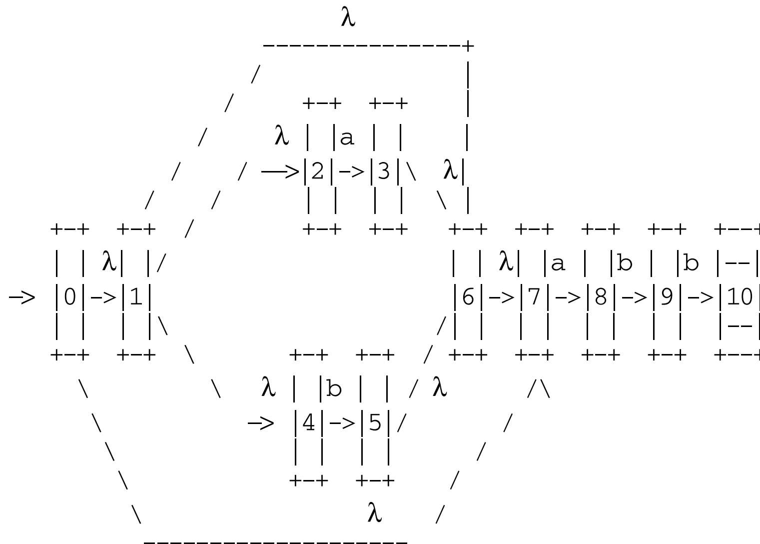
Funcţia *mutare* : $P(Q) \times T \rightarrow P(Q)$ este utilizată pentru construcţia stării următoare a automatului determinist. Astfel pentru o stare *q* a automatului determinist, căruia îi corespunde o mulţime $Q' \subseteq Q$, şi o intrare $a \in T$,

$$\text{mutare}(Q',a) = \bigcup_{s \in Q'} m(s,a)$$

Construcţia automatului determinist se face prin construcţia unei tabele de tranziţii *tranz_AFD[]* şi a unei mulţimi de stări *stari_AFD*.

```
stari_AFD = {λ-inchidere({s0})}
A = ∅
cât timp stari_AFD \ A ≠ ∅ execută
    fie t ∈ stari_AFD \ A
    A = A ∪ {t}
    pentru fiecare a ∈ T execută
            B = λ-inchidere(mutare(t,a))
            stari_AFD = stari_AFD ∪ {B}
            tranz_AFD[t,a] = B
    □
□
```

Fie automatul nedeterminist :

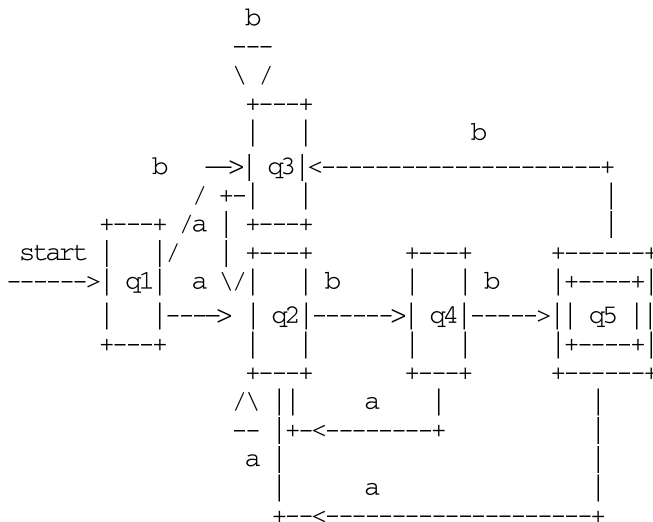


Se observă că

λ-inchidere(0) = {0, 1, 2, 4, 7}, λ-inchidere(mutare({0,1,2,4,7}),a)) = λ-inchidere({3,8}) = {1,2,3,4,6,7,8}. Dacă q1 = {0,1,2,4,7}, q2 = {1,2,3,4,6,7,8} atunci tran_AFD(q1,a) = q2. Continuând se obține următoarea tabelă de tranziție :

stare	intrare a	intrare b	multime corespunzatoare AFN
q1	q2	q3	{0,1,2,4,7} λ-inchidere({0})
q2	q2	q4	{1,2,3,4,6,7,8} λ-inchidere({3,8})
q3	q2	q3	{1,2,4,5,6,7} λ-inchidere({5})
q4	q2	q5	{1,2,4,5,6,7,9} λ-inchidere({5,9})
q5	q2	q3	{1,2,4,5,6,7,10} λ-inchidere({5,10})

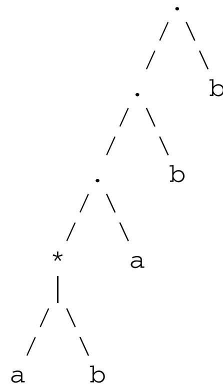
Graful automatului finit determinist care a rezultat este:



Se observă că s-a obţinut un automat cu 5 stări faţă de 4 stări cât avea automatul finit determinist cu care s-au exemplificat automatele finite deterministe. Deci algoritmul utilizat nu produce neapărat o soluţie optimă.

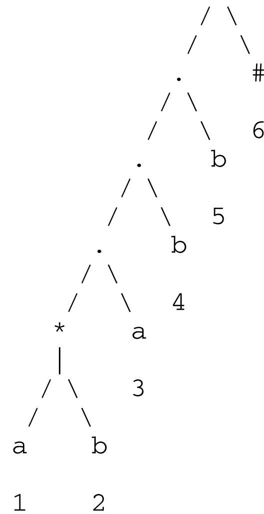
2.3.1.3 Construcţia unui automat finit determinist care acceptă limbajul descris de o expresie regulată dată

O expresie regulată poate să fie reprezentată sub forma unui arbore (sintactic). De exemplu pentru expresia regulata *(a|b)*abb* rezultă arborele:



Se observă că în nodurile interioare apar operatori iar frunzele sunt reprezentate de către simbolii de intrare. Există trei operatori (|, *, .). Pentru a simplifica construcţiile considerăm pentru orice expresie regulata şi un simbol de sfârşit pe care îl vom nota cu # astfel încât orice expresie regulata r va fi reprezentată de *r#*. Reluând pentru expresia anterioară va rezulta graful:

.



Vom ataşa fiecărei frunze un cod unic, de exemplu acest cod poate să fie reprezentat de poziția simbolului respectiv în expresia regulată.

Fie *C* mulțimea codurilor ataşate fiecărei frunze şi *N* mulțimea nodurilor arborelui. Fie c funcția de codificare şi c^{-1} funcția inversă (c : N → C). Definim patru funcții:

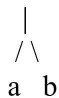
nullable : N → {adevărat, fals}

firstpos : N → *P*(C)

lastpos : N → *P*(C)

followpos : C → P(C)

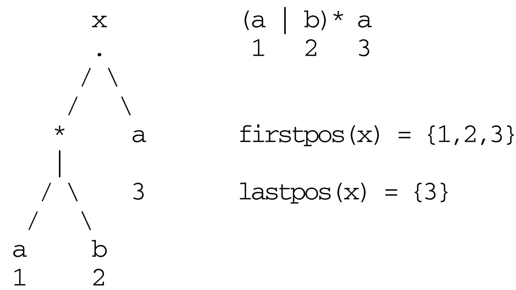
Funcția *nullabl***e** este o funcție logică care are valoarea adevărat dacă şi numai dacă subarborele corespunzător nodului respectiv reprezintă o expresie regulată care poate să genereze şirul vid. Astfel funcția *nullable* aplicată subarborelui:



are valoarea *fals* în schimb aplicată asupra subarborelui:

are valoarea *adevărat.*

Funcția *firstpos* aplicată unui subarbore are ca valoare mulțimea codurilor frunzelor corespunzătoare pozițiilor de început pentru subşirurile care pot să fie generate de către expresia regulată corespunzătoare subarborelui respectiv. De exemplu pentru nodul rădăcină al subarborelui:



Funcţia *lastpos* aplicată unui subarbore are ca valoare setul codurilor frunzelor corespunzătoare poziţiei de sfârşit pentru subşirurile care pot să fie generate de către expresia regulată corespunzătoare subarborelui respectiv.

Funcţia *followpos* este definită asupra mulţimii codurilor frunzelor. Dacă *i* este un cod şi i este asociat simbolului x, ($i = c^{-1}(x)$) atunci *followpos*(i) este mulţimea codurilor *j* ($j = c^{-1}(z)$) care satisfac următoarea condiţie: xy (etichetate cu i şi respectiv j) pot să apară într-un şir generat din rădăcină.

Altfel spus dacă se consideră şirurile obţinute din cuvintele din L(r) (limbajul generat de expresia regulată) prin înlocuirea simbolilor din T cu codurile asociate frunzelor din graf care le generează, *followpos*(i) conţine toate codurile care pot să apară imediat după *i* în aceste şiruri. De exemplu *followpos*(1) = {1,2,3}. Pentru a calcula funcţia *followpos* trebuie să se calculeze funcţiile *firstpos* şi *lastpos* care la rândul lor necesită calculul funcţiei *nullable*. Regulile pentru calculul funcţiilor *nullable*, *firstpos* şi *lastpos* sunt următoarele :

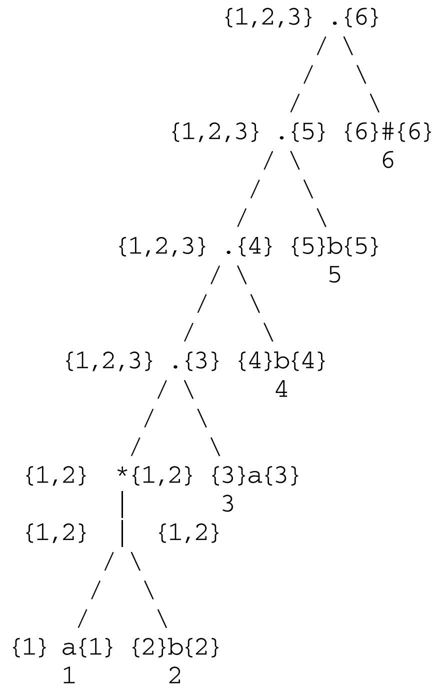
forma nod n	**nullable(n)**	**firstpos(n)**	**lastpos(n)**
n este o frunză cu eticheta λ	adevărat	∅	∅
n este o frunză cu eticheta ≠ λ având codul i	fals	{i}	{i}
n | (c1, c2)	nullable(c1) **sau** nullable(c2)	firstpos(c1) ∪ firstpos(c2)	lastpos(c1) ∪ lastpos(c2)
n . (c1, c2)	nullable(c1) **si** nullable(c2)	**dacă** nullable(c1) firstpos(c1) ∪ firstpos(c2) **altfel** firstpos(c1)	**dacă** nullable(c2) lastpos(c1) ∪ lastpos(c2) **altfel** lastpos(c2)
n * (c)	adevărat	firstpos(c)	lastpos(c)

Pentru a calcula followpos(i) care indică ce coduri pot să urmeze dup[codul **i** conform arborelui se utilizează două reguli :

1. dacă n este un nod corespunzător operatorului de concatenare cu subarborii c1 şi c2 (respectiv stânga dreapta) atunci dacă *i* este un cod din *lastpos*(c1) toate codurile din *firstpos*(c2) sunt în mulţimea ***followpos***(i).

2. dacă n este un nod corespunzător operatorului *, şi *i* este un cod din *lastpos*(n) atunci toate codurile din *firstpos*(n) sunt în mulţimea *followpos*(i).

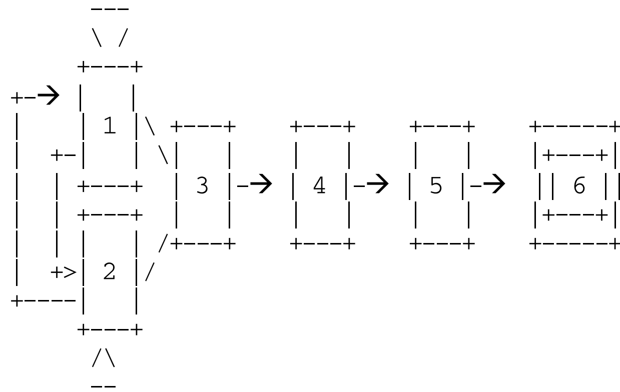
Pentru arborele anterior valorile funcţiilor *firstpos* şi *lastpos* sunt reprezentate în stânga respectiv dreapta fiecărui nod.



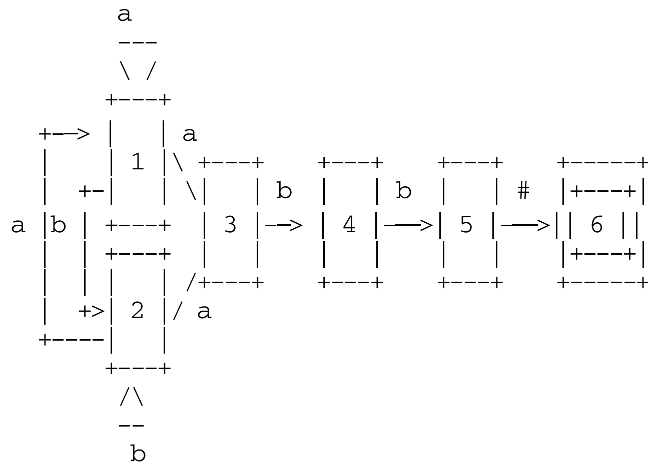
Singurul nod pentru care funcţia *nullable* are valoarea adevărat este nodul etichetat cu *. Pe baza acestor valori să calculăm *followpos*(1). În *followpos*(1) vor apare codurile din *firstpos* pentru nodul etichetat cu * şi codurile din *firstpos* pentru subarborele dreapta corespunzător operatorului de concatenare (followpos(1) = *followpos*(2) = {1,2,3}. Rezultă următoarele valori :

nod	**followpos**
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	-

Pornind de la aceste valori se poate construi un graf în care fiecare nod reprezintă un cod iar între două noduri corespunzătoare codurilor i şi j există un arc dacă j ∈ followpos(i).



Pe baza unui astfel de graf se poate obţine un automat finit nedeterminist fără λ-tranziţii, etichetând arcele în modul următor. Dacă între nodul i şi nodul j există un arc acesta va fi etichetat cu simbolul care are codul j. Fiecare stare a automatului corespunde unui nod din graf. Automatul are ca stări iniţiale stările din *firstpos* pentru nodul rădăcină. Stările finale sunt stările asociate cu simbolul #. Automatul care a rezultat, având mai multe stări iniţiale nu se încadrează de fapt în definiţia pe care am dat-o pentru un automat finit. Dacă însă mai adăugăm o stare suplimentară care să reprezinte unica stare iniţială din care pe baza unor λ-tranziţii ajungem într-una dintre stările automatului obţinut, se vede că ne încadrăm în definiţia considerată



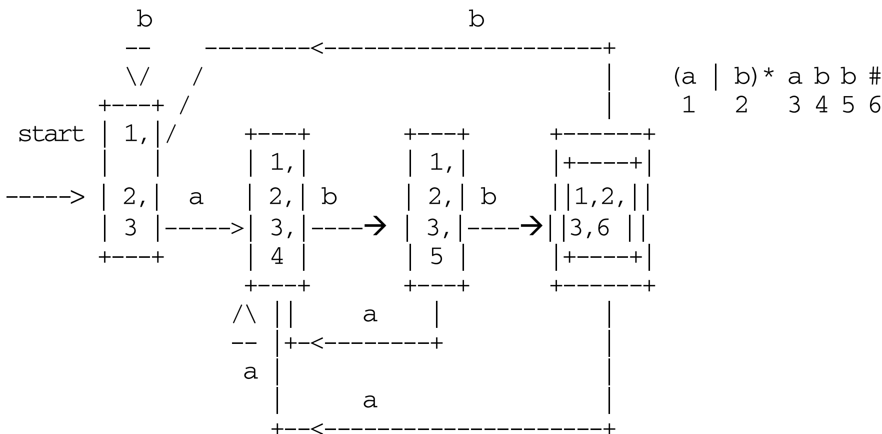
Mai interesant însă este faptul că funcţia ***followpos*** poate să fie utilizată pentru construirea unui automat determinist utilizând un algoritm similar celui pentru construirea submulţimilor.

```
stari_AFD = { first_pos(radacina) } // multime de multimi de coduri
A = ∅
cât timp stari_AFD \ A ≠ ∅ execută
    fie t ∈ stari_AFD \ A              // mulţime de coduri
    A = A ∪ {t}
    pentru fiecare a ∈ T execută
        X = ∪ { followpos(p) | c^-1(p) = a}
              p ∈ t
        daca X ≠ ∅
            stari_AFD = stari_AFD ∪ {X}
            tranz_AFD(t,a) = X
        □
    □
□
```

În algoritm *firstpos*(*rădăcină*) este valoarea funcţiei *firstpos* aplicată nodului rădăcină.

Se observă că *firstpos(rădăcina)* corespunde cu λ-inchidere(s0) iar ***followpos****(p)* reprezintă *λ-închidere(mutare(...))*.

Aplicând acest algoritm şi rezultatele anterioare se obţine automatul finit determinist reprezentat de următorul graf :



Fie q1 = *firstpos*(rădăcina) = {1,2,3}. Se observă că r(1) = r(3) = a, r(2) = r(4) = r(5) = b. Pentru q1 şi a se obţine q2 = *followpos*(1) ∪ *followpos*(3) = {1,2,3,4}, deci *tranz_AFD*(q1,a) = q2. Pentru q2 şi b, *followpos*(2) = {1,2,3} = q1 şi *tranz_AFD*(q1,b) = q1, etc.

Nici aceasta construcţie nu garantează faptul că automatul obţinut este minim.

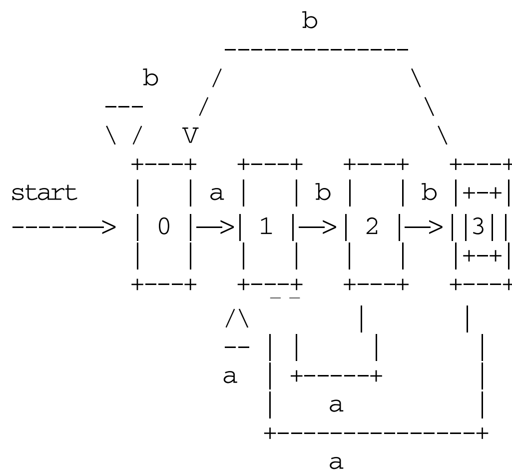
2.3.1.4 Simularea unui automat finit determinist

Construim un algoritm care urmează să răspundă la întrebarea - dându-se un automat finit determinist D şi un şir, este şirul acceptat de către automat ?

Algoritmul utilizează funcţiile *mutare* (care determină starea următoare pentru o stare şi un simbol de intrare dat) şi *caracterul_următor* (care obţine caracterul următor din şirul de intrare).

```
s = s0
c = caracterul_următor
cat timp c ≠ eof executa
     s = mutare(s,c)
     c = caracterul_următor
□
daca s ∈ F
  atunci rezultat "DA"
  altfel rezultat "NU"
□
```

Dacă aplicăm acest algoritm pentru automatul :



care accepta limbajul (a | b)* abb, pentru şirul ababb, se observă că va parcurge secvenţa de stări 012123.

2.3.1.5 Simularea unui automat finit nedeterminist

Prezentăm în continuare un algoritm pentru simularea unui automat finit nedeterminist. Algoritmul citeşte un şir de intrare şi verifică dacă automatul îl accepta sau nu. Se consideră că automatul a fost construit direct pornind de la expresia regulată. În consecință automatul va satisface următoarele proprietăți:

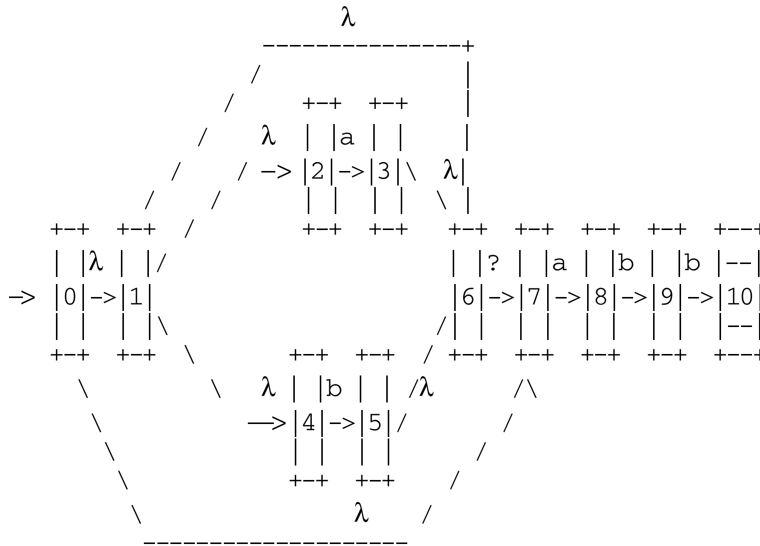
1. are cel mult de două ori mai multe stări față de numărul de simboli şi operatori care apar în expresia regulată;
2. are o singură stare de start şi o singură stare de acceptare. Dintr-o stare de acceptare nu pleacă nici o tranziție;
3. din fiecare stare pleacă fie o tranziție pentru un simbol din alfabetul de intrare T, fie o λ-tranziție sau doua λ-tranziții.

Se consideră că există un mecanism prin care se poate stabilii faptul că s-a ajuns la sfârşitul şirului de intrare (test de tip *eof*). Algoritmul de simulare se aseamănă cu cel utilizat pentru construcția unui automat finit determinist echivalent cu automatul N, diferențele constau în faptul că pentru fiecare mulțime de stări Q în care poate să ajungă automatul pentru un prefix al şirului x se consideră pentru construirea mulțimii Q' de stări următoare numai simbolul curent din şirul x. În momentul în care s-a ajuns la sfârşitul şirului (s-a ajuns la *eof*) dacă în setul stărilor curente este inclusă şi o stare de acceptare (finală), atunci răspunsul este "DA" altfel răspunsul este "NU".

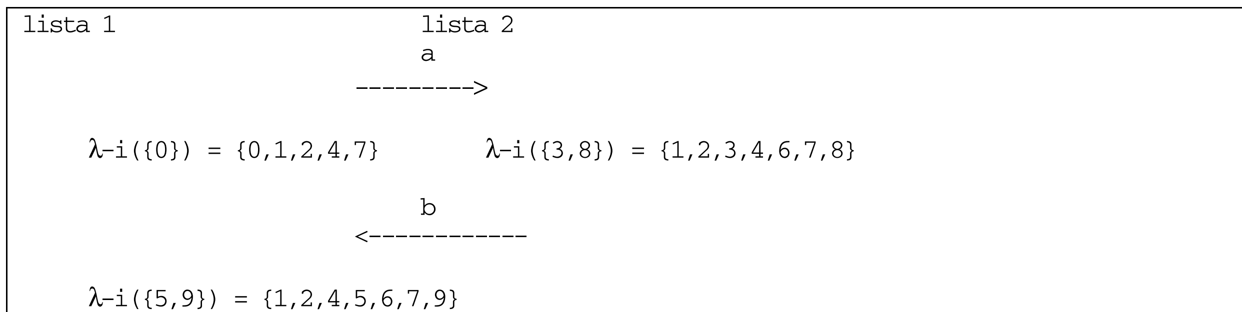
```
S = λ-inchidere({s0})  # λ-inchidere(S) = {s' ∈ Q |∃ s ∈ S, s' ∈ m(s,λ)} ∪ S
a = caracterul_următor
cât timp a ≠ eof execută
    S = λ-inchidere(mutare(S,a)) # mutare(S,a) = ∪ m(s',a)
                                                 s'∈ S
    a = caracterul_următor
□

daca S ∩ F ≠ ∅
    atunci rezultat "DA"
    altfel rezultat "NU"
□
```

Algoritmul poate să fie implementat eficient utilizând două liste şi un vector indexat cu stările automatului finit nedeterminist. Într-o listă se păstrează mulțimea stărilor curente ale automatului, în cealaltă mulțimea stărilor următoare. Utilizând algoritmul de calcul al stărilor următoare pentru λ-tranziții se calculează setul λ-închidere pentru o stare dată. Vectorul este utilizat pentru a asigura funcționarea listei ca o mulțime în sensul că dacă o stare este conținuta în listă atunci ea nu trebuie să mai poată fi adăugată. După ce s-a terminat calculul λ-inchiderii pentru o stare se face schimbarea rolului celor două liste. Deoarece fiecare stare are cel mult două tranziții, fiecare stare poate să producă cel mult două stări noi. Fie |N| numărul de stări pentru AFN. Deoarece în lista pot să fie memorate maximum |N| stări, calculul stării următoare se poate face într-un timp proporțional |N|. Deci timpul necesar pentru simularea AFN pentru şirul de intrare x va fi proporțional cu |N| x |x|. Să reluăm de exemplu automatul :



Acest AFN acceptă expresia regulata *(a|b)*abb*. Să considerăm de exemplu şirul de intrare *x = ab*. Evoluţia celor două liste va fi următoarea :



```
lista 1                    lista 2
                           a
                       --------->

    λ-i({0}) = {0,1,2,4,7}        λ-i({3,8}) = {1,2,3,4,6,7,8}

                           b
                       <-----------

    λ-i({5,9}) = {1,2,4,5,6,7,9}
```

Se observă că s-a ajuns la mulţimea {1,2,4,5,6,7,9} care nu conţine o stare de acceptare deci şirul x nu este acceptat de către automat.

2.3.1.6 Probleme de implementare pentru automatele finite deterministe şi nedeterministe

În compilatoare, automatele finite sunt utilizate pentru implementarea analizoarelor lexicale. Specificarea atomilor lexicali se face sub forma de expresii regulate. Din acest motiv este interesant să discutăm modul în care pornind de la o expresie regulată putem să ajungem la un program care să realizeze "execuţia" sau altfel spus simularea automatului finit corespunzător. Din cele prezentate anterior se observă că dându-se o expresie regulată r şi un şir de intrare x există doua metode pentru a verifica dacă $x \in L(r)$.

Se poate construi automatul finit nedeterminist (AFN) corespunzător expresiei regulate. Dacă expresia regulata r conţine |r| simboli atunci aceasta construcţie se face într-un timp proporţional cu |r|. Numărul maxim de stări pentru AFN este de maximum 2 x |r|, iar numărul de tranziţii pentru fiecare stare este 2 (conform construcţiei), deci memoria maximă necesară pentru tabela de tranziţii este proporţională cu |r|. Algoritmul de simulare pentru AFN are un număr de operaţii proporţional cu |Q| x |x| deci cu |r| x |x| (unde |x| reprezintă lungimea şirului x). Dacă |x| nu este foarte mare soluţia este acceptabilă.

O altă abordare posibilă este construirea automatului finit determinist (AFD) corespunzător automatului finit nedeterminist construit pornind de la expresia regulată. Simularea AFD se face într-un

timp proporţional cu |x| indiferent de numărul de stări ale AFD. Această abordare este avantajoasă dacă este necesară verificarea mai multor şiruri de lungime mare pentru o aceeaşi expresie regulată (vezi de exemplu situaţia în care un editor de texte are o funcţie de căutare pentru un subşir de o formă generală dată). Pe de alta parte există expresii regulate pentru care AFD-ul necesită un spaţiu de memorie exponenţial în numărul de simboli şi operatori din expresia regulata. Să considerăm de exemplu expresia :

(a|b)*a(a|b)(a|b)...(a|b)

care reprezintă un şir de simboli *a* şi *b* şi care se termina cu (n -1) simboli (a|b). Şirul reprezentat de aceasta expresie regulată este format din cel puţin n simboli *a* şi *b*, astfel încât există un simbol a pe poziţia n de la sfârşitul şirului. Se observă că un AFD pentru recunoaşterea acestei expresii trebuie să memoreze ultimele n caractere, deci sunt necesare cel puţin 2^n stări (pentru a codifica toate combinaţiile posibile de caractere a şi b de lungime n).

O alta abordare este de a utiliza un AFD, fără a construi întreaga tabela de tranziţii. Ideea este de a determina o tranziţie numai dacă este necesară. Odată calculată o tranziţie aceasta este memorată într-o memorie asociativă. Ori de câte ori este necesară o tranziţie se face întâi o căutare în memoria asociativă. Dacă nu se găseşte se face calculul.

Concentrând rezultatele se obţine următoarea tabelă :

	AFN	**AFD**
complexitate construcţie	O(|r|)	O(|r| x |T|)
memorie ocupată	O(|r|)	$O(c^{\|r\|})$
complexitate simulare	O(|r| x |x|)	O(|x|)

2.3.1.7 Minimizarea numărului de stări pentru AFD

Se poate demonstra că fiind dată o mulţime regulata există un unic automat finit determinist cu număr minim de stări care acceptă limbajul generat de mulţimea respectivă. În cele ce urmează considerăm că automatul finit are funcţia de tranziţie definită pentru întreg domeniul Q X T. Dacă există combinaţii (s,i) $\in$ Q x T pentru care m(s,i) nu este definită considerăm o extindere a automatului pentru care adăugăm o stare d. Pentru $\forall$ i $\in$ T, m(d, i) = d, pentru $\forall$ s $\in$ Q, $\forall$ i $\in$ T pentru care m(s,i) este nedefinită în automatul iniţial vom considera că m(s, i) = d.

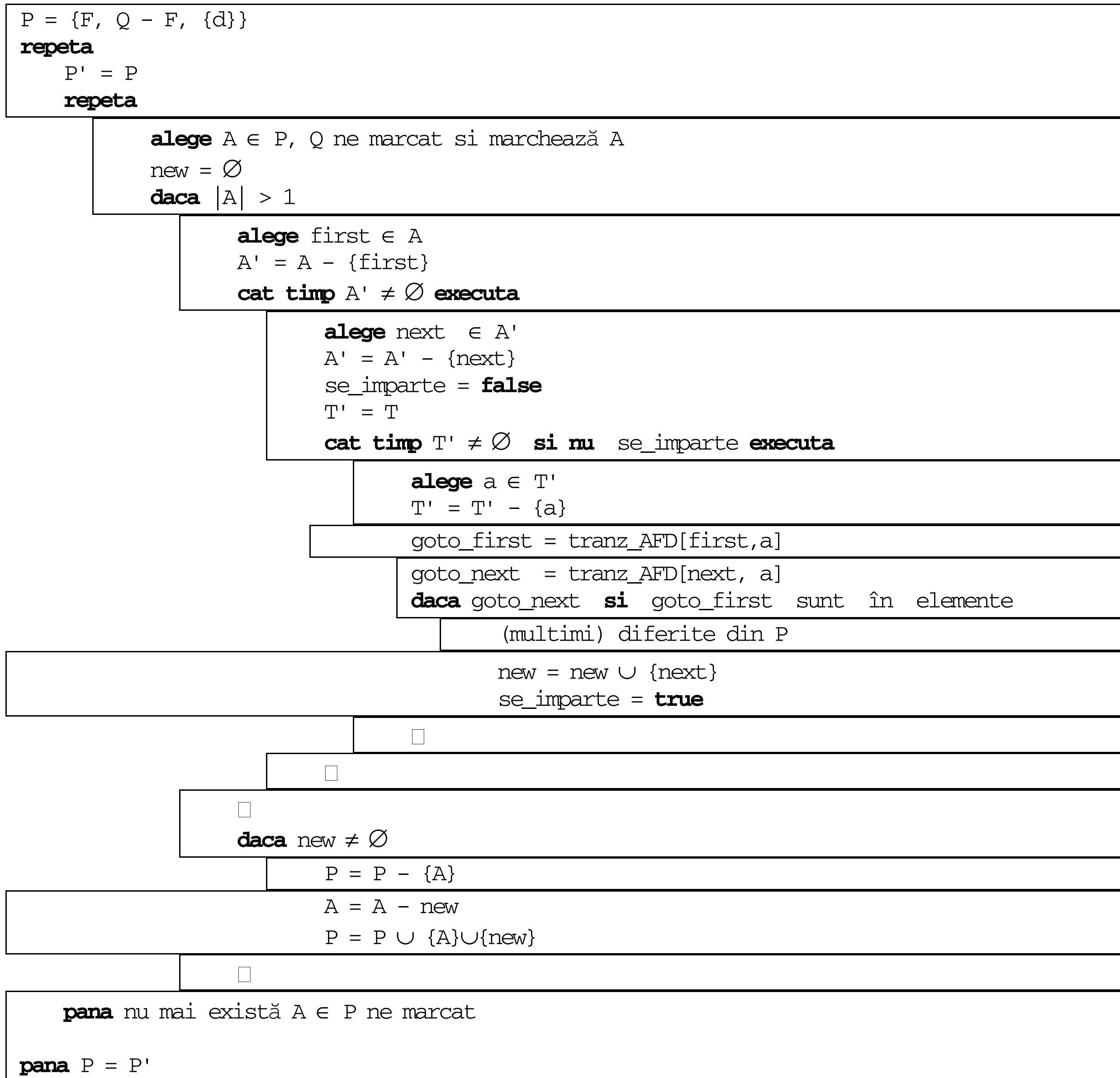
Construim o extensie a funcţiei m, $\overline{m}$: x T* $\rightarrow$ Q în modul următor :

1. $(\forall s \in Q)(\overline{m}(s, \lambda) = s)$
2. $(\forall s \in Q)(\forall w \in T^*)(\forall i \in I)(\overline{m}(s, wi) = m(\overline{m}(s, w), i)$

Fie w $\in$ T* spunem că w *separă* stările s, t $\in$ Q dacă şi numai dacă $\overline{m}(s,w) \in F$ şi $\overline{m}(t,w) \notin F$ sau $\overline{m}(t,w) \in F$ şi $\overline{m}(s,w) \notin F$.

Un automat este redus dacă şi numai dacă pentru oricare două stări q1, q2 $\notin$ F există o secvenţă de intrare care să le separe. Problema construirii automatului redus echivalent unui automat dat este de fapt problema construirii partiţiilor induse de o relaţie de echivalenţă. Iniţial, considerăm ca mulţimea Q este împărţită în F, Q - F şi {d}. Fie P = { Q - F, F, {d}}. În continuare pentru o mulţime de stări A $\in$ P, pentru care |A| > 1, se parcurg simbolii de intrare. Dacă pentru un simbol a $\in$ $\in$ şi stările q1, q2 $\in$ A, m(q1, a) şi m(q2, a) fac parte din elemente diferite din P înseamnă că stările succesoare stărilor q1 şi q2 nu fac parte din aceeaşi mulţime a partiţiei. Corespunzător Q va fi împărţit în 4 mulţimi, etc. Algoritmul corespunzător este:

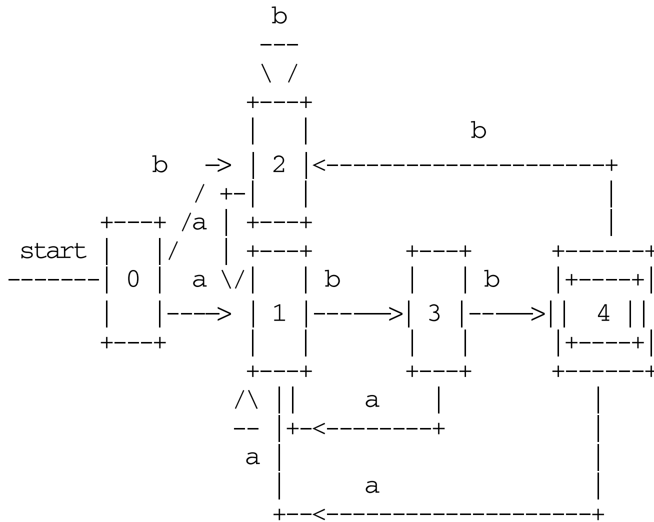
Descrierea algoritmului de minimizare este:



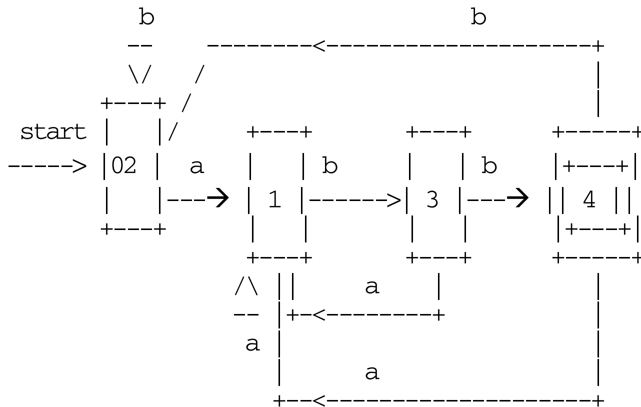
```
P = {F, Q - F, {d}}
repeta
    P' = P
    repeta
        alege A ∈ P, Q ne marcat si marchează A
        new = ∅
        daca |A| > 1
            alege first ∈ A
            A' = A - {first}
            cat timp A' ≠ ∅ executa
                alege next  ∈ A'
                A' = A' - {next}
                se_imparte = false
                T' = T
                cat timp T' ≠ ∅  si nu  se_imparte executa
                    alege a ∈ T'
                    T' = T' - {a}
                    goto_first = tranz_AFD[first,a]
                    goto_next  = tranz_AFD[next, a]
                    daca goto_next  si  goto_first  sunt  în  elemente
                        (multimi) diferite din P
                        new = new ∪ {next}
                        se_imparte = true
                    □
                □
            □
            daca new ≠ ∅
                P = P - {A}
                A = A - new
                P = P ∪ {A}∪{new}
            □
    pana nu mai există A ∈ P ne marcat

pana P = P'
```

Algoritmul utilizat nu este optim din punct de vedere al timpului de execuție. La fiecare pas mulțimea de stări Q se împarte în cel mult două noi mulțimi : new şi Q \ new. Există algoritmi pentru care numărul de operații este proporțional cu n log n. Să considerăm următorul exemplu :

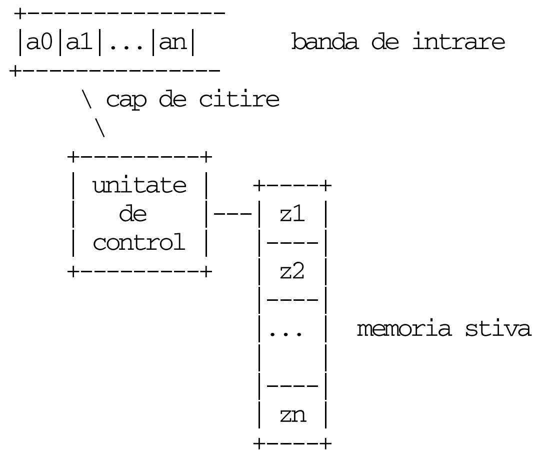


Inițial P = {{0, 1, 2, 3}, {4}}, A = {0, 1, 2, 3}. Fie first = 0, next = 1, m(0, a), m(1, a) ∈ A; m(0, b), m(1, b) ∈ A. Pentru next = 2, m(0, a) ∈ A, m(2, a) ∈ A; m(0, b) ∈ A, m(2,b) ∈ A. Continuând pentru next = 3, m(0, b) ∈ A dar m(3, b) ∉ A, deci new = {3}. S-a obținut P = {{0, 1, 2}, {3}, {4}}. Se observă că P ≠ P'. Urmează A = {0, 1, 2}. Se obține m(0, b) ∈ Q dar de aceasta data m(1, b) ∉ A. Rezultă P = {{0, 2}, {1}, {3}, {4}}. Pentru A = {0, 2} se obține m(0, a), m(2, a) ∈ {1}; m(0, b), m(2, b) ∈ A. Se obține automatul finit determinist cu număr minim de stări:



2.3.2 Automate cu stivă (pushdown)

Un astfel de automat are o memorie organizată ca o stivă :



Un automat cu stivă (PDA) este un obiect matematic P = (Q, T, Z, m, q0, z0, F) unde:

- Q - este o mulțime finită de simboli ce reprezintă stările posibile pentru unitatea de control a automatului;
- T - este mulțimea finită a simbolilor de intrare;
- Z - este mulțimea finită a simbolilor utilizați pentru stivă;
- m - este o funcție, $m : S \times (T \cup \{\lambda\}) \times Z \rightarrow P(S \times Z^*)$ este funcția care descrie modul în care se obține starea următoare şi informația care se introduce în stivă pentru o combinație stare, intrare, conținut stivă dată;
- $q0 \in Q$ este starea inițială a unității de control;
- $z0 \in Z$ este simbolul aflat în vârful stivei în starea inițială;
- $F \subseteq Q$ reprezintă mulțimea finită a stărilor finale.

O configurație de stare a automatului este un triplet $(q, w, \alpha) \in Q \times T^* \times Z^*$ unde :

- q - reprezintă starea curentă a unității de control;
- w - reprezintă partea din şirul de intrare care nu a fost încă citită. Dacă $w = \lambda$ înseamnă că s-a ajuns la sfârşitul şirului de intrare;
- α - reprezintă conținutul stivei.

O tranziție a automatului este reprezentată prin relația |- asupra mulțimii configurațiilor automatului, este definită în modul următor :

$$(q, aw, z\alpha) \vdash (q', w, \beta\alpha)$$

unde $(q', \beta) \in m(q, a, z)$, $q \in Q$, $a \in T \cup \{\lambda\}$, $w \in T^*$, $z \in Z$, $\alpha \in Z^*$.

Dacă $a \neq \lambda$ înseamnă că, dacă unitatea de control este în starea q, capul de citire este pe simbolul a iar simbolul din vârful stivei este z atunci automatul poate să îşi schimbe configurația în modul următor : starea unității de control devine q', capul de citire se deplasează cu o poziție la dreapta iar simbolul din vârful stivei se înlocuieşte cu β.

Dacă a = λ înseamnă că avem o λ-tranziţie pentru care simbolul aflat în dreptul capului de citire pe banda de intrare nu contează (capul de citire nu se va deplasa), însă starea unităţii de control şi conţinutul memoriei se pot modifica. O astfel de tranziţie poate să aibă loc şi după ce s-a parcurs întregul şir de intrare.

Dacă se ajunge într-o configuraţie pentru care stiva este goală nu se mai pot executa tranziţii. Relaţia |- se poate generaliza la $|-^i$, $|-^+$, $|-^*$, într-o manieră similară relaţiei de derivare pentru forme propoziţionale.

O configuraţie iniţială pentru un automat cu stivă este o configuraţie de forma (q0, w, z0) unde w ∈ T*. O configuraţie finală este o configuraţie de forma (q, λ, α) cu q ∈ F, α ∈ Z*.

Aşa cum a fost definit automatul cu stivă este nedeterminist, adică pentru o configuraţie dată în cazul general pot să urmeze mai multe configuraţii următoare.

Un automat cu stivă este determinist dacă pentru orice configuraţie există cel mult o singură tranziţie următoare. Mai precis un automat cu stivă PD = (Q, T, Z, m, q0, z0, F) este determinist dacă pentru orice (q, a, z) ∈ x (T ∪ {λ}) x Z sunt îndeplinite următoarele condiţii:

1. |m(q, a, z)| ≤ 1,
2. dacă m(q, λ, z) ≠ ∅ atunci m(q, a, z) = ∅ pentru ∀a ∈ T

Spunem că un şir w este *acceptat* de un automat cu stivă prin stări finale dacă este posibilă o evoluţie ca (q0, w, z0) |-* (q, λ, α) pentru q ∈ F şi α ∈ Z*. Limbajul acceptat de un automat cu stivă P în acest mod se notează cu L(P). Se observă că pentru un automat cu stivă nedeterminist pentru un şir dat sunt posibile mai mute evoluţii.

Să considerăm de exemplu automatul cu stivă care acceptă limbajul L = $\{0^n1^n \mid n > 0\}$.

P = ({q0, q1 ,q2}, {0,1}, {z, 0}, m, q0, z,{q0}) unde

```
m(q0,0,z) = {(q1,0z)}
m(q1,0,0) = {(q1,00)}
m(q1,1,0) = {(q2,λ)}
m(q2,1,0) = {(q2,λ)}
m(q2,λ,z) = {(q0,z)}
```

Pentru toate celelalte elemente (s, a, z) ∈ Q x (T ∪ {λ}) x Z care nu sunt descrise de regulile de mai sus, m(s, a, z) = ∅. Se observă că automatul copiază toate zerourile de pe banda de intrare în stivă şi apoi descarcă stiva pentru fiecare simbol 1 întâlnit la intrare. De exemplu pentru şirul 0011, automatul va executa următoarea secvenţă de tranziţii :

```
(q0,0011,z)     |- (q1,011,0z)
                |- (q1,11,00z)
                |- (q2,1,0z)
                |- (q2,λ,z)
                |- (q0,λ,z)
```

Pentru şirul de intrare 011 care nu aparţine limbajului evoluţia va fi :

```
(q0,011,z)      |- (q1,11,0z)
                |- (q2,1,z)
                |- (q2,1,z)
                |- (q0,1,z)
```

Se observă că deşi s-a ajuns într-o stare finală nu s-a ajuns la sfârşitul şirului de intrare deci şirul nu a fost acceptat. Pentru şirul de intrare 001, evoluția este :

```
(q0,001,z)        |- (q1,01,0z)
                  |- (q1,1,00z)
                  |- (q2,λ,0z)
```

În acest caz s-a ajuns la sfârşitul şirului dar starea în care se găseşte automatul nu este finală deci nici acest şir nu este acceptat.

Pentru a demonstra că $L(P) = \{0^n1^n\}$ trebuie să arătăm că $\{0^n1^n\} \subseteq L(P)$ şi $L(P) \subseteq \{0^n1^n\}$. Se pot face următoarele observații :

$$(q0,0,z) \vdash (q1,\lambda,0z)$$
$$(q1,0^i,0z) \vdash^i (q1,\lambda,0^{i+1}z)$$
$$(q1,1,0^{i+1}z) \vdash (q2,\lambda,0^iz)$$
$$(q2,1^i,0^iz) \vdash^i (q2,\lambda,z)$$
$$(q2,\lambda,z) \vdash (q0,\lambda,z)$$

Rezultă că pentru şirul 0^n1^n, $n > 1$ se obține :

$$(q0,0^n1^n,z) \vdash^{2n+1} (q0, \lambda, z),\ n > 1$$
$$(q0,\lambda,z) \vdash^0 (q0,\lambda,z)$$

Deci $\{0^n1^n\} \subseteq L(P)$. Trebuie să arătăm că şi $L(P) \subseteq \{0^n1^n\}$. Se observă că pentru a accepta un şir din $\{0^n1^n\}$ pentru care $n > 0$, P trece în mod obligatoriu prin succesiunea de stări q0, q1, q2, q0.

Dacă $(q0, w, z) \vdash^i (q1, \lambda, \alpha)$, $i > 1$ înseamnă ca $w = 0^i$ şi $\alpha = 0^iz$.
Dacă $(q2, w, \alpha) \vdash^i (q2, \lambda, \beta)$ înseamnă că $w = 1^i$ şi $a = 0^i\beta$.
Dacă $(q1, w, \alpha) \vdash (q2, \lambda, \beta)$ înseamnă că $w = 1$ şi $a = 0\beta$
Dacă $(q2, w ,z) \vdash^* (q0, \lambda, z)$ înseamnă că $w = \lambda$.

Deci dacă $(q0, w, z) \vdash^i (q0, \lambda, z)$ atunci fie $w = \lambda$ şi $i = 0$ fie $w = 0^n1^n$, $i = 2 n + 1$. Deci $L(P) \subseteq \{0^n1^n\}$.

Să considerăm şi automatul care acceptă limbajul $L = \{wcw^R| w \in \{a,b\}^+\}$, P = ({q0, q1, q2}, {a, b}, {z, a, b}, m,q0, z, {q2}) unde :

```
m(q0,x,y) = {(q0,xy)}, x ∈ {a, b}, y ∈ {z, a, b}
m(q0,c,x) = {(q1,x)}, x ∈ {z, a, b}
m(q1,x,x) = {(q1,λ)}, x ∈ {a, b}
m(q1,λ,z) = {(q2,λ)}
```

Funcționarea automatului parcurge trei etape. Întâi se copiază în stivă şirul w, apoi se identifică simbolul c care marchează mijlocul şirului, după care se descarcă stiva conform w^R.

Să considerăm şi automatul care acceptă limbajul $L = \{ww^R| w \in \{a,b\}^+\}$, P = ({q0, q1, q2}, {a, b}, {z, a, b}, m,q0, z, {q2}) unde :

```
m(q0,x,z) = {(q0,xz)}, x ∈ {a, b}
m(q0,x,x) = {(q0,xx),(q1,λ)}, x ∈ {a, b}
m(q0,a,b) = {(q0,ab)}
m(q0,b,a) = {(q0,ba)}
m(q1,x,x) = {(q1,λ)}, x ∈ {a, b}
m(q1,b,b) = {(q1,λ)}
```

De data aceasta nu se ştie unde se găseşte mijlocul şirului, astfel încât ori de câte ori simbolul din vârful stivei este acelaşi cu cel de la intrare s-ar putea să se fi găsit mijlocul şirului. Se observă că cel de al doilea automat nu este determinist şi că nici nu se poate construi un automat determinist care să accepte limbajul $L = \{ww^R | w \in \{a,b\}^+\}$.

Să considerăm de exemplu secvenţa de mişcări pentru şirul abba.

```
(q0,abba,z)      |- (q0,bba,az)
                 |- (q0,ba,baz)
                 |- (q0,a,bbaz)
                 |- (q0,λ,abbaz)
```

Se observă că pentru această secvenţă de tranziţii nu se ajunge într-o stare finală. Automatul fiind nedeterminist trebuie să cercetăm însă toate secvenţele de tranziţii posibile. O altă secvenţă este:

```
(q0,abba,z)      |- (q0,bba,az)
                 |- (q0,ba,baz)
                 |- (q1,a,az)
                 |- (q1,λ,z)
                 |- (q2,λ,λ)
```

Rezultă ca automatul ajunge în starea q2 deci acceptă şirul abba.

2.3.2.1 Automate cu stivă cu acceptare prin stivă goală

Fie P = (Q, T, Z, m, q0, z0, F) un automat cu stivă. Spunem că un şir $w \in T^*$ este acceptat de P prin stiva goală dacă există o evoluţie pentru care $(q0, w, z0) |-^+ (q, \lambda, \lambda)$ pentru $q \in Q$.

Fie Le(P) mulţimea şirurilor acceptate de P prin stiva goală. Se poate demonstra următorul rezultat. Dacă L(P) este limbajul acceptat de automatul P prin stări finale atunci se poate construi un automat cu stivă P' astfel încât L(P) = Le(P').

Automatul P' va simula funcţionarea automatului P. Ori de câte ori P intră într-o stare finală, P' poate continua simularea sau intră într-o stare speciala qe în care goleşte stiva ($qe \notin Q$). Se poate însă observa că P poate să execute o secvenţă de mişcări pentru un şir de intrare w care să permită golirea stivei fără ca şirul să fie acceptat de către automatul P (pentru P nu contează decât ajungerea într-o stare finală). Pentru a evita astfel de situaţii se consideră un simbol special pentru iniţializarea stivei, simbol care nu poate să fie scos din stiva decât în starea qe. Fie $P' = (Q \cup \{qe, q'\}, T, Z \cup \{z'\}, m', q', z', \emptyset)$ unde funcţia m' este definită în modul următor :

1. dacă $(r, t) \in m(q, a, z)$ atunci $(r, t) \in m'(q, a, z)$, pentru $r, q \in Q$, $t \in Z^*$, $a \in T \cup \{\lambda\}$, $z \in Z$.
2. $m'(q', \lambda, z') = \{(q0, z0z')\}$. P' îşi iniţializează stiva cu z0z' (z' reprezintă simbolul de iniţializare al stivei pentru P').
3. pentru $\forall q \in F$ şi $z \in Z$, $(qe, \lambda) \in m'(q, \lambda, z)$;
4. pentru $\forall z \in Z \cup \{z'\}$, $m'(qe, \lambda, z) = \{(qe, \lambda)\}$

Se observă că :

```
(q',w,z') |- (q0,w,z0z') |-ⁿ (q, λ, y1 y2 ... yr)
|- (qe, λ, y2 ... yr)
|-^(r-1) (qe, λ, λ)
```

cu y_r = z' dacă şi numai dacă (q0, w, z0) $|-^n$ (q, λ, y_1 y_2 ... y_{r-1}) pentru q ∈ F şi y_1 y_2 ... y_{r-1} ∈ Z*. Deci Le(P') = L(P).

Se poate demonstra şi rezultatul invers, adică dându-se un automat cu stivă şi limbajul acceptat de acesta prin stiva goală se poate construi un automat cu stivă care acceptă acelaşi limbaj prin stări finale.

2.3.2.2 Relația între automate cu stivă şi limbajele independente de context

Limbajele acceptate de automatele cu stivă sunt limbajele care pot să fie descrise de gramatici independente de context. Dându-se o gramatică independentă de context să construim automatul care acceptă limbajul generat de aceasta gramatica. Fie G = (N, T, P, S), se obține automatul cu stivă R = ({q}, T, N ∪ T, m, q, S, ∅) unde m este construită conform următoarelor reguli :

1. dacă A → α ∈ P atunci (q, α) ∈ m(q, λ, A)
2. m(q, a, a) = {(q, λ)} pentru orice a ∈ T.

Se observă că automatul astfel construit va accepta limbajul prin stivă goală. Se poate arata că A $\Rightarrow^k$ w dacă şi numai dacă (q,w,A) $|-^{n}$(q, λ, λ) k, n > 1. Altfel spus există o derivare S $\Rightarrow^{+}$w dacă şi numai dacă (q, w, S) $|-^+$ (q, λ, λ). Rezultă deci că L(G) = L(R). Să construim de exemplu automatul care acceptă limbajul expresiilor aritmetice G = ({E, T, F}, {a, +, (,)}, P, E) unde P = {E → E + T | T, T → T * F | F, F → (E) | a}. Aplicând construcția propusă rezultă R = ({q}, {a, +, *, (,)},{E, T, F, a, +, *, (,)}, m, q, ∈, ∅) unde m este definită în modul următor :

```
m(q, λ, E) = {(q, ∈ + T),(q, T)}
m(q, λ, T) = {(q, T * F),(q, F)}
m(q, λ, F) = {(q, (E)),(q, a)}
m(q, b, b) = {(q, λ)} cu b ∈ {a,+,*,(,)}.
```

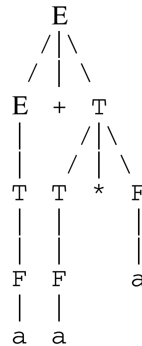
Să considerăm de exemplu o succesiune de configurații pentru a + a * a.

```
(q, a + a * a, E )     |- (q, a + a * a, E + T)
                       |- (q, a + a * a, T + T)
                       |- (q, a + a * a, F + T)
                       |- (q, a + a * a, a + T)
                       |- (q, + a * a, + T)
                       |- (q, a * a, T)
                       |- (q, a * a, T * F)
                       |- (q, a * a, F * F)
                       |- (q, a * a, a * F)
                       |- (q, * a, * F)
                       |- (q, a, F)
                       |- (q, a, a)
                       |- (q, λ, λ).
```

În reprezentarea tranzițiilor vârful stivei a fost reprezentat în stânga. Se observă că secvența de tranziții considerată, simulează următoarea secvență de derivări de forme propoziționale :

$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow a + T \Rightarrow a + T * F \Rightarrow a + F * F \Rightarrow a + a * F \Rightarrow a + a * a$

Se observă ca în aceasta secvenţă s-a înlocuit întotdeauna cel mai din stânga neterminal. Din acest motiv se spune că am utilizat o derivare stânga. Secvenţa de derivări anterioare poate să fie reprezentată de următorul arbore de derivare.



Acest gen de construire de secvenţe de derivare şi deci automatele cu stivă care acceptă limbaje prin stivă goală sunt utilizate în analiza sintactică descendentă.

Un tip special de automat cu stivă îl reprezintă automatul cu stivă extins $\in = (Q, T, Z, m, q0, z0, F)$ pentru care $m : Q \times (T \cup \{\lambda\}) \times Z^* \rightarrow P(Q \times Z^*)$. Se observă că în acest caz din stivă se consideră nu numai simbolul din vârful stivei ci un şir de simboli plasat în vârful stivei. Considerăm pentru acest tip de automat că acceptarea se face prin stări finale (şi în acest caz se poate construi un automat echivalent care să facă acceptarea prin stivă goală). Să considerăm de exemplu automatul care acceptă limbajul $L = \{ww^R \mid w \in \{a,b\}^*\}$. Fie $P = (\{q, p\}, \{a, b\}, \{a,b,S,z\}, m, q, z, \{p\})$ unde funcţia m este definită în modul următor:

```
m(q,a,λ) = {(q,a)}
m(q,b,λ) = {(q,b)}
m(q,λ,λ) = {(q,S)}
m(q,λ,aSa) = {(q,S)}
m(q,λ,bSb) = {(q,S)}
m(q,λ,Sz) = {(p,λ)}
```

De exemplu pentru secvenţa de intrare aabbaa este posibilă următoarea secvenţă de tranziţii :

```
(q, aabbaa, z)  |- (q, abbaa, az)
                |- (q, bbaa, aaz)
                |- (q, baa, baaz)
                |- (q, baa, Sbaaz)
                |- (q, aa, bSbaaz)
                |- (q, aa, Saaz)
                |- (q, a, aSaaz)
                |- (q, a, Saz)
                |- (q, λ, aSaz)
                |- (q, λ, Sz)
                |- (p, λ, λ)
```

Se observă ca datorită tranziţiei $m(q, \lambda, \lambda) = \{(q, S)\}$ automatul este nedeterminist.

Se poate demonstra că pentru orice automat cu stivă extins E, există un automat cu stivă P astfel încât L(E) = L(P).

Să revenim la problema acceptării limbajelor independente de context de către automate cu stivă şi să considerăm din nou exemplul expresiilor aritmetice. Pentru generarea şirului a + a * a putem să considerăm şi următoarea secvenţă de derivări :

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * a \Rightarrow E + F * a \Rightarrow \ \Rightarrow E + a * a \Rightarrow T + a * a \Rightarrow F + a * a \Rightarrow a + a * a$$

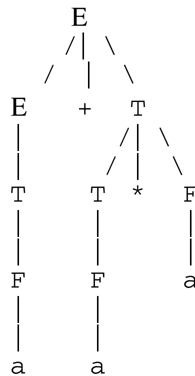
Se observă ca în această secvenţă s-a înlocuit întotdeauna cel mai din dreapta neterminal. Din acest motiv se spune ca s-a construit o derivare dreapta. Considerând acest tip de derivări şi parcurgându-le în ordine inversă (de la sfârşit) se poate introduce noţiunea de reducere stânga. Fie gramatica G = (N,T,P,S) o gramatică independentă de context să presupunem că $S \Rightarrow^* aAw \Rightarrow a\beta w \Rightarrow^* xw$. Dacă în fiecare derivare a fost înlocuit de fiecare dată neterminalul cel mai din dreapta atunci se spune că $a\beta w$ se reduce stânga la aAw dacă $A \rightarrow \beta \in P$. Dacă β este cel mai din stânga astfel de şir atunci spunem ca β este *începutul* şirului $a\beta w$. Un *început* (handle) pentru o derivare dreapta este orice şir care este partea dreapta a unei producţii şi poate să fie înlocuit de neterminalul corespunzător într-o formă propoziţională care a apărut dintr-o derivare dreapta obţinându-se forma propoziţională anterioară din derivarea dreapta.

Să considerăm de exemplu gramatica: G = ({S, A, B},{a, b, c, d}, {S → Ac | Bd, A → aAb | ab, B → aBbb | abb}, S). Această gramatică generează limbajul $\{a^nb^nc \mid n > 1\} \cup \{a^nb^{2n}d \mid n > 1\}$. Să considerăm următoarea secvenţă de derivări $S \Rightarrow Bd \Rightarrow aBbbd \Rightarrow aabbbbd$. Pentru şirul obţinut *ab*b este început deoarece *aBbbd* este forma propoziţională anterioară din derivarea dreapta, în timp ce *ab* nu este deoarece *aAbbbd* nu este o forma propoziţională care poate să apară într-o derivare dreapta.

Reluând din nou gramatica pentru expresii aritmetice şi secvenţa de derivări dreapta :

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * a \Rightarrow E + F * a \Rightarrow \ \Rightarrow E + a * a \Rightarrow T + a * a \Rightarrow F + a * a \Rightarrow a + a * a$$

Rezultă arborele de derivare :



Dacă analizăm acest arbore se vede că *a* (care reprezintă frunza celui mai din stânga subarbore) este un început, dacă se face o reducere se obţine arborele :

```
      E
     /|\
    / | \
   E  +  T
   |    /|\
   |   / | \
   T  T  *  F
   |  |     |
   |  |     |
   F  F     a
      |
      |
      a
```

Continuând se ajunge la :

```
       E
      /|\
     / | \
    E  +  T
         /|\
        / | \
       T  *  F
       |     |
       F     a
       |
       a
```

Următorul început este *a*, etc.

Se observă că pentru a identifica un *început* pentru o formă propoziţională obţinută printr-o derivare dreapta se consideră secvenţa frunzelor celui mai din stânga subarbore de adâncime 1 (toate nodurile acestuia sunt fie frunze fie noduri din care se derivează frunze). De exemplu :

```
|       |       |
a sau F sau T sau            T
                            /|\
                           / | \
                          T  *  F
```

pentru care *a*, *F*, *T* respectiv *T * F* reprezintă părţi dreapta ale unor producţii. Aceasta metoda de reducere a arborilor de derivare se numeşte reducerea capetelor ("handle pruning").

Dându-se o gramatică independentă de context G = (N, T, P, S) se poate construi un automat cu stivă extins care să opereze conform metodei reducerii capetelor în modul următor :

R = ({q, r},T,N $\cup$ T $\cup$ {\$}, m, q, \$, {r}) unde:

1. dacă $A \to \alpha \in P$ atunci $(q, A) \in m(q, \lambda, \alpha)$
2. $m(q, a, \lambda) = \{(q, a)\}$ pentru orice $a \in T$. Se observă că terminalele sunt copiate în stiva;
3. $m(q, \lambda, \$S) = \{(r, \lambda)\}$.

Să construim utilizând această tehnică automatul cu stivă extins care acceptă limbajul expresiilor aritmetice :

R = ({q, r},{a, +, *, (,)},{a, +, *, (,), T, F, $\in$, \$}, m, q, \$,{r}) unde

```
m(q, b, λ) = {(q, b)}, pentru b ∈ {a, +, *, (, )}
m(q, λ, E + T) = {(q, E)}
m(q, λ, T) = {(q, ∈)}
m(q, λ, T * F) = {(q, T)}
m(q, λ, F) = {(q, T)}
m(q, λ, (∈)) = {(q, F)}
m(q, λ, a) = {(q, F)}
m(q, λ, $∈) = {(r, λ)}
```

Fiind vorba de o derivare dreapta, vârful stivei a fost figurat în dreapta. Se observă că această secvenţă de tranziţii este unica secvenţă ce duce la acceptarea şirului de intrare. Pentru şirul a + a * a, automatul R va executa următoarele tranziţii :

```
(q, a + a * a, $) |- (q, + a * a, $a)
                  |- (q, + a * a, $F)
                  |- (q, + a * a, $T)
                  |- (q, + a * a, $E)
                  |- (q, a * a, $E +)
                  |- (q, * a, $E + a)
                  |- (q, * a, $E + F)
                  |- (q, * a, $E + T)
                  |- (q, a, $E + T *)
                  |- (q, λ, $E + T * a)
                  |- (q, λ, $E + T * F)
                  |- (q, λ, $E + T)
                  |- (q, λ, $E)
                  |- (q, λ, ?)
```

Automatele cu stivă construite după modelul prezentat anterior vor fi utilizate în analiza sintactică de tip ascendentă.

Având în vedere echivalenţa dintre un automat cu stivă şi limbajele independente de context, vom spune că un limbaj independent de context este determinist dacă este acceptat de un automat cu stivă determinist. Vom modifica puţin condiţia de acceptare, considerând că un limbaj independent de context este determinist dacă există un automat cu stivă determinist care acceptă limbajul L$. Simbolul $ este un simbol care nu apare în alfabetul peste care este definit limbajul L. Se observă că utilizarea acestui simbol care apare concatenat cu fiecare şir din L indică de fapt posibilitatea automatului cu stivă de a recunoaşte sfârşitul şirului. În acest mod se extinde puţin clasa limbajelor independente de context deterministe.

Se pune întrebarea dacă orice limbaj independent de context este determinist. Să considerăm de exemplu limbajul :

$L = \{ww^R \mid w \in \{a, b, c\}$

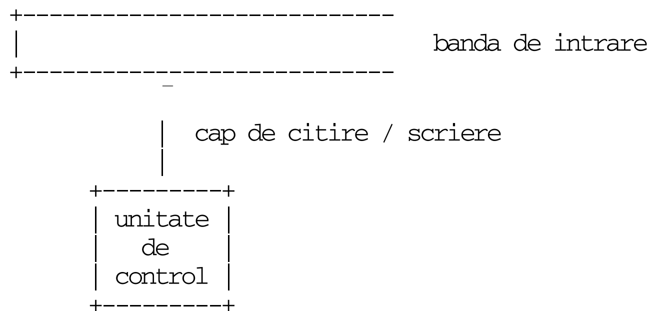
După cum s-a mai discutat un automat cu stivă care acceptă acest limbaj trebuie să ghicească unde este mijlocul şirului pentru ca să înceapă descărcarea stivei. Intuitiv, o astfel de operaţie nu se poate realiza cu un dispozitiv determinist.

Propoziţie. Există limbaje independente de context care sunt acceptate de automate cu stivă nedeterministe şi nu sunt acceptate de automate cu stivă deterministe.

2.3.3 Maşina Turing

O maşină Turing este un acceptor care "ştie" să scrie pe banda de intrare. În acest mod banda de intrare devine memorie auxiliară. Maşina Turing a fost definită de către matematicianul englez Alan Turing (1912 - 1954) în anul 1930. Când a inventat maşina care îi poartă numele, Turing era interesat nu de calculatoare ci de posibilitatea specificări şi rezolvări problemelor matematice utilizând mijloace automate. Adică, poate să fie rezolvată orice problema matematică utilizând nişte reguli elementare de prelucrare a şirurilor ?.

Structura unei maşini Turing este:



Unitatea de control comandă execuţia mişcărilor. O mişcare este formată din :

- stabilirea stării următoare pentru unitatea de control;
- scrierea unui simbol pe banda de intrare sau execuţia unei deplasări cu o poziţie spre stânga sau spre dreapta.

Banda de intrare este mărginită la stânga şi infinită la dreapta. După poziţia ocupată de şirul de intrare pe banda aceasta conţine blancuri. În cele ce urmează vom indica simbolul blanc cu ajutorul caracterului # (presupunând că acest simbol nu face parte din alfabetul limbajului pentru care se construieşte maşina). De asemenea vom utiliza caracterele L şi respectiv R pentru a indica o deplasare la stânga respectiv la dreapta. Maşina Turing poate să scrie pe banda de intrare, deci poate să scrie atât informaţii intermediare cât şi un răspuns înainte de a se oprii. Maşina Turing are o stare specială numită stare de oprire (de halt) pe care o vom nota în continuare cu h. Să considerăm o definiţie formală a noţiunii de maşină Turing.

Se numeşte maşină Turing obiectul matematic :

$MT = (Q, T, m, q_0)$

unde

- Q este o mulţime finită a stărilor maşinii Turing, $h \notin Q$;
- T este alfabetul finit de intrare care conţine simbolul # dar nu conţine simbolii L şi R;
- m este funcţia de tranziţie

  $m : Q \times T \to (Q \cup \{h\}) \times (T \cup \{L, R\})$.

- $q_0 \in Q$ este starea iniţială a maşinii Turing;

Dacă $q \in Q$, $a \in T$ şi $m(q, a) = (p, b)$ înseamnă că fiind în starea *q*, având simbolul *a* sub capul de citire maşina trece în starea *p*. Dacă $b \in T$ atunci simbolul a va fi înlocuit cu b, altfel capul se va deplasa cu o poziţie la stânga sau la dreapta în funcţie de valoarea simbolului $b \in \{L, R\}$. Se observă că maşina Turing a fost definită ca un acceptor determinist.

Încetarea funcţionării maşinii Turing se face dacă se ajunge în starea h (când maşina se opreşte) sau dacă se încearcă deplasarea la stânga dincolo de capătul din stânga al benzii. În acest ultim caz se spune ca maşina s-a agăţat (a ajuns într-o stare de agăţare - hanging state).

Definiţia anterioară corespunde maşinii Turing standard. Faţă de această definiţie există diferite variante, care permit ca în aceeaşi mişcare să se facă şi o scriere şi o deplasare sau care consideră că banda este infinită la ambele capete. Se va arăta că toate aceste variante conduc la acceptoare echivalente ca putere.

Să considerăm un exemplu de maşină Turing MT = ({q0, q1}, {a, #}, m, q0), unde:

```
m(q0, a) = (q1, #)
m(q0, #) = (h, #)
m(q1, a) = (q0, a)
m(q1, #) = (q0, R)
```

Să considerăm că pe banda de intrare se găseşte şirul *aaa* şi că poziţia capului este pe primul caracter din şir. Se observă că pentru această stare iniţială, primul simbol a este înlocuit cu #, noua stare fiind *q1*. Pentru *q1*, dacă sub capul de citire se găseşte un caracter # atunci se va face o deplasare la dreapta starea devenind *q0*. Se continuă evoluţia într-o manieră similară până când în starea *q0* capul de citire se găseşte pe un caracter #. În acest moment se trece în starea *h* şi maşina se opreşte. Rezultă ca evoluţia maşinii duce la ştergerea unui şir de a-uri înscris pe banda de intrare.

Să considerăm şi următorul exemplu MT = ({q0}, {a, #}, m, q0), unde

```
m(q0, a) = (q0, L)
m(q0, #) = (h, #)
```

În acest caz maşina caută primul simbol # la stânga poziţiei din care pleacă capul de citire / scriere. La găsirea acestui simbol se trece în starea *h*. Dacă nu există un astfel de simbol maşina încetează să funcţioneze când ajunge la capătul din stânga al benzii, dar nu se opreşte (este în stare de agăţare).

O configuraţie pentru o maşină Turing este un element din mulţimea :

$$(Q \cup \{h\}) \times T^* \times T \times (T^* (T - \{\#\}) \cup \{\lambda\})$$

Elementele unei configuraţii sunt:

- starea curentă $q \in (Q \cup \{h\})$
- şirul aflat pe banda de intrare la stânga capului de citire / scriere ($a \in T^*$)
- simbolul aflat sub capul de citire / scriere ($i \in T$)
- şirul aflat la dreapta capului de citire / scriere ($\beta \in (T^* (T - \{\#\}) \cup \{\lambda\})$).

Se observă că, deoarece şirul de intrare este finit, în timp ce banda de intrare este infinită la dreapta putem să considerăm întotdeauna că şirul se termină cu un caracter care nu este # (oricum în continuare pe bandă apar numai caractere #). Deci o configuraţie este de forma (q, a, i, β). Pentru simplificare, o configuraţie se poate reprezenta şi sub forma (q, a$\underline{i}$β), subliniind simbolul pe care se găseşte capul de citire / scriere.

Asupra configuraţiilor se poate defini o relaţie de tranziţie |- în modul următor. Fie (q1, w1, a1, u1) şi (q2, w2, a2, u2) două configuraţii. Atunci:

(q1,w1,a1,u1) |- (q2,w2,a2,u2)

# w1 $\underline{a1}$ u1#	# w1$\underline{a1}$ u1 #	# w1 $\underline{a1}$ u1 #
# w2 $\underline{a2}$ u2#	# w2 $\underline{a2}$ u2 #	# w2 $\underline{a2}$ u2 #

dacă şi numai dacă există $b \in T \cup \{L, R\}$ astfel încât m(q1, a1) = (q2, b) şi este îndeplinită una dintre următoarele condiţii:

1. $b \in T$, w1 = w2, u1 = u2, a2 = b;

2. b = L, w1 = w2a2
   dacă a1 ≠ # sau u1 ≠ λ
   u2 = a1 u1
   altfel u2 = λ // capul de citire / scriere este poziţionat după şirul de intrare
3. b = R, w2 = w1a1
   dacă u1 = λ
   u2 = λ
   a2 = #
   altfel u1 = a2 u2

În cazul 1 se face înlocuirea simbolului curent de pe banda de intrare (a) cu b. În al doilea caz este descrisă o mişcare la stânga, respectiv în cazul 3 se descrie o mişcare la dreapta.

Dacă b = L şi w1 = λ atunci (q1, w1, a1, u1) nu are o configuraţie următoare, în acest caz se spune că (q1, w1, a1, u1) este o configuraţie de agăţare, maşina fiind într-o stare de agăţare (hanging state).

Pentru relaţia |- se poate considera închiderea reflexivă şi tranzitivă notată cu |-*.

2.3.3.1 Calcule realizate de Maşina Turing

Un calcul efectuat de o maşina Turing este o secvenţă de configuraţii c0, c1, ..., cn astfel încât $n \geq 0$ şi c0 |- c1 |- ... |- cn. Se spune despre calcul că are loc în n paşi.

Termenul de calcul nu a fost utilizat întâmplător. Se poate considera că o maşină Turing ştie să evalueze funcţii definite pe şiruri de simboli cu valori în şiruri de simboli. Dacă T1 şi T2 sunt doua mulţimi alfabet care nu conţin simbolul #, să considerăm funcţia $f : T1^* \rightarrow T2^*$. Spunem ca MT = (Q, T, m, s) calculează f dacă $T1, T2 \subseteq T$ şi pentru orice şir $w \in T1^*$ dacă u = f(w), $u \in T2^*$, atunci (s, #w<u>#</u>) |-* (h, #u<u>#</u>). Se observă că s-a considerat că capul este poziţionat după sfârşitul şirului de intrare atât la începutul execuţiei cât şi la sfârşitul acesteia.

Dacă pentru o funcţie f dată există o maşina Turing care o calculează spunem că funcţia f este Turing calculabilă.

Să considerăm de exemplu maşina Turing MT = ({q0, q1, q2}, {a, b, #}, m ,q0) unde funcţia m este definită în modul următor:

```
m(q0, a) = (q1, L)
m(q0, b) = (q1, L)
m(q0, #) = (q1, L)
m(q1, a) = (q0, b)
m(q1, b) = (q0, a)
m(q1, #) = (q2, R)
m(q2, a) = (q2, R)
m(q2, b) = (q2, R)
m(q2, #) = (h, #)
```

Să considerăm evoluţia pentru şirul de intrare aab.

```
(q0, #aab#)       |- (q1, #aab#)
                  |- (q0, #aaa#)
                  |- (q1, #aaa#)
                  |- (q0, #aba#)
                  |- (q1, #aba#)
                  |- (q0, #bba#)
                  |- (q1, #bba#)
                  |- (q2, #bba#)
                  |- (q2, #bba#)
                  |- (q2, #bba#)
                  |- (q2, #bba#)
                  |- (h, #bba#)
```

Pentru şirul vid evoluţia este :

```
(q0, ##) |- (q1, ##) |- (q2, ##) |- (h, ##)
```

Funcţia calculată de această maşină Turing este funcţia de interschimbare a caracterelor a şi b.

Din domeniul şirurilor putem să trecem în domeniul numerelor naturale considerând reprezentarea unară a numerelor naturale. Adică utilizând un alfabet I care conţine un singur simbol, I diferit de #, reprezentarea unui număr natural n este data de şirul $\alpha = I^n$. Se observă că numărul 0 este reprezentat de şirul vid. În acest caz o funcţie $f : N \rightarrow N$ este calculată de o maşină Turing, dacă aceasta calculează $f' : I^* \rightarrow I^*$ unde $f'(\alpha) = \beta$ cu $\alpha \in I^n$ şi $\beta \in I^{f(n)}$ pentru orice număr natural.

De la un singur argument funcţia f se poate extinde la mai multe argumente - $f : N^k \rightarrow N$ pentru care corespunde $f' : (I \cup \{,\})^* \rightarrow I^*$ cu $f'(\alpha) = \beta$ pentru α un şir de forma $i_1, i_2, ..., i_k$,... $i_j \in I^{nj}$, $1 \leq j \leq k$, şi $\beta \in I^{f(n1,...,nk)}$.

Să considerăm de exemplu funcţia succesor f(n) = n + 1, n număr natural. MT = ({q0}, {i, #}, m, q0) cu

```
m(q0, I) = (h, R)
m(q0, #) = (q0, I)
```

De exemplu (q0, #II#) |- (q0, #III#) |- (h, #III#). În general :

```
(q0, #I^n#) |- (q0, #I^nI#) |- (h, #I^(n+1)#)
```

O maşină Turing poate să fie utilizată şi ca acceptor de limbaj. Şi anume să considerăm un limbaj L definit asupra alfabetului T care nu conţine simbolii #, D şi N. Fie dL : T* → {D, N} o funcţie definită în modul următor - pentru ∀ w ∈ T*

```
               D dacă w ∈L
          /
dL(w) =
          \
               N dacă w ∉ L
```

Se spune ca limbajul L este decidabil în sens Turing (Turing decidable) dacă şi numai dacă funcţia dL este calculabilă Turing. Dacă dL este calculată de o maşina Turing MT spunem ca MT decide L.

Să considerăm de exemplu limbajul L = {w ∈ T* | |w| mod 2 = 0, T = {a}} (limbajul şirurilor peste alfabetul T = {a} care au lungimea un număr par). Funcţia dL corespunzătoare poate să fie calculată de următoarea maşină Turing :

MT = ({q0, q1, q2, q3, q4, q5, q6}, {a, D, N, #}, m , q0)

unde m este o funcție parțială definită în modul următor :

```
m(q0, #) = (q1, L) pornim spre stânga
m(q1, a) = (q2, #) se şterge un a
m(q1, #) = (q4, R) s-au şters un număr par de a
m(q2, #) = (q3, L) s-a şters un a se continuă spre stânga
m(q3, a) = (q0, #) s-au şters doi de a, suntem ca la început
m(q3, #) = (q6, R) s-a ajuns la stânga cu un număr imapar de a
m(q4, #) = (q5, D) s-au şters zero sau un număr par de a
m(q5, D) = (h, R) răspuns D şi oprire
m(q5, N) = (h, R) răspuns N şi oprire
m(q6, #) = (q5, N) au fost un număr impar de a
```

Să considerăm evoluția maşinii pentru un şir corect :

```
(q0, #aa#) |- (q1, #aa#) |- (q2, #a#) |- (q3, #a#)
           |- (q0, ##) |- (q1, ##) |- (q4, ##)
           |- (q5, #D#) |- (h, #D#)
```

Pentru un şir incorect se obține :

```
(q0, #aaa#)|- (q1, #aaa#) |- (q2, #aa#) |- (q3, #aa#)
               |- (q0, #a#) |- (q1, #a#) |- (q2, ##)
               |- (q3, ##) |- (q6, ##) |- (q5, #N#)
               |- (h, #N#)
```

Noțiunea de acceptare este mai largă decât noțiunea de decidabilitate în legatura cu limbajele. Şi anume dacă L este un limbaj asupra alfabetului T, spunem ca limbajul L este Turing acceptabil dacă există o maşina Turing care se opreşte dacă şi numai dacă $w \in L$.

Limbajele Turing decidabile sunt şi Turing acceptabile. Reciproca nu este însă adevarată.

Fie limbajul :

L = {$w \in$ {a, b}* | w conține cel puțin un simbol a}

considerând maşina Turing MT = ({q0}, {a, b, #}, m, q0) cu

m(q0, a) = (h, a)
m(q0, b) = (q0, L)
m(q0, #) = (q0, L)

Se poate constata ca dacă pe banda de intrare se găseşte un şir din limbaj atunci pornind din configurația inițială se va face căutarea unui simbol a prin deplasare spre stânga. Dacă un astfel de caracter este găsit maşina se opreşte (deci acceptă şirul). Dacă însă pe banda de intrare se găseşte un şir care nu aparține limbajului atunci maşina va intra în starea de agățare după ce a ajuns la capătul din stânga al benzii.

2.3.3.2 Compunerea maşinilor Turing

Pentru a evidenţia "puterea" de calcul a maşinilor Turing vom prezenta mecanismul prin care o maşină Turing se poate construi pe baza unor maşini Turing mai simple.

Propoziţie Fie MT o maşina Turing şi fie (q_i, $w_i\underline{a_i}u_i$), (i = 1, 2, 3) configuraţii ale acestei maşini. Dacă sunt posibile evoluţiile :

(q_1, $w_1\underline{a_1}u_1$) |-* (q_2, $ww_2\underline{a_2}u_2$) şi
(q_2, $w_2a_2u_2$) |-* (q_3, $w_3\underline{a_3}u_3$)

atunci este posibila şi evoluţia :

(q_1, $w_1\underline{a_1}u_1$) |-* (q_3, $ww_3\underline{a_3}u_3$)

Justificare. În evoluţia din (q_2, $ww_2\underline{a_2}u_2$) în (q_3, $w_3\underline{a_3}u_3$) maşina nu poate să ajungă pe banda de intrare pe un simbol aflat la stânga şirului w2 (într-o astfel de situaţie s-ar ajunge la marginea din stânga a benzii, deci s-ar intra în starea de agăţare şi deci nu s-ar mai ajunge în configuraţia următoare). Deoarece maşina are o funcţionare deterministă va rezultă aceeaşi evoluţie şi dacă şirul $w_2a_2u_2$ nu este la începutul benzii de intrare. Rezultă că este posibilă şi evoluţia (q_2, $ww_2\underline{a_2}u_2$) |-* (q_3, $ww_3\underline{a_3}u_3$).

Pe baza propoziţiei anterioare putem să realizăm compunerea maşinilor Turing într-o manieră similară utilizării subprogramelor. Să considerăm de exemplu o maşina Turing M1 care "ştie" să funcţioneze primind un şir pe banda de intrare pornind cu capul de citire / scriere poziţionat după şir. Să presupunem că M1 nu poate să între într-o stare de agăţare. La oprire, capul va fi plasat de asemenea după şirul care a rezultat pe banda de intrare. Dacă M1 este utilizat în cadrul unei alte maşini Turing care printre alte acţiuni pregăteşte şi un şir de intrare pentru M1, atunci după ce M1 primeşte controlul nu va depăşi limita stânga a şirului său de intrare, deci nu va modifica un şir care eventual a fost memorat înainte pe banda de intrare.

În cele ce urmează considerăm ca maşinile Turing care se combină nu pot să între în starea de agăţare şi că fiecare are stările sale proprii diferite de stările tuturor celorlalte maşini Turing. Ca maşini de baza pornind de la care vom construi noi maşini Turing vom considera câteva cu funcţionare elementară. Aceste maşini au un un alfabet de intrare T comun.

1. există |T| maşini care scriu fiecare câte un simbol din T în poziţia curentă a capului. O astfel de maşină este definită ca fiind Wa = ({q}, T, m, q), $a \in T$, m(q, b) = (h, a), $\forall b \in T$. Pentru a simplifica notaţiile Wa va fi reprezentată de simbolul a.

1. există două maşini care execută deplasări elementare. Şi anume VL = ({q}, T, m, q) cu m(q, a) = (h, L) şi VR = ({q}, T, m, q) cu m(q, a) = (h, R), $\forall a \in T$. Vom nota aceste maşini cu L şi respectiv R.

Reprezentarea compunerii maşinilor Turing se face într-o manieră asemănătoare unei structuri de automat finit pentru care o stare este reprezentată de o maşina Turing. Intrarea într-o stare a automatului corespunde cu iniţierea funcţionării unei maşini Turing. La oprirea unei maşini Turing (intrarea acesteia în starea h) în funcţie de simbolul aflat sub capul de citire se va trece într-o altă stare, adică se va iniţia funcţionarea unei alte maşini Turing.

O schemă de maşină Turing este tripletul Ψ = (M, η, M0) unde

- M este o mulţime finită de maşini Turing care au toate un alfabet comun T şi mulţimi de stări distincte;
- M0 $\in$ M este maşina iniţială;
- η este o funcţie parţială , η : M x T$\rightarrow$ M.

O schemă de maşină Turing reprezintă o maşina Turing Ψ compusă din maşinile care formează mulțimea M. Funcționarea acesteia începe cu funcționarea maşinii M0. Dacă M0 se opreşte atunci Ψ poate continua eventual funcționarea conform altei maşini din M. Dacă M0 se opreşte cu capul de citire / scriere pe caracterul a atunci există următoarele situații :

- η(M0, a) este nedefinită, în acest caz Ψ se opreşte;
- η(M0, a) = M', în acest caz funcționarea continuă cu starea inițială a maşinii M'.

Acelaşi mod de înlănțuire va avea loc şi la oprirea (dacă intervine) maşinii M'. În mod formal acest gen de compunere de maşini Turing poate să fie descris în modul următor. Fie M = {M0, .., Mk}, $k \geq 0$ astfel încât $M_i = (Q_i, T, m_i, s_i)$, $0 \leq i \leq k$. Fie q0, ..., qk stări care nu apar în mulțimile Q_i, $0 \leq i \leq k$. Dacă (M, η, M0) este o schemă de maşină Turing , ea va reprezenta maşina MT = (Q, T, m, s) unde

- $Q = Q_0 \cup ... \cup Q_k \cup \{q_0, ..., q_k\}$
- $s = s_0$
- m este definită în modul următor:
  a. dacă $q \in Q_i$, $0 \leq i \leq k$, $a \in T$ şi $m_i(q,a) = (p, b)$, $p \neq h$ atunci $m(q, a) = m_i(q, a) = (p, b)$;
  b. dacă $q \in Q_i$, $0 \leq i \leq k$, $a \in T$ şi $m_i(q, a) = (h, b)$ atunci m(q, a) = (qi, b);
  c. dacă η(Mi, a) ($0 \leq i \leq k$, $a \in T$) este nedefinită atunci m(qi, a) = (h,a);
  d. dacă η(Mi, a) = Mj ($0 \leq i \leq k$, $a \in T$) şi mj(sj, a) = (p, b) atunci

$$m(qi, a) = \begin{cases} (p,b) & p \neq h \\ (qj, b) & p = h \end{cases}$$

Se observă că stările noi q0, ..., qk au fost introduse ca puncte de trecere de la o maşină la alta. Din definiție rezultă că fiecare maşină funcționează "normal" până când se opreşte. Dacă o maşină se opreşte şi este definită maşina următoare se va trece în starea care face legatura cu maşina următoare. Dacă nu este definită maşina următoare, MT se va opri. Dintr-o stare de legatură se intra în funcționarea maşinii următoare pe baza simbolului curent de pe banda de intrare.

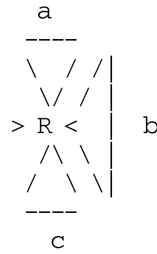
Să considerăm de exemplu mulțimea M = {M0} cu M0 = R = ({q}, T, m, q). Definim funcția η în modul următor :

```
η(M0, a) = M0 dacă a ≠ #
η(M0, #) este nedefinită
```

În acest caz (M, η, M0) reprezintă o maşină notată cu R#, R# = ({q, q0}, T, m, q) unde

```
m(q, a) = (q0, R), a ∈ T
m(q0, a) = (q0, R), a ≠ #
m(q0, #) = (h, a)
```

Se observă că maşina se deplasează la dreapta până la primul simbol #. Dacă T = {a, b. c} putem să reprezentăm maşina R# sub forma:

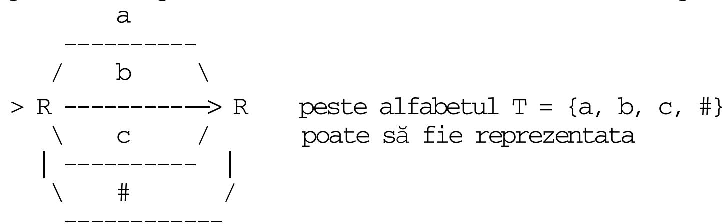


În cele ce urmează vom reprezenta grafic compunerea maşinilor Turing utilizând o serie de reguli :

- fiecare maşină Mi ∈ M apare o singură dată;
- maşina inițială (M0) este indicată prin semnul >;
- dacă a ∈ T şi η(Mi, a) este definită şi de exemplu M' = η(Mi,a) atunci va exista un arc de la Mi la M' etichetat cu a.

Se poate întâmpla ca în M să apară mai multe copii ale aceleiaşi maşini (fiecare cu stările ei proprii diferite de stările celorlalte). În acest caz fiecare dintre exemplare reprezintă o altă maşină Turing deci va fi desenată separat.

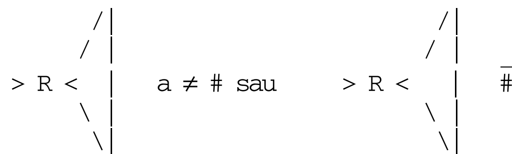
În reprezentările grafice următoare vom utiliza o serie de simplificări. De exemplu schema :



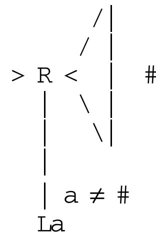
a,b,c,#
> R ---------→ R sau > R --→ R sau RR sau R^2

indicând faptul că se fac două deplasări la dreapta indiferent de conținutul benzii de intrare.

Dacă a ∈ T, $\underline{a}$ reprezintă orice simbol din T diferit de a. Deci maşina care caută simbolul # la dreapta poate să fie reprezentat ca :



Următoarea schemă:



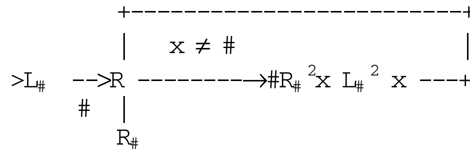
caută mergând la dreapta un simbol diferit de #, când îl găseşte îl copiază la stânga poziţiei pe care l-a găsit. Se observă că din notaţia a ≠ # rezultă că simbolul găsit, şi care este diferit de #, este notat generic cu a şi poate să fie utilizat în continuare.

În cele ce urmează utilizăm următoarele notaţii :

- $R_{\#}$, este maşina care caută primul simbol # la dreapta
- $L_{\#}$, este maşina care caută primul simbol # la stânga
- $R_{\bar{\#}}$, este maşina care caută primul simbol diferit de # la dreapta
- $L_{\bar{\#}}$, este maşina care caută primul simbol diferit de # la stânga

Utilizând aceste maşini "simple" să construim o maşina de copiere C care funcţionează în modul următor. Să presupunem că pe banda de intrare se găseşte un şir w care nu conţine simboli #

(eventual şirul poate să fie şi vid). Presupunem ca la stânga şirului de intrare se găseşte un singur simbol #; pe banda de intrare nu se mai găsesc simboli diferiţi de # care să nu facă parte din w. Capul este poziţionat pe simbolul # care marchează sfârşitul şirului w. Evoluţia maşinii trebuie să fie (s, #w#) |-* (h, #w#w#). Se spune ca maşina transformă şirul #w# în #w#w#. Reprezentarea grafică pentru aceasta maşină este :



Să urmărim evoluţia maşinii pentru conţinutul benzii #abc#

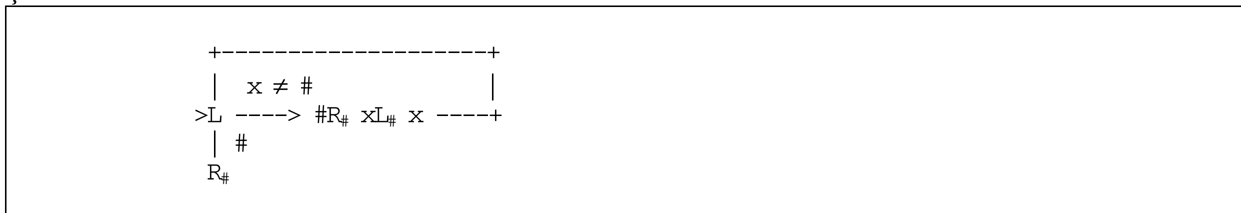
```
(q, #abc#) |-* (q, #abc#)    |- (q, #abc#) |- (q, ##bc#)
            L#                R             #
           |- (q,##bc#)      |- (q,##bc##)
            R#                R#
           |- (q, ##bc#a#)   |- (q, ##bc#a#) |- (q, ##bc#a#)
            x                 L#              L#
           |- (q, #abc#a#)   |- (q, #abc#a#) |- (q, #a#c#a#)
            x                 R               #
           |- (q, #a#c#a#)   |- (q, #a#c#a#)
            R#                R#
           |- (q, #a#c#ab#)  |- (q, #a#c#ab#)
            x                 L#
           |- (q, #a#c#ab#)  |- (q, #abc#ab#)
            L#                x
           |- (q, #abc#ab#)  |- (q, #ab##ab#)
            R                 #
           |- (q, #ab##ab#)  |- (q, #ab##abc#)
            R#R#             x
           |- (q, #ab##abc#)  |- (q,#abc#abc#)
            L#L#             x
           |- (q, #abc#abc#) |- (q,#abc#abc#)
            R                 R#
           |- (h,#abc#abc#)
```

În evoluția prezentată anterior starea nu contează aşa că a fost notată cu un simbol generic q. Se observă că dacă C începe să funcționeze cu un conținut al benzii de forma #u#v#w# la sfârşitul execuției se obține pe banda #u#v#w#w#. C nu funcționează corect dacă la pornire la dreapta capului de citire se găseşte un simbol diferit de #.

Maşina SL :

transformă şirul #w# în w#. Similar se poate construi o maşina SR care transformă şirul #w# în ##ww#.

Fie T0 = T - {#} cu f : T0* → T0* astfel încât f(w) =ww^R, f poate să fie calculată de următoarea maşină:



Pe parcursul funcționării maşina transformă şiruri de forma #x1 ... xn# în #x1 ... xi ... xnxnxn-1 ... xi+1#, i = n-1, ..., 0. Pentru i = 0 se ajunge la #x1 ... xnxn ... x1#.

2.3.3.3 Extensii pentru maşina Turing

Având în vedere simplitatea deosebită a funcționării unei maşini Turing standard, se pune problema dacă aceasta nu poate să fie făcută mai "puternică": prin adăugarea unor extensii care să

o apropie eventual de un calculator real. Se poate demonstra că adăugarea unor facilităţi ca de exemplu :

- banda de intrare infinită la ambele capete;
- mai multe capete de citire / scriere;
- mai multe benzi de intrare;
- o banda de intrare organizată în două sau mai multe dimensiuni (eventual o memorie aşa cum apare la calculatoare);

nu creşte puterea de calcul oferită. Adică, nu se schimbă nici clasa funcţiilor care pot să fie calculate şi nici a limbajelor acceptate sau decise.

Demonstraţiile de echivalenţă sunt constructive realizându-se pentru fiecare extensie construirea unei maşini Turing standard capabilă să simuleze funcţionarea unei maşini Turing având extensia considerată. Să considerăm de exemplu o maşina Turing care are o bandă infinită la ambele capete. Schimbarea definiţiei formale pentru acest model de maşina Turing intervine în modul în care se defineşte noţiunea de configuraţie şi relaţia |- asupra configuraţiilor. În acest caz o configuraţie este de forma (q, w, a, u) şi w nu începe cu un simbol # iar u nu se termina cu un

simbol #. Nu mai există limitări referitoare la deplasarea la stânga şi deci dispar noţiunile de stare şi respectiv de configuraţie de agăţare.

Propoziţie. Fie M1 = (Q1, T1, m1, q1) o maşină Turing cu banda de intrare infinită la ambele capete. Există o maşină Turing standard M2 = (Q2, T2, m2, q2) astfel încât, pentru orice w ∈ (T1 - {#})* sunt îndeplinite următoarele condiţii :

- dacă M1 se opreşte pentru w, adică:

```
(q1, w#) |-* (h, uav), u, v ∈ T1*, a ∈ T1
```

atunci şi M2 se opreşte pentru w, adică:

```
(q2, #w#) |-* (h, #uav), u, v ∈ T1*, a ∈ T1
```

dacă M1 nu se opreşte pentru w atunci nici M2 nu se opreşte pentru w.

Demonstraţie. Banda de intrare pentru M2 se obţine prin îndoirea benzii de intrare a maşinii M1. Adică, dacă considerăm că banda de intrare M1 are un "mijloc" ales arbitrar :

```
 ---------------------------------------
   | -3 | -2 | -1 |  0 |  1 |  2 |  3 |
 ---------------------------------------
```

atunci banda de intrare pentru M2 va avea forma:

```
 +---------------------------
 |    |  0 |  1 |  2 |  3 |
 | $  |----+----+----+----|
 |    | -1 | -2 | -3 | -4 |
 +---------------------------
```

În simulare se va consideră deci că funcţionarea maşini M1 în partea dreapta a benzii de intrare corespunde funcţionării maşini M2 pe pista superioară a benzii sale de intrare, respectiv funcţionarea maşini M1 în partea stângă corespunde funcţionării maşini M2 pe pista inferioară a benzii sale de intrare.

Rezultă deci că pentru M2 se va utiliza un alfabet care conţine perechi de simboli din alfabetul T1. O astfel de pereche (a, b) va indica faptul că a este conţinut de pista superioară iar b este conţinut de pista inferioară. Vom nota cu $\overline{T}$ o mulţime definită în modul următor :

$$\overline{T} = \{ \overline{a} \mid a \in T1\}$$

În acest caz $T2 = \{\$\} \cup T1 \cup (T1 \times T1) \cup (T1 \times \overline{T}) \cup (\overline{T} \times T1)$. M2 simuleaza M1 în modul următor :

1. se împarte banda de intrare în două piste şi se copiază şirul de intrare pe pista superioară;
2. se simulează funcţionarea M1 pe banda obţinută;
3. când şi dacă M1 se opreşte, se reface forma iniţială a benzii.

Pentru a realiza prima etapa trebuie să se realizeze o prelucrare de şiruri. Adică, dintr-o banda de intrare de forma :

```
 ---------------------------------------
   | #  | a1 | a2 | ...| an |  # |  # |
 ---------------------------------------
                                   ⇑
```

trebuie să se ajungă la un conţinut al benzii de forma :

```
 +-----------------------------------
 |    | a1 | a2 | ...| an | # |    |
 | $  |----+----+----+----+----| #  |
 |    | #  | #  | ...| #  | # |    |
 +-----------------------------------
                               ⇑
```

unde a1, ..., an ∈ (T1 - {#})*. Se observă că prelucrările necesare nu sunt deosebit de dificile.

Etapa a doua poate să fie realizată de către o maşina M1' care are în mulţimea sa de stări pentru fiecare q ∈ Q1 o pereche de stări notate <q, 1> şi <q, 2>. Dacă M1' este într-o stare <q, 1> înseamnă că M1' acţionează asupra pistei superioare a benzii de intrare. Dacă M1' este în starea <q, 2> atunci înseamnă că M1' acţionează asupra pistei inferioare. De asemenea există stările <h, 1> şi <h, 2> care reprezintă oprirea în situaţia în care M1' lucra pe pista superioară respectiv inferioară.

Funcţia de tranziţie m' pentru M1' este definită în modul următor :

a) dacă q ∈ Q1, (a1, a2) ∈ T1 x T1 şi m1(q, a1) = (p,b)
atunci

```
                      / (<p, 1>, L)       dacă b = L
   m1'(<q,1>,(a1,a2) =  (<p, 1>, R)       dacă b = R
                      \ (<p, 1>, (b, a2)) dacă b ∈ T1
```

b) dacă q ∈ Q1, (a1, a2) ∈ T1 x T1 şi m1(q, a2) = (p, b)
atunci

```
                         / (<p, 2>, R)       dacă b = L
     m1'(<q,2>,(a1,a2) =  (<p, 2>, L)        dacă b = R
                         \ (<p, 2>, (a1, b)) dacă b ∈ T1
```

c) dacă q ∈ Q1 ∪ {h} atunci

```
m1'(<q, 1>, $) = (<q, 2>, R)
m1'(<q, 2>, $) = (<q, 1>, R)
```

d) dacă q ∈ Q1 ∪ {h} atunci

```
m1'(<q, 1>, #) = (<q, 1>, (#, #))
m1'(<q, 2>, #) = (<q, 2>, (#, #))
```

e) dacă a1, a2 ∈ T1 atunci

```
m1'(<h, 1>, (a1, a2)) = (h, (a', a2))
m1'(<h, 2>, (a1, a2)) = (h, (a1, a'))
```

f) pentru situaţiile care nu se încadrează în nici unul dintre cazurile anterioare m1' se defineşte arbitrar.

Se observă că în cazurile a şi b este descrisă funcţionarea maşinii M1 la dreapta şi respectiv la stânga "mijlocului" ales. Cazul c. tratează comutarea între piste care trebuie realizată atunci când se ajunge la limita din stânga a benzii. Cazul d. indică modul în care se face extinderea celor doua piste la dreapta. Cazul e tratează intrarea în starea de oprire (h) a maşini simulate M1. Se observă că la intrarea în aceasta stare se va produce intrarea în starea h şi a maşinii M1' cu înregistrarea pistei pe care s-a făcut oprirea maşinii simulate.

Dacă banda de intrare a maşini M1 conţine şirul w ∈ (T1 - {#})* şi aceasta se opreşte cu un conţinut de forma :

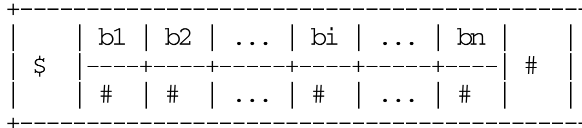
```
---------------------------------------------
 | #  | b1 | b2 | ...| bi | ...| bn | # | # |
---------------------------------------------
                      ⇑
```

atunci M1' se va opri cu un conţinut de forma :

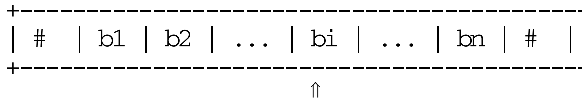
```
+------------------------------------
|    | ck+1 | ck+2 | ... | c2k |    |
| $  |------+------+-----+-----| #  |
|    | ck   | ck-1 | ... | c1  |    |
+------------------------------------
```

unde c1c2 ... c2k = # ... # b1b2 ... bi-1 $\overline{b}$ bi+1 ... bn # ... # cu bi ≠ #, 1 ≤ i ≤ n.

Pentru etapa a treia se face translatarea simbolilor diferiţi de # de pe pista inferioară pe pista superioară :



Din nou aceasta prelucrare este simplă. În final se reface banda sub forma :



Rolul alfabetului $\overline{T}$ utilizat în construirea maşinii M1' este de a permite identificarea poziției capului la oprirea maşinii M1'.

Similar definiției din cazul maşinilor Turing standard putem să considerăm şi în acest caz definiția noțiunii de calcul. şi anume dacă T1 şi T2 sunt doua mulțimi de tip alfabet care nu conțin simbolul # atunci spunem că funcția f : T1* → T2* este calculată de maşina Turing, M care are banda de intrare infinită la ambele capete, dacă şi numai dacă pentru orice w ∈ T1*, dacă f(w) = u atunci (q, w#) |-* (h,u#). De asemenea noțiunile de acceptare respectiv de decidabilitate referitoare la limbaje pot să fie definite pentru maşinile Turing cu banda infinită la ambele capete într-o maniera similara celei de la maşina Turing standard.

Orice funcție care este calculată şi orice limbaj care este acceptat sau decis de o maşina Turing cu banda infinită la ambele capete este calculată respectiv acceptat sau decis de o maşină Turing standard.

Construcții şi rezultate similare pot fi realizate şi pentru celelalte extensii considerate la începutul acestui subcapitol. Din acest motiv în tratarea pe care o vom realiza în continuare putem să considerăm pentru a simplifica demonstrațiile în loc de maşini Turing standard maşini care au oricare dintre extensiile amintite sau chiar combinații ale acestora.

Extinderea maşini Turing se poate face şi prin renunțarea la caracterul determinist al acestora. Pentru o maşină Turing nedeterministă funcția m este definită în modul următor :

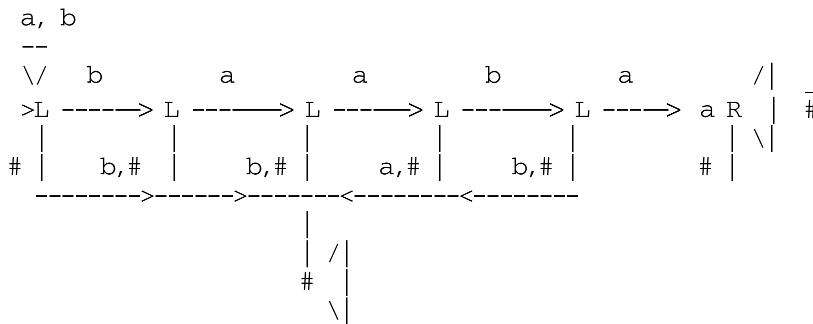
$$m : Q \times T \rightarrow P((Q \cup \{h\}) \times (T \cup \{L, R\}))$$

După cum s-a arătat pentru orice automat finit nedeterminist se poate construi un automat finit determinist echivalent. În cazul automatelor cu stivă aceasta echivalență nu există. Cu alte cuvinte automatele cu stivă nedeterministe sunt "mai puternice" decât cele deterministe deoarece automatele cu stivă nedeterministe accepta o clasa mai larga de limbaje.

În cazul maşinilor Turing nedeterministe deoarece pentru acelaşi şir de intrare se pot obține rezultate diferite, se pune problema cum se alege "calculul util" efectuat de maşina Turing. Vom restringe puțin problema considerând numai noțiunea de acceptare. În acest caz ceea ce contează este că maşina Turing se opreşte în una din evoluțiile sale posibile. Rezultatul depus pe bandă nu este în acest caz neapărat semnificativ. Să considerăm mai întâi un exemplu de maşină Turing nedeterministă. Fie limbajul

```
L = { w ∈ { a, b}*  | w contine sirul abaab}
```

Evident, limbajul este de tip regulat şi poate să fie acceptat de un automat finit determinist. O soluție de maşina Turing nedeterminista care accepta acest limbaj este :



Configuraţia din care se pleacă este (q, #w#). Maşina Turing porneşte spre stânga. La un moment dat, la întâlnirea unui simbol b, maşina Turing "ghiceşte" ca a găsit sfârşitul sub şirului căutat. Dacă ghicirea este corectă atunci maşina se va opri, altfel maşina nu se opreşte. Fiind vorba de o funcţionare nedeterministă tot ce contează este că dintre toate soluţiile posibile există una pentru care maşina se opreşte. Se poate demonstra următoarea propoziţie.

Propoziţie Pentru orice maşina Turing, M1 nedeterministă există o maşina Turing M2 deterministă (standard) astfel încât pentru orice şir w care nu conţine simbolul # :

i) dacă M1 se opreşte pentru w atunci şi M2 se va opri pentru w;
îi) dacă M1 nu se opreşte pentru w atunci nici M2 nu se opreşte pentru w.

Să considerăm o schiţă de demonstraţie pentru propoziţia anterioară. Fie M1 = (S, T1, m, s). M2 trebuie să încerce toate evoluţiile posibile pentru M1 în căutarea unei evoluţii pentru care M1 se opreşte. M1 poate avea însă un număr infinit de evoluţii pornind din s şi având un şir w pe banda de intrare (să nu uităm că dacă şirul nu este acceptat de M1 aceasta nu se va opri). Se pune problema cum trebuie să facă M2 căutarea între aceste evoluţii (în ce ordine ?).

Ideea este următoarea, dacă M1 este în configuraţia C atunci există mai multe configuraţii C' astfel încât C |- C'. Numărul configuraţiilor C' este însă finit şi depinde numai de definiţia funcţiei m (nu de configuraţia C). Dacă starea curenta este q şi simbolul curent la intrare este a, atunci $m(q, a) \in P((Q \cup \{h\}) \times (T \cup \{L, R\}))$ este o mulţime finită, deoarece $|m(q,a)| \leq (|Q| + 1) \times (|T| + 2)$. Fie

$$r = \max_{\substack{q \in Q \\ a \in T}} (|m(q,a)|)$$

Valoarea r rezultă din definiţia funcţiei m şi reprezintă numărul maxim de configuraţii următoare posibile pentru o configuraţie dată. Deci dacă pornim din configuraţia iniţială C = (q,#w#), M1 poate să treacă în maximum r configuraţii următoare diferite. Fiecare dintre aceste r configuraţii diferite poate să treacă la rândul ei în alte r configuraţii diferite. Rezultă deci că pentru o evoluţie în k paşi se poate ajunge la un număr de maximum r^k configuraţii diferite pornind din configuraţia iniţială. Pentru fiecare mulţime m(q, a), $q \in Q$, $a \in T$, considerăm câte o ordonare arbitrară a elementelor care o compun.

M2 va analiza în mod sistematic întâi toate configuraţiile în care se poate ajunge într-un pas, apoi în doi paşi, etc. Dacă în aceasta căutare se trece printr-o configuraţie pentru care M1 se opreşte se va opri şi M2. Dacă M1 se opreşte vreodată atunci ea se opreşte într-un număr finit de paşi. Corespunzător M2 va descoperi aceasta oprire într-un număr finit de paşi.

Maşina M2 poate să fie construită având trei piste. O pistă este utilizată ca martor, pe ea se păstrează şirul iniţial. A doua pistă este utilizată pentru a simula funcţionarea maşini M1. Fiecare simulare începe cu o copiere a conţinutului primei benzi (partea utilă) pe a doua bandă. A treia bandă păstrează evidenţa încercărilor făcute. Şi anume pe această bandă se găseşte un şir dintr-un alfabet D = {d1, ..., dr}, unde r este numărul maxim de configuraţii următoare posibile pentru orice configuraţie a maşini M1. Pe aceasta

bandă se vor genera şiruri care codifică modul în care se aleg alternativele posibile în fiecare pas al simulării. Astfel, un şir de forma d3 d5 d1 indică faptul că se va încerca un calcul în trei paşi. În primul pas se alege dacă există a treia alternativă din cele maximum r posibile, în al doilea pas se alege a cincea alternativă iar în al treilea pas se alege prima alternativă posibilă. Dacă în aceasta alegere nu există o alternativă cu numărul celei indicate se abandonează calculul.

Generarea şirurilor pe a treia bandă se face în ordine lexicografică. Dupa fiecare încercare care nu a dus la oprire se va şterge conţinutul celei de a doua benzi şi se generează un nou şir pe a treia bandă.

Dacă încercarea duce la oprirea maşini simulate se opreşte şi M2.

Se observă că toate prelucrările descrise anterior sunt prelucrări simple asupra unor şiruri de simboli ce pot să fie realizate de către maşini Turing standard.

Pe baza propoziţiei enunţate anterior rezultă ca orice limbaj acceptat de o maşină Turing nedeterministă va fi acceptat şi de o maşina Turing deterministă.

2.3.3.4 Automate liniar mărginite

Un automat liniar mărginit este un caz particular de maşină Turing nedeterministă. Definiţia unui astfel de automat este:

ALM = (Q, T, m, s, F)

unde:

- Q este o mulţime finită de stări
- T este alfabetul benzii de intrare, #, L, R $\notin$ T
- m este funcţia de tranziţie

  $m : Q \times T \rightarrow P(Q \times (T \cup \{L, R\}))$.
- s $\in$ Q este starea iniţială pentru automatul liniar mărginit.
- F $\subseteq$ Q este mulţimea stărilor finale

Dacă q $\in$ Q este o stare, a $\in$ T este un simbol de pe banda de intrare şi (p, b) $\in$ m(q, a) înseamnă că din starea curentă q pentru simbolul de intrare a se poate trece în starea p, înlocuînd sibmbolul de pe banda de intrare sau efectuând o mişcare la stânga sau la dreapta. Definiţia este a unui acceptor nedeterminist. Funcţionarea automatului porneşte cu o configuraţie având pe banda de intrare un şir cuprins între doi simboli # (două blancuri). Deoarece funcţia de tranziţie nu este definită pentru simbolul #, înseamnă că spaţiul de mişcare al capului de citire/scriere pe banda d intrare este limitat la lungimea iniţială a şirului de intrare. Cu alte cuvinte dacă printr-o deplasare la stânga sau la dreapta capul de citire/scriere ajunge în afara spaţiului ocupat iniţial de către şirul analizat automatul se opreşte. Conform definiţiei, pentru automatele liniat mărginite se consideră o acceptare prin stări finale. Dacă la oprirea automatului capul de citire se găseşte la capătul din dreapta al şirului şi starea automatului face parte din F înseamnă că evoluţia automatului a dus la acceptare.

Se poate demonstra că pentru orice gramatică dependentă de context există o gramatică echivalentă liniar mărginită. Producţiile unei gramatici în liniar mărginită au în partea dreaptă cel mult doi simboli. Dacă există o producţie S $\rightarrow$ AB (unde S este simbolul de start al gramaticii) atunci A = S. Pentru orice gramatică dependentă de context liniar mărginită se poate construi un automat liniar mărginit care acceptă acelaşi limbaj. De asemenea pentru orice automat liniar mărginit este posibil să se construiască o gramatică dependentă de context care generează acelaşi limbaj. Adică, clasa limbajelor acceptate de automate liniar mărginite este clasa limbajelor dependente de context. Intuitiv, relaţia de echivalenţă dintre gramaticile dependente de context şi automatele liniar mărginite este susţinută de condiţia pe care o satisfac producţiile unei gramatici dependente de context. Şi anume - dacă $\alpha A\beta \rightarrow \alpha\gamma\beta$ este o producţie atunci $|\alpha A\beta| \leq |\alpha\gamma\beta|$. Adică orice şir din limbajul generat de către gramatică este mai lung cel puţin egal cu orice formă propoziţională obţinută în secvenţa de derivări prn care se obţine şirul respectiv.

Se poate construi şi o definiţie de automat liniar mărginit determinist. Automatele liniar mărginite nedeterministe sunt mai puternice decât cele deterministe, în sensul că existe limbaje acceptate de către automate liniar mărginite nedeterministe dar care nu sunt acceptate de automate liniar mărginite deterministe. În particular limbajele independente de context sunt acceptate şi de automatele liniar mărginite deterministe (acceptoarele obişnuite pentru limbajele independente de context sunt automatele cu stivă).

2.3.3.5 Relaţia între maşina Turing şi gramatici

Propoziţie. Fie M = (Q, TM, m, q) o maşină Turing. Există o gramatică G = (N, TG, P, S) astfel încât pentru orice pereche de configuraţii (q, u$\underline{a}$v) şi (q', u'$\underline{a}$'v') ale maşini Turing

$$(q,\ u\underline{a}v) \overset{*}{\underset{M}{\vdash}} (q',\ u'\underline{a}'v')$$

dacă şi numai dacă

$$[uqav] \overset{*}{\underset{G}{\Rightarrow}} [u'q'a'v']$$

Demonstraţie. Se observă că formele propoziţionale ce se obţin în gramatica G conţin şiruri care corespund configuraţilor maşini Turing. Astfel, pentru o configuraţie de forma (q,u,a,w) se obţine o formă propoziţională [uqaw]. Poziţia pe care o ocupa simbolul corespunzător stării indică de fapt poziţia capului de citire pe banda de intrare.

Mulţimea simbolilor terminali pentru G este TG = TM ∪ {[,]}. Mulţimea simbolilor neterminali N = Q ∪ {h, S}. Mulţimea producţiilor se construieşte în modul următor :

i) ∀ q ∈ Q, ∀ a ∈ TM dacă m(q,a) = (p, b) b ∈ T atunci qa → pb ∈ P

îi) ∀ q ∈ Q, ∀ a ∈ TM dacă m(q,a) = (p, R) atunci
qab → apb ∈ P pentru ∀ b ∈ TM şi qa] → ap# ∈ P

iii) ∀ q ∈ Q, ∀ a ∈ TM dacă m(q,a) = (p, L) atunci dacă a ≠ # sau c ≠] atunci
bqac → pbac ∈ P, ∀ b ∈ TM, ∀ c ∈ TM ∪ {]}
dacă a = # atunci
bq#] → pb] ∈ P, ∀ b ∈ TM.

Se observă că în construcţia anterioara simbolul de start al gramatici nu joacă nici un rol.

În toate producţiile se observă ca apare câte un singur neterminal atât în partea dreapta cât şi în partea stânga. Acest neterminal corespunde unei stări. Poziţia sa în cadrul fiecărui şir reprezintă de fapt poziţia capului de citire. Simbolul de start al gramaticii nu este utilizat. Se poate demonstra utilizând această construcţie că :

(q, u, a, v) |- (q', u', a', v')

dacă şi numai dacă

[uqav] ⇒ [u'q'a'v']

Demonstraţia se realizează considerând toate formele posibile de producţii. Mai departe rezultatul se extinde natural pentru închiderile acestor relaţii.

Să considerăm de exemplu maşina Turing

M = ({q0, q1, q2}, {a, #}, m, q0)

pentru care m este definită în modul următor :

```
m(q0, a) = (q1, R)
m(q0, #) = (q0, a)
m(q1, a) = (q2, R)
m(q1, #) = (q1, a)
m(q2, #) = (h,  #)
m(q2, a) = (q2, R)
```

să considerăm pentru această maşină evoluţia pentru configuraţia (q0, #a#)

```
(q0, #a#)  |- (q0, aa#)
           |- (q1, aa#)
           |-+ (h, aa#).
```

Să construim gramatica corespunzătoare :

G = ({q0, q1, q2, h, S}, {q, #, [,]}, P, S)

P va conţine următoarele producţii :

i) în cazul în care se face scriere pe banda

```
q0# → q0 a ( corespunde m(q0, #) = (q0, a) )
q1# → q1 a ( corespunde m(q1, #) = (q1, a) )
q2# → h  # ( corespunde m(q2, #) = (h,  #) )
```

îi) în cazul în care se fac deplasări dreapta

```
q0aa → a q1 a ( corespunde m(q0, a) = (q1, R))
q0a# → a q1 #
q0a] → a q1 #]

q1aa → a q2 a (corespunde m(q1, a) = (q2, R))
q1a# → a q2 #
q1a] → a q2 # ]

q2aa → a q2 a (corespunde m(q2, a) = (q2, R))
q2a# → a q2 #
q2a] → a q2 #]
```

Un calcul realizat de M poate să fie :

```
(q0, #) |- (q0, a#) |- (q1, a#) |- (q1, aa# |- (q2, aa#)
        |- (h,  aa#)
```

O derivare în G care porneşte din şirul [q0#] este :

```
[q0#] ⇒ [q0a] ⇒ [aq1#] ⇒ [aq1a] ⇒ [aaq2#] ⇒
          [aah#]
```

Se observă că în aceasta derivare simbolul de start al gramatici nu a intervenit. Gramatica este în acest caz utilizată ca mecanism de specificare a unor substituţii de şiruri şi nu ca generator de limbaj.

Aşa cum o maşină Turing a fost tratată ca acceptor de limbaj o gramatica poate să fie privită ca un dispozitiv capabil să reprezinte (efectueze) calcule.

Fie T1 şi T2 două mulţimi de tip alfabet care nu conţin simbolul # şi fie $f : T1^* \to T2^*$, spunem că f este calculabilă gramatical dacă şi numai dacă există o gramatica G = (N, T, P, S) astfel încât $T1, T2 \subseteq T$ şi există şirurile x, y, x', y' $\in (N \cup T)^*$ care îndeplinesc următoarea condiţie pentru $\forall$ u $\in T1^*$ şi $\forall$ v $\in T2^*$ v = f(u) dacă şi numai dacă xuy $\Rightarrow^*$ x'vy'. În acest caz se spune că G calculează funcţia f. De la şiruri definiţia se poate extinde la numere naturale într-o manieră similară celei utilizate la maşina Turing.

Să considerăm de exemplu T1 = {a, b} şi $f : T1^* \to T1^*$, $f(w) = w^R$. Funcţia f este calculabilă de către gramatica G = (N, T, P, S} unde N = {A, B, S}, T = {a, b, *, [,]} iar P conţine următoarele producţii :

```
1. [a → [A, [b → [B
2. Aa → aA, Ab → bA, Ba → aB, Bb → bB
3. A* → *a, B* → *b
```

şirurile x, y, x' şi y' sunt [, *], [* şi respectiv]. G transformă un şir de forma [w*] cu w ∈ {a, b}* în modul următor. Se transformă întâi primul simbol (a sau b) cu care începe şirul în A sau B utilizând una dintre producţiile [a → [A sau [b → [B. În continuare neterminalul astfel obţinut migrează spre limita din dreapta a şirului. La întâlnirea simbolului * se aplică una dintre producţiile A* → *a sau B* → *b. De exemplu pentru şirul abb se obţine :

```
[abb*] ⇒ [Abb*] ⇒ [bAb*] ⇒ [BAb*] ⇒ [BbA*] ⇒
           [Bb*a] ⇒ [bB*a] ⇒ [b*ba] ⇒ [B*ba] ⇒
           [*bba]
```

Se observă că nu este semnificativă ordinea în care se aplică producţiile.

Propoziţie Orice funcţie calculabilă în sens Turing este calculabilă şi gramatical.

Demonstraţia acestei propoziţii este constructivă, adică pornind de la o maşina Turing se construieşte o gramatică care realizează acelaşi calcul.

Din rezultatele prezentate putem trage concluzia echivalenţei între gramaticile de tip 0 (fără restricţii) şi maşinile Turing din punctul de vedere al limbajelor acceptate (generate) şi al puterii de calcul.

Considerând echivalenţa dintre gramatici şi maşinile Turing se pune întrebarea cum se poate construi o maşină Turing care să accepte limbajul generat de o gramatica G = (N, T, P, S). Având în vedere ca operaţia de derivare presupune de fapt prelucrări simple de şiruri de simboli, se poate construi o maşină Turing care să execute operaţia de derivare. O astfel de maşină se poate obţine de exemplu utilizând câte o maşină Turing pentru fiecare producţie. Având în vedere modul în care se obţine un şir

care face parte din L(G) se poate construi o maşină Turing eventual nedeterministă care să realizeze aceleaşi operaţii.

2.3.3.6 Elemente de calculabilitate

Discuţia conţinută în acest subcapitol depăşeşte cadrul limbajelor formale dar este importantă pentru a realiza legătura dintre modelele de calcul considerate şi lumea reală a calculatoarelor, mai ales pentru a oferi o caracterizare a posibilităţilor oferite de acestea.

În momentul în care s-a dat definiţia limbajelor s-a precizat faptul că numai o anumită parte a acestora poate să fie descrisă într-o formă finită. Gramaticile, automatele finite, automatele push down, maşina Turing sunt astfel de reprezentări finite. Se pune problema dacă odată cu maşina Turing s-a atins într-adevăr limita de reprezentare finită a limbajelor. De asemenea, având în vedere că o gramatica poate să fie interpretată şi ca un dispozitiv capabil să efectueze calcule, se pune întrebarea dacă nu există mecanisme mai puternice de reprezentare decât gramaticile şi maşina Turing (având în vedere faptul că un calculator real pare mult mai puternic decât o maşina Turing).

2.3.3.6.1 Maşina Turing Universală

Spre deosebire de o maşina Turing standard, modelul Von Neumann de calculator presupune noţiunea de program memorat. Ar apare aici o diferenţă importantă între o maşină Turing şi un calculator real. Şi anume o maşină Turing standard este dedicată rezolvării unei anumite probleme (calculului unei anumite funcţii) în timp ce un calculator este un dispozitiv universal, capabil să treacă la calculul unei noi funcţii prin schimbarea programului, deci a ceva conţinut în memorie, fără a schimba însă modul de funcţionare a unităţii de control. Se pune întrebarea dacă nu se poate construi o maşina Turing care să funcţioneze într-o manieră similară adică, să primească la intrare atât şirul care se transformă cât şi o descriere a transformării. O astfel de maşină Turing se numeşte maşina Turing universală. Pentru a construi o maşină Turing universală este necesar să obţinem întâi un mod de descriere al unei maşini Turing sub forma de şir astfel încât această descriere să poată să fie utilizată ca intrare pentru o alta maşină Turing .

Fiecare maşină Turing este definită utilizând patru elemente MT = (Q, T, m,s). Deoarece Q şi T sunt mulţimi finite descrierea unei maşini Turing poate să fie făcută sub forma unui şir utilizând simboli din Q, T, paranteze, virgule şi alte semne de punctuaţie. O asemenea reprezentare sub forma de şir, nu este recomandabilă deoarece avem nevoie pentru maşina pe care o construim de un alfabet finit pe care să îl utilizăm în definiţia acesteia, alfabet care să permită reprezentarea oricărui alfabet posibil pentru o maşină Turing. Vom consideră ca există mulţimile infinit numărabile :

Q_{inf} = {q1, q2, ...} şi T_{inf} = {a1, a2, ...}.

astfel încât pentru orice maşină Turing, mulţimea stărilor şi respectiv alfabetul de intrare sunt submulţimi finite din Q_{inf} şi respectiv T_{inf}. Se consideră următoarea codificare pentru elementele care aparţin unei definiţii de maşină Turing utilizând un alfabet ce conţine un singur simbol {I}:

element	cod(element)
q_i	I^{i+1}
h	I
L	I
R	II
a_i	I^{i+2}

Se observă că fiecare stare $q \in Q \cup \{h\}$ are un cod unic, acelaşi lucru este valabil şi pentru fiecare simbol din alfabetul de intrare. Fie c un simbol $c \neq I$. Pentru construirea şirului care descrie o maşină Turing se utilizează numai simbolii c şi I. Fie MT = (Q, T, m, s) o maşină Turing cu $Q \subset S_{inf}$ şi $T \subset T_{inf.}$ Deci $Q = \{q_{i1}, q_{i2}, ..., q_{ik}\}$, $i1 < i2 < ... < ik$, iar $T = \{a_{j1}, a_{j2}, ... a_{jl}\}$. Construim kl şiruri notate cu S_{pr}, unde $1 \leq p \leq k$ şi $1 \leq r \leq l$. Fiecare şir S_{pr} codifică valoarea funcţiei de tranziţie pentru o pereche (q_{ip}, a_{jr}). Fie de exemplu $m(q_{ip}, a_{jr}) = (q', b)$, $q' \in Q \cup \{h\}$ şi $b \in T \cup \{L, R\}$ atunci S_{pr} = cw1cw2cw3cw4c unde :

```
w1 = cod(q_ip)
w2 = cod(a_jr)
w3 = cod(q')
w4 = cod(b).
```

Notăm cu codif(M) şirul $cS_0cS_{11}S_{12} ... S_{1l}S_{21} ... S_{2l} ... S_{k1} S_{k2} ... S_{kl}$ c. S0 este codificarea stării iniţiale, S_0 = cod(s). Se observă că şirul astfel obţinut reprezintă o codificare unică pentru o maşina Turing. De asemenea pe baza unui astfel de şir se poate reconstrui descrierea "clasica" a maşinii.

Să considerăm de exemplu MT = (Q, T, m,s) cu Q = {q2}, T = {a1, a3, a6}, s = q2 şi m(q2, a3) = m(q2, a6) = (q2, R) şi m(q2, a1) = (h,a3). Utilizând notaţiile anterioare, rezultă k = 1, i1 = 2, l = 3 şi j1 = 1, j2 = 3, j3 = 6. Se obţine :

S11	m(q2, a1) = (h, a3)	cIIIcIIIcIcIIIIIc
S12	m(q2, a3) = (q2,R)	cIIIcIIIIIcIIIcIIc
S13	m(q2, a6) = (q2, R)	cIIIcIIIIIIIIcIIIcIIc

Rezultă

$$codif(M) = cI^3c|cI^3cI^3cIcI^5c|cI^3cI^5cI^3cI^2c|cI^3cI^8cI^3cI^2c|c$$

O maşină Turing universală U primeşte pe banda de intrare un şir de tip codif(M) şi un şir w pe care trebuie să funcţioneze M. Şirul w trebuie să fie şi el codificat utilizând alfabetul {c, I}. Codificarea se face în modul următor. Dacă w = b1 ... bn, cu bi $\in T_{inf}$ atunci:

```
codif(w) = c cod(b1) c cod(b2) c ... c  cod(bn) c.
```

Se observă că şirul obţinut codif(w) nu poate să conţină simboli #, chiar dacă w conţine simboli #. U = (Qu, {I, c, #}, mu, qu) trebuie să satisfacă următoarele condiţii - pentru orice maşină Turing , M = (Q, T, m, s) :

1. dacă (h, u$\underline{a}$w) este o configuraţie de oprire pentru M astfel încât (s, #w$\underline{\#}$) |-^{*M} (h, u$\underline{a}$v) atunci, (qu, #codif(M) codif(w) #) |-^{*U} (h, #codif(uav)#)
2. dacă (qu, #codif(M)codif(w)$\underline{\#}$) |-^{*U} (h,u'$\underline{a}$'v') pentru o configuraţie de oprire (h, u'$\underline{a}$'v') pentru U atunci a' = #, v' = λ, u' = #codif(uav) pentru u, a, v astfel încât (h, u$\underline{a}$v) este o configuraţie de oprire pentru M şi (s, #w$\underline{\#}$) |-^{*M} (h, u$\underline{a}$v).

Adică dacă M se opreşte pentru w atunci şi U se opreşte pentru un şir obţinut prin concatenarea şirurilor codif(M) şi codif(w). Mai mult, pentru o astfel de oprire conţinutul benzii U reprezintă o codificare a răspunsului maşini Turing, M pentru w. De asemenea dacă U se opreşte pentru un şir care reprezintă codif(M)codif(w) atunci şi M se va opri pentru w şi rezultatul lăsat de U reprezintă codificarea rezultatului lăsat de M.

Pentru a simplifica discuţia vom consideră o variantă U' a maşini U, variantă care utilizează trei benzi. Conform celor menţionate anterior pentru aceasta variantă se poate construi o maşină Turing echivalentă cu o singură bandă. Prima bandă va conţine iniţial codif(M)codif(w), a doua bandă conţine în timpul simulării codif(M) iar a treia bandă va conţine codificarea stării curente a maşini M care se simulează. Funcţionarea începe cu prima bandă conţinând codif(M)codif(w), celelalte benzi find goale. U' va copia şirul codif(M) pe a doua bandă şi va modifica conţinutul primei benzi la forma #c codif(#w#). Se observă că localizarea începutului şirului codif(w) în şirul iniţial se face uşor pentru că aici există trei simboli c consecutivi. Din codif(M) se identifică S_0 (codul pentru starea iniţială) şi se copiază pe cea de a treia bandă. În continuare U' începe să simuleze funcţionarea maşini Turing M. Între paşii de simulare cele trei capete de citire / scriere utilizate (există trei benzi de intrare) sunt poziţionate în modul următor :

- pentru prima bandă pe ultimul c care indică sfârşitul codului pentru simbolul curent parcurs de M;
- pentru a doua şi a treia bandă la marginea stânga a benzilor respective.

În acest mod când M porneşte să trateze ultimul simbol # din şirul de intrare #w#, U' va începe simularea deplasându-şi capul pe ultimul simbol c de pe banda. U' va căuta pe a doua bandă un subşir de forma $ccI^icI^jcI^kcI^lcc$ unde I^i este şirul conţinut în a treia banda, iar I^j este şirul curent de pe prima banda. La identificarea unui astfel de subsir U' se va mişca corespunzător. Dacă I^l este codif(L) sau codif(R) atunci se realizează deplasarea pe prima banda de intrare la stânga respectiv la dreapta. Dacă I^l este cod(a) pentru a $\in T_{inf}$ se observă că înlocuirea simbolului curent poate să presupună deplasarea conţinutului primei benzi pentru ca spaţiul ocupat de codul pentru a poate să fie diferit de cel ocupat de simbolul înlocuit. În cadrul aceleiaşi mişcări U' înlocuieşte şirul înscris în a treia bandă cu I^k. Dacă acum conţinutul acestei benzi este cod(h) U' va deplasa capul de pe prima banda pe primul simbol # aflat la dreapta poziţiei curente după care se opreşte. Dacă pe banda a treia nu se găseşte cod(h) atunci se continuă simularea.

Rezultă deci că o maşină Turing este suficient de puternică pentru a fi comparata cu un calculator real. De fapt o maşină Turing este echivalentă cu noţiunea intuitivă de algoritm. Mai

mult conform ipotezei Church (Church's Thesis) nici o procedură de calcul nu este un algoritm dacă nu poate să fie executată de o maşina Turing. Acest rezultat care este acceptat, în prezent constituie numai o ipoteză pentru că nu constituie un rezultat matematic, ci indică numai faptul că un anumit obiect matematic (maşina Turing) este echivalent cu un concept informal.

Revenind la problema reprezentări finite a limbajelor se pune problema cum arată limbajele care nu sunt Turing decidabile. Cu alte cuvinte putem să dăm o reprezentare finită pentru un astfel de limbaj ?. Dacă aşa ceva este posibil înseamnă că putem să construim un algoritm care să decidă dacă un şir face parte din limbajul respectiv. Conform ipotezei Church însă noţiunea de algoritm şi cel de maşina Turing sunt echivalente, Rezultă deci că ar există o maşina Turing care să verifice dacă şirul face parte din limbaj, adică limbajul ar fii Turing decidabil ceea ce contrazice ipoteza.

Să discutăm puţin şi relaţia dintre limbajele Turing acceptabile şi limbajele Turing decidabile. Se poate demonstra următorul rezultat:

Propoziţie. Orice limbaj L Turing decidabil este şi Turing acceptabil.
Demonstraţie. Fie M o maşină Turing care decide L. Atunci se poate construi M' care să accepte L.

```
   N /|
> M L   ------→ N |
      \|
```

Se observă că M' simulează M după care studiază răspunsul obţinut pe bandă. Dacă răspunsul este N, M' va intra într-un ciclu infinit.

Se pune întrebarea care este raportul invers între limbajele acceptabile şi cele decidabile Turing.

Dacă ar fi posibil ca pentru orice maşina Turing şi orice şir de intrare w, să prevedem dacă maşina Turing se va opri pentru w, atunci orice limbaj Turing acceptabil ar fi şi Turing decidabil, deoarece dacă M1 este maşina Turing care acceptă L putem să construim M2 care decide L în modul următor. M2 va executa calculele necesare pentru o prevedea dacă M1 se opreşte pentru w. Corespunzător M2 se va opri cu un simbol D sau N pe banda de intrare după cum M1 acceptă sau nu şirul w.

Rezultă că cele doua probleme:

- este orice limbaj Turing acceptabil şi Turing decidabil ?
- se poate construi o maşină Turing care să prevadă pentru orice maşină Turing, M, dacă aceasta se opreşte sau nu pentru un şir, w ?

sunt echivalente.

Am arătat că dacă se poate construi o maşină Turing care să prevadă pentru orice maşină Turing dacă aceasta se opreşte sau nu pentru orice şir atunci orice limbaj acceptabil este şi decidabil. Să presupunem acum că orice limbaj acceptabil este şi decidabil. În acest caz limbajul:

```
L0 = { codif(M)codif(w) | M accepta w}
```

care este acceptat de maşina Turing universală, este un limbaj decidabil. Fie M^+ maşina Turing care decide acest limbaj. În acest caz M^+ poate să prevadă pentru orice maşină Turing dacă aceasta se opreşte sau nu pentru orice şir w.

Vom arăta în continuare că L_0 nu este decidabil. În mod corespunzător răspunsul la cele doua probleme enunţate anterior este : NU.

Dacă L_0 este Turing decidabil atunci limbajul :

```
L1 = { codif(M) | M accepta codif(M) }
```

este Turing decidabil. Dacă M_0 este maşina care decide L_0 atunci se poate construi maşina M_1 care să decidă L_1. Funcţionarea M_1 constă din transformarea intrării din #w# în #wcodif(w)# pe care va funcţiona apoi M_0. Deci M_1 va calcula acelaşi rezultat asupra şirului #w# ca şi M_0 asupra şirului #wcodif(w)#. Conform definiţiei limbajului L_0, M_0 va scrie D pe banda dacă şi numai dacă :

a. w este codif(M) pentru o maşină Turing
b. maşina Turing M, acceptă şirul w, adică şirul de intrare codif(M).

Dar asta este exact definiţia limbajului L_1. Rezultă că este suficient să arătăm că L1 nu este Turing decidabil. Să presupunem că L_1 este Turing decidabil în acest caz şi limbajul

```
 L̅1 = { w ∈ {I,c}* | w nu este codif(M) pentru nici o maşină Turing
sau w = codif(M) pentru o maşina Turing, M,
care nu acceptă sirul codif(M) }
```

este decidabil. O maşină Turing care decide $\overline{L}_1$ se obţine din maşina Turing care decide L_1 inversând la oprirea maşinii răspunsurile D şi N.

Limbajul $\overline{L}_1$nu este însă Turing acceptabil. Să presupunem că există o maşina Turing M* care accepta $\overline{L}_1$. Din definiţia limbajului $\overline{L}_1$ rezultă că:

$\forall$ M, codif(M) $\in$ $\overline{L}_1 \Leftrightarrow$ M nu acceptă codif(M) (1)

dar, M* a fost construită pentru a acceptă $\overline{L}_1$, adică:

codif(M*) $\in$ $\overline{L}_1$ => M* acceptă codif(M*) (2)

Din (1) şi (2) rezultă

M* nu acceptă codif(M*) $\Rightarrow$ M* acceptă codif(M*)

adica M* nu poate să existe, adică $\overline{L}_1$, L_1 şi deci L_0 nu sunt Turing decidabile.

Deoarece L_0 nu este Turing decidabil nu există un algoritm care să stabilească dacă o maşină Turing oarecare se opreşte pentru un şir oarecare. Se observă că am găsit o problema pentru care nu avem o soluţie sub formă de algoritm. Astfel de probleme se numesc nedecidabile (nerezolvabile). De fapt L_0 descrie cea mai cunoscută problema nedecidabila şi anume problema opriri maşini Turing. Enunţul acestei probleme este : "să se determine pentru o maşină Turing M, arbitrară dacă se opreşte pentru un şir de intrare dat ".

Alte probleme nedecidabile echivalente de fapt cu această problemă sunt :

- dându-se o maşină Turing, M, se opreşte M pentru şirul vid ?
- dându-se o maşina Turing , M, există un şir pentru care M se opreşte ?
- dându-se o maşină Turing, M, se opreşte M pentru orice şir de intrare ?
- dându-se două maşini Turing, M1 şi M2 se opresc ele pentru acelaşi şir de intrare ?
- dându-se o maşină Turing, M este limbajul acceptat de M regulat, independent de context, Turing decidabil ?
- dându-se două gramatici G1 şi G2 arbitrare L(G1) = L(G2) ?
- dându-se o gramatică G arbitrară este L(G) =∅ ?
- dându-se o gramatică G independentă de context este G ambiguă ?

2.3.3.7 . Maşina Turing cu limită de timp

Chiar dacă o problemă poate să fie rezolvată cu ajutorul unei maşini Turing dacă timpul de rezolvare este mult prea mare înseamnă că practic problema nu poate să fie rezolvabilă cu ajutorul unei maşini Turing. Unul dintre exemplele clasice pentru o astfel de enunţ este problema comis-voiajorului. Ideea este că un comis-voiajor trebuie să viziteze 10 oraşe. Se cunosc distanţele dintre oraşe şi se cere să se construiască un intinerariu de lungime minimă care să permită comis-voiajorului să viziteze toate oraşele. Evident, problema este rezolvabilă. Numărul de drumuri posibile este finit (9!) şi deci o parcurgere a tuturor va produce o soluţie. Să considerăm că 9! este încă o valoare siportabilă, dar dacă am vorbi de 30 de oraşe. În acest caz numărul de posibilităţi este mai mare decât 10^{30}. Timpul pentru a a analiza toate aceste soluţii reprezintă mai multe miliarde de vieţi omeneşti. Rezultă deci, că din punct de vedere practic definiţia de problemă rezolvabilă/nerezolvabilă trebuie să fie redefinită. În continuare vom

lucra cu maşini Turing cu mai multe benzi. Se poate demonstra că timpul de decizie pentru un limbaj nu depinde de numărul de benzi utilizate.

Fie **t** o funcţie pe numere naturale. Fie $L \subseteq T0^*$ un limbaj şi M = (Q, T, m, s) o maşină Turing cu k piste având $T0 \subseteq T$. Vom spune că **M decide L în timp t** dacă pentru orice $w \in L$

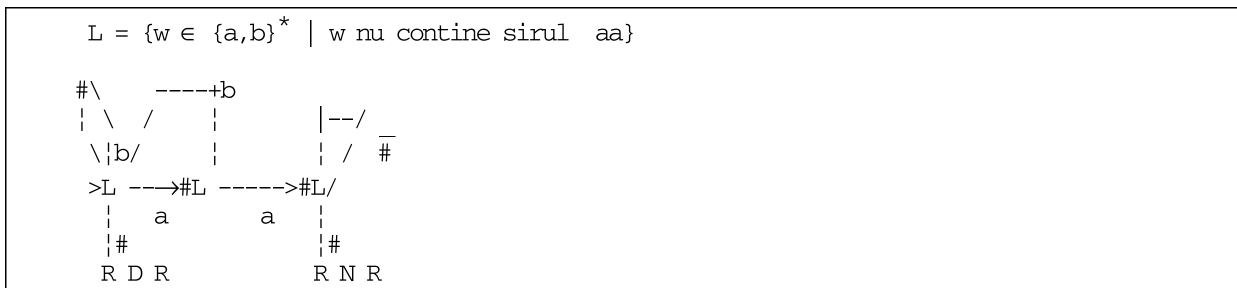
```
(s, #w#, #, ..., #) |-x (h, #D#, #, ..., #)  pentru  x ≤  t(|w|);
```

iar pentru $w \notin L$

```
(s, #w#, #, ..., #) |-x (h, #N#, #, ..., #) pentru  x ≤  t(|w|).
```

Vom spune că L este **decidabil în timp t** dacă există un **k** > 0 şi o maşină Turing cu k piste astfel încât să decidă L în timp **t**. Clasa limbajelor decidabile în timp t este reprezentată de TIME(t).

Rezultă că numărul de paşi executaţi de o maşină Turing pentru un şir dat depinde de lungimea sa. Deoarece maşina Turing trebuie cel puţin să şteargă şirul de intrare, şi să scrie un simbol (D sau N) şi să îşi poziţioneze capul după acest simbol înseamnă că sunt necesare cel puţin |w| + 1 operaţii de scriere şi |w| + 3 operaţii de deplasare. Rezultă că pentru funcţia t utilizată ca limită de timp t(n) > 2n + 4. Să considerăm de exemplu maşina care decide limbajul:



În timp ce maşina parcurge şirul de intrare de la stânga la dreapta îl şi şterge. M va executa numărul minim de paşi, adică M decide L în timp **t**, cu t(n) = 2n + 4 adică $L \in$ TIME(t). Acelaşi lucru se poate scrie sub forma - $L \in$ TIME(2n + 4).

Obiectivul unui studiu de complexitate pentru rezolvarea unei probleme constă în construirea unei maşini Turing care să asigure că decizia limbajului respectiv se va face în maximum t paşi, cu t propus sau dacă asta nu este posibil să se construiască o demonstraţie riguroasă că o astfel de maşină nu se poate construi.

Se poate demonstra că dându-se un limbaj $L \in$ TIME(t) atunci $L \in$ TIME(t') unde $t'(n) = 2n + 18 + \lceil x\, t(n) \rceil$ unde x este o valoare reală orcât de mică ($\lceil r \rceil$ reprezintă cel mai mic număr întreg m astfel încât $m \geq r$). Ceea ce exprimă rezultatul anterior este faptul că ceea ce contează este viteza de creştere şi nu factorii constanţi sau termenii de ordin mai mic.

Fie f şi g două funcţii pe numere naturale. Vom nota cu f = O(g) dacă şi numai dacă există o constantă c > 0 şi un număr întreg n0 astfel încât:

$$f(n) \leq c\, g(n) \text{ pentru } \forall\, n \geq n0.$$

Dacă $f(n) = \sum^{d} a_j n^j$ atunci $f = O(n^d)$. Adică viteza de creştere polinomială este reprezentată de

j = 0

rangul polinomului. În afară de polinoame mai există şi alte funcţii care deşi nu sunt polinomiale sunt mărginite de polinoame. De exemplu $n \lfloor \log_2(n+1) \rfloor = O(n^2)$. Evident creşterile polinomiale sunt de preferat celor exponenţiale (de forma r^n) pentru că se poate demonstra că orice funcţie exponenţială creşte strict mai repede decât una polinomială.

Clasa ***P*** (limbajelor decidabile în timp polinomial) este definită ca:

$$P = \bigcup \{TIME(n^d) \mid d > 0\}$$

Adică este clasa tuturor limbajelor care pot să fie decise de o maşină Turing într-un timp care este mărginit de o funcţie polinomială.

Fie **t** o funcţie pe numere naturale. Fie $L \subseteq T0^*$ şi fie M = (Q, T, m, s) o maşină Turing nedeterministă. Spunem că M acceptă limbajul L în timp **t** nedeterminist dacă pentru $\forall w \in T0^*$,

```
w ∈ L daca si numai daca (s, #w#) |-x (h, vau)
```

pentru $v, u \in T^*$, $a \in T$ şi $x \le t(|w|)$. Vom spune că limbajul L este acceptabil într-un timp **t** nedeterminist dacă există o maşină M care acceptă L în timp t nedeterminist. Clasa limbajelor acceptabile în timp **t** nedeterminist este notată cu NTIME(t). Vom definii

$$NP = \bigcup \{NTIME(n^d) \mid d > 0\}$$

Mai puţin formal, *P* este clasa tuturor mulţimilor pentru care apartenenţa unui element poate să fie testată eficient. Acelaşi lucru nu este adevărat pentru *NP*. Clasa *NP* conţine probleme pentru care apartenenţa la *P* nu a fost demonstrată.

Fie $L1 \subseteq T1^*$ şi $L2 \subseteq T2^*$. O funcţie $g : T1^* \rightarrow T2^*$ este calculabilă în timp polinomial dacă există o maşină Turing M care calculează f în timp t polinomial. O funcţie $g : T1^* \rightarrow T2^*$ calculabilă în timp polinomial. este o transformare polinomială din L1 în L2 dacă şi numai dacă pentru orice $w \in T1^*$, $w \in L1$ dacă şi numai dacă $g(w) \in L2$. Cu alte cuvinte şirurile din L1 pot să fie transformate în şiruri din L2 într-un timp polinomial.

Se spune că un limbaj L este ***NP*-complet** dacă şi numai dacă

(a) $L \in NP$;
(b) pentru $\forall L' \in NP$, există o transformare în timp polinomial din L' în L.

Se poate demonstra că dacă L este ***NP*-complet** atunci *P* = *NP* dacă şi numai dacă $L \in P$. Deocamdată nu s-a demonstrat că $P \neq NP$ sau $P = NP$. Există numeroase probleme "clasice" care sunt *NP*-complete. Pentru aceste probleme de fapt nu se ştie dacă există algoritmi de rezolvare în timp polinomiali. Se consideră că este mai probabil că $P \neq NP$ şi deci că astfel de algoritmi nu există. De obicei dacă pentru o problemă "nouă" se doreşte să se cerceteze apartenenţa la clasa *P* sau *NP* se încearcă construirea unei transformări polinomiale la una dintre problemele cunoscute (de exemplu problema comis voiajorului este o problemă *NP*-completă).

3 2. Analiza lexicală

Rolul analizei lexicale este de a traduce textul programului într-o secvenţă de atomi lexicali. În acest mod se obţine un "text" mai uşor de prelucrat de către celelalte componente ale compilatorului,

deoarece se realizează o abstractizare a unei infinităţi de şiruri de intrare în atomi de tip - identificatori, constante, etc. De asemenea prin eliminarea blancurilor şi a altor separatori irelevanţi (ca de exemplu comentarii), textul prelucrat se poate reduce drastic. De cele mai multe ori analiza lexicală realizează şi alte activitati auxiliare ca de exemplu păstrarea evidenţei numărului de linii citite. O astfel de informaţie este foarte utilă pentru construirea mesajelor de eroare.

Când se proiectează un analizor lexical se pune în general problema care este nivelul de complexitate al atomilor lexicali consideraţi. De exemplu în cazul în care un limbaj utilizează numere complexe, se pune problema dacă analizorul lexical va fi cel care va recunoaşte o constantă de forma:

```
(<real>, <real>)
```

producând un atom lexical corespunzător sau recunoaşterea unei astfel de constante ramâne în sarcina nivelului analizei sintactice.

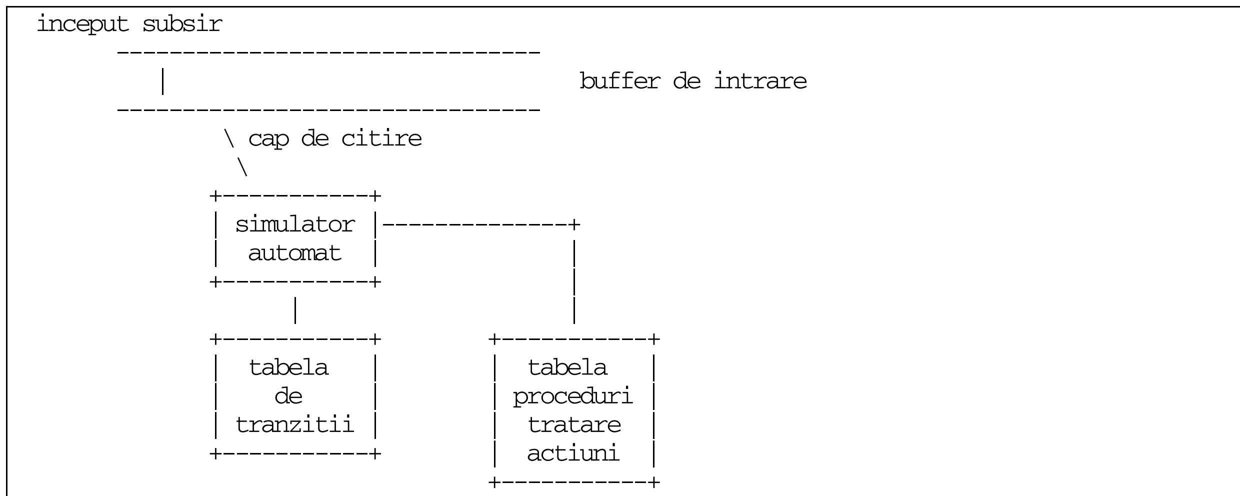
În general un analizor lexical este specificat sub forma :

```
p1   {actiune 1}
p2   {actiune 2}
...
pn   {actiune n}
```

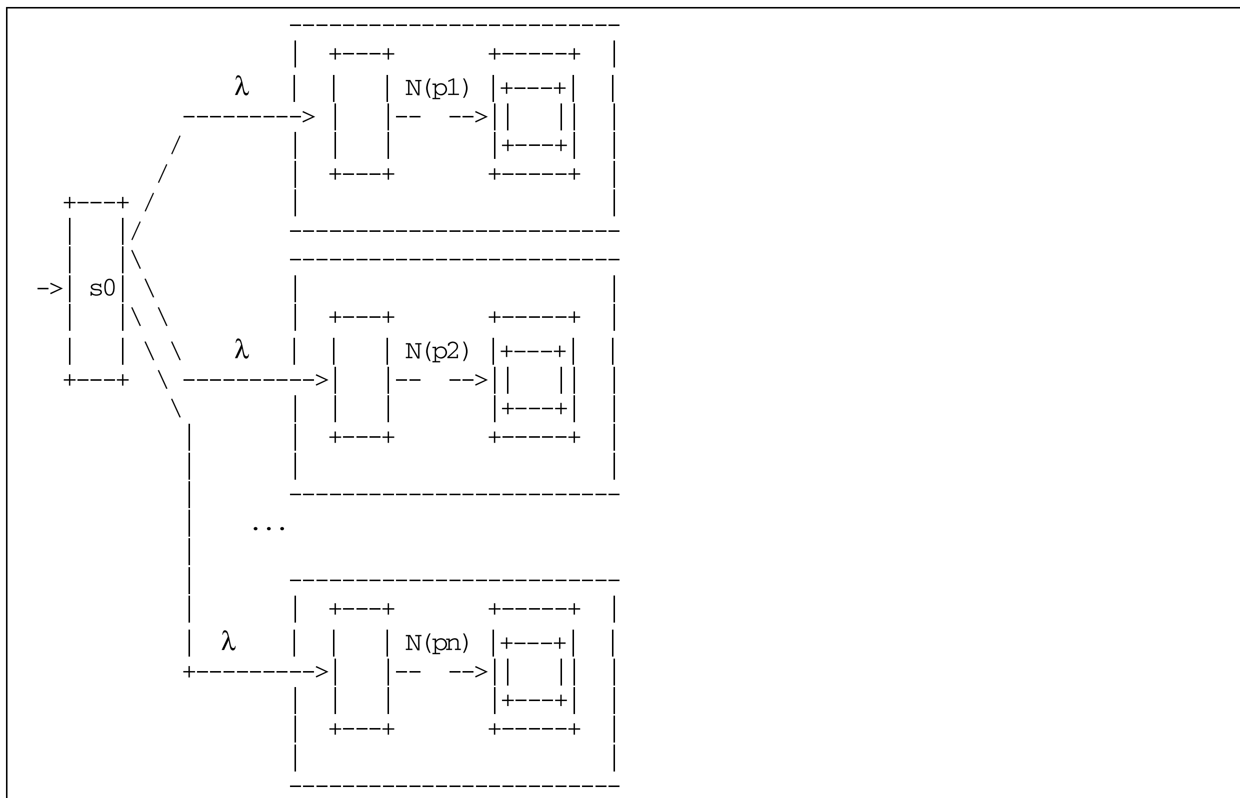
unde p_i este o expresie regulata, iar actiune i este o secvenţă de operaţii care se execută pentru fiecare subşir care corespunde modelului oferit de p_i.

Să considerăm de exemplu şirurile de caractere 99.E10 şi 99.EQ.10, care pot să apara într-un program FORTRAN. În primul caz este vorba de numărul 99 x 10^{10}, în al doilea caz de o expresie relaţională. Dacă analiza se opreşte dupa identificarea punctului zecimal cu concluzia că s-a identificat un număr real se observa că în primul caz oprirea se face prea devreme iar în al doilea caz prea târziu. O soluţie constă din identificarea acestor situaţii realizându-se aşa numita căutare înainte. Astfel, pentru cazul considerat, după recunoaşterea şirului 99.E se mai cercetează cel puţin un caracter pentru a se hotarî dacă analiza trebuie să se incheie la 99 sau trebuie să continue până la 99.E10 . O abordare mai bună consta din căutarea sistematică a celui mai lung subşir care satisface unul din modelele $p_1,...,p_n$. Dacă există mai multe modele de aceaşi lungime care sunt satisfăcute se va alege o convenţie de alegere : de exemplu în ordinea $p_1,...,p_n$.

Un analizor lexical este de fapt implementarea unui automat finit. Ceea ce se schimbă de la un analizor lexical la altul este numai specificarea modelelor căutate şi a acţiunilor asociate.



Să discutăm întâi cum se construieşte un automat finit nedeterminist pornind de la specificațiile analizorului lexical. Se observă că trebuie să se construiasca un AFN care să recunoasca o expresie regulata de forma p1 | p2 | ... | pn. În general automatul care rezultă este :



Pentru a simula acest automat se utilizează o modificare a algoritmului de simulare a AFN pentru a asigura recunoaşterea celui mai lung prefix care se potriveşte unui model. În acest scop se realizează simularea continua până când se obține o mulțime de stari pentru care nu mai există nici o tranziție posibilă pentru simbolul curent. Se presupune că limbajul este astfel încât bufferul de intrare nu poate fi mai scurt decât o lexema (şirul de caractere corespunzător unui atom lexical recunoscut).

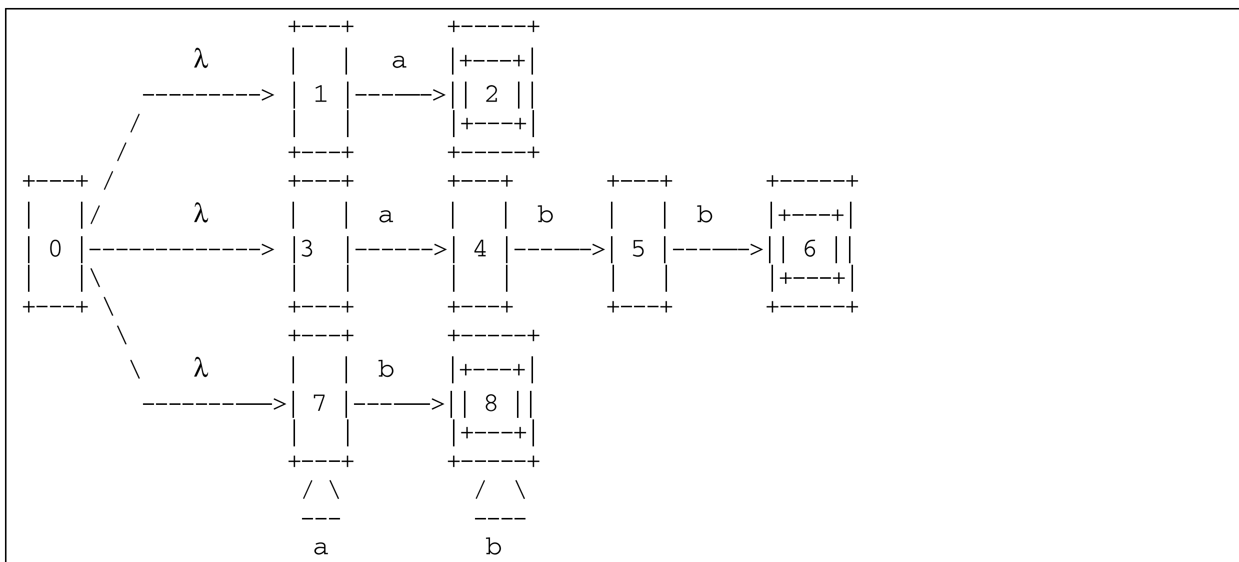
De exemplu orice compilator are o limită (eventual foarte mare) pentru numărul de caractere dintr-un identificator.

Ori de câte ori într-o mulțime de stări se adaugă o stare finală se va memora atât poziția în şirul de intrare cât şi modelul p_i corespunzător. Dacă în mulțimea respectivă există deja o stare finală atunci se va memora numai unul dintre modele (conform convenției stabilite). Când se ajunge în situația de terminare se revine la ultima poziție în şirul de intrare la care s-a facut o recunoaştere a unui model.

Să considerăm de exemplu un analizor lexical specificat în modul următor :

```
a
abb
a*b+
```

Pentru acest analizor nu s-au specificat acțiunile corespunzătoare unităților lexicale. Automatul finit nedeterminist corespunzător este:



Să considerăm comportarea acestui automat pentru şirul de intrare aaba :

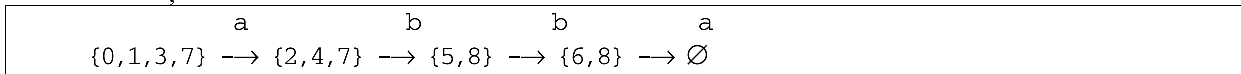
```
          a            a        b        a
{0,1,3,7} ⟶ {2,4,7} ⟶ {7} ⟶ {8} ⟶ ∅
```

Se observă că prima stare de acceptare este 2 în care se ajunge după primul simbol a, apoi va fi găsită starea 8 la care se revine după ce se ajunge la terminare dupa simbolul a de la sfârşitul şirului.

Să considerăm şi varianta care utilizează un automat determinist. Pentru exemplul anterior se obține automatul determinist echivalent descris de următoarea tabela de tranziții:

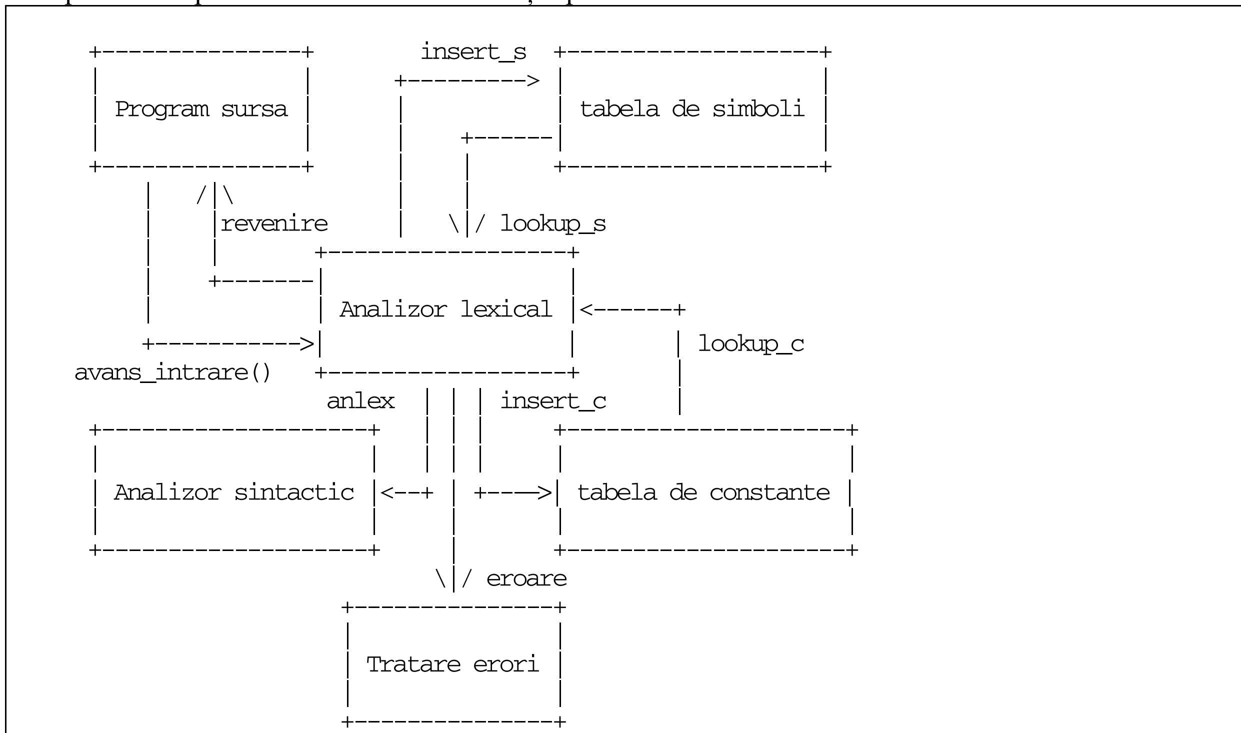
stare	intrare a	intrare b	model recunoscut
{0,1,3,7}	{2,4,7}	{8}	-
{2,4,7}	{7}	{5,8}	a (pentru 2)
{8}	-	{8}	a*b+
{7}	{7}	{8}	-
{5,8}	-	{6,8}	a*b+
{6,8}	-	{8}	abb

Se observă că starea {6,8} conține două stări finale deci recunoaşte atât modelul a^*b^+ cât şi abb. Considerăm prin convenție că model recunoscut în acest caz trebuie să fie abb. De exemplu pentru şirul abba evoluția va fi :



3.1 Interfața analizorului lexical

Dupa cum se ştie analizorul lexical funcționeză ca un modul în cadrul unui compilator sau a oricărui program care realizează prelucrări ce necesită identificarea unor şiruri ce pot fi descrise de o gramatică regulată. În funcționarea sa analizorul lexical interacționează cu celelalte componente ale compilatorului prin intermediul unei interfețe specifice :



Interacţiunea cu fişierul sursă se face prin intermediul a două subprograme : avans_intrare (care preia caracterul următor din programul sursa) şi revenire (care realizeza revenirea înapoi în textul sursă, dacă întotdeauna revenirea se face cu un singur caracter se poate utiliza o funcţie de tip ungetc).

Tabela de simboli este o structură de date al carui rol este de a memora informaţii referitoare la atomii lexicali de tip identificator recunoscuţi de către analizorul lexical. Informaţiile referitoare la acesti simboli sunt utilizate atât în cadrul analizei sintactice cât şi în faza de generare de cod. Introducerea simbolilor în tabela de simboli se face de către analizorul lexical, completarea informaţiilor realizându-se de către analizorul sintactic care va adăuga informaţii ca de exemplu tipul simbolului (procedura, variabila, eticheta, etc) şi modul de utilizare. Legatura cu tabela de simboli se face prin intermediul a doua proceduri : insert_s şi lookup_s. Procedura insert_s are ca argumente un şir de caractere şi codul atomului lexical reprezentat de şir. Rezultatul procedurii este adresa în tabela de simboli la care s-a facut memorarea simbolului. Procedura lookup_s are ca argument un şir iar ca rezultat adresa în tabela de simboli la care se găseşte simbolul reprezentat de şirul respectiv. Dacă în tabela nu există şirul căutat rezultatul este 0.

Pentru cuvintele cheie de obicei se face o tratare speciala, de exemplu se poate initializa tabela de simboli cu intrari corespunzătoare tuturor cuvintelor cheie din limbajul respectiv executând apeluri de forma :

```
insert_s("if", cod_if);
insert_s("else", cod_else);
```

etc. înainte de a se începe execuţia efectivă a compilatorului. În acest caz recunoaşterea cuvintelor cheie se va face apoi într-un mod similar cu a oricarui alt identificator. Se observă deci de ce în majoritatea limbajelor de programare cuvintele cheie nu pot să fie utilizate ca nume cu altă semnificaţie. O astfel de tratare are avantajul ca în cazul în care se face o declaraţie de tip de forma:

```
typedef int intreg;
```

după introducerea în tabela de simboli a cuvintului intreg de către analizorul lexical, analizorul sintactic va putea să completeze apoi intrarea în tabela de simboli cu informaţiile corespunzătoare numelui unui tip. În acest mod în următoarele întâlniri ale acestui cuvânt analizorul lexical va putea să transmită ca ieşire atomul lexical corespunzător unui nume de tip.

În cazul constantelor analizorul lexical realizează recunoaşterea acestora şi memorează valorile corespunzătoare în tabela de constante pentru a permite utilizarea ulterioară în cadrul generării de cod.

Analizorul sintactic apelează de obicei analizorul lexical ca pe o funcţie care are ca valoare codul atomului lexical recunoscut de către analizorul lexical. În general între analizorul lexical şi analizorul sintactic trebuie să se mai transfere şi alte informaţii ca de exemplu adresa în tabela de simboli sau în tabela de constante a unui identificator, cuvântul cheie respectiv constanta identificata de către analizorul lexical. Aceste informaţii reprezintă atributele atomului lexical.

Un compilator recunoaşte erori în toate fazele sale: analiza lexicală, sintactică şi în generarea de cod. Dacă se intâlneşte o eroare, se pune problema localizării sale cât mai precise şi apoi a modului în care se continua analiza astfel încât în continuare să se semnaleze cât mai puţine erori care decurg din aceasta.

Un aspect important al interfetei analizorului lexical o constituie interfaţa cu sistemul de operare corespunzătoare citirii textului programului sursă. În general este bine ca interfaţa cu sistemul de operare să fie cât mai bine separată de restul compilatorului pentru a se obţine un cod portabil prin păstrarea aspectelor dependente de maşină în cadrul funcţiilor care formează aceasta interfaţă.

Din timpul consumat de către procesul de compilare o parte foarte importantă se consumă în cadrul analizei lexicale. La rândul sau partea cea mai consumatoare de timp este de obicei partea legată de operaţiile de citire. În mod standard obţinerea unui caracter în execuţia unui program

presupune copierea acestuia în mai multe etape - de pe disc în zona tampon a sistemului de operare, din această zonă în zona prevazută în program, de unde apoi se face copierea în variabila care memorează caracterul curent.

Chiar şi în cazul unui analizor lexical foarte simplu care ar trebui să recunoască şirurile : xxyy, xx şi y dacă în şirul de intrare se întâlneste şirul xxy trebuie să se mai citească încă un caracter pentru a se putea stabili dacă este vorba de atomul lexical xxyy sau de atomii lexicali xx şi y. Se observă ca în acest caz utilizarea unei proceduri de tip ungetc() nu este suficientă.

În general o colecţie de funcţii care asigură citirea textului sursă pentru analizorul lexical trebuie să satisfacă următoarele condiţii:

- funcţiile trebuie să fie cât mai rapide, realizând un număr minim de copieri pentru caracterele parcurse;
- existenţa unui mecanism care să permită examinarea unor caractere în avans şi revenirea pe şirul de intrare;
- să admită atomi lexicali suficient de lungi;

Pentru a obţine o utilizare eficientă a operaţiilor legate de accesul la disc este necesar ca dimensiunea bufferuluisă fie adaptată modului de alocare a spatiului pe disc. Astfel, pentru sistemul de operare MS-DOS utilizarea unui buffer mai mic decât 512 octeţi nu are nici o justificare (o operaţie de citire va citii cel puţin un sector de pe disc). De preferat este ca bufferul să fie un multiplu al unităţii de alocare a spatiului pe disc.

3.2 Un exemplu elementar de analizor lexical

Să considerăm analizorul lexical corespunzător unui translator pentru limbajul descris de urmatoarea gramatică :

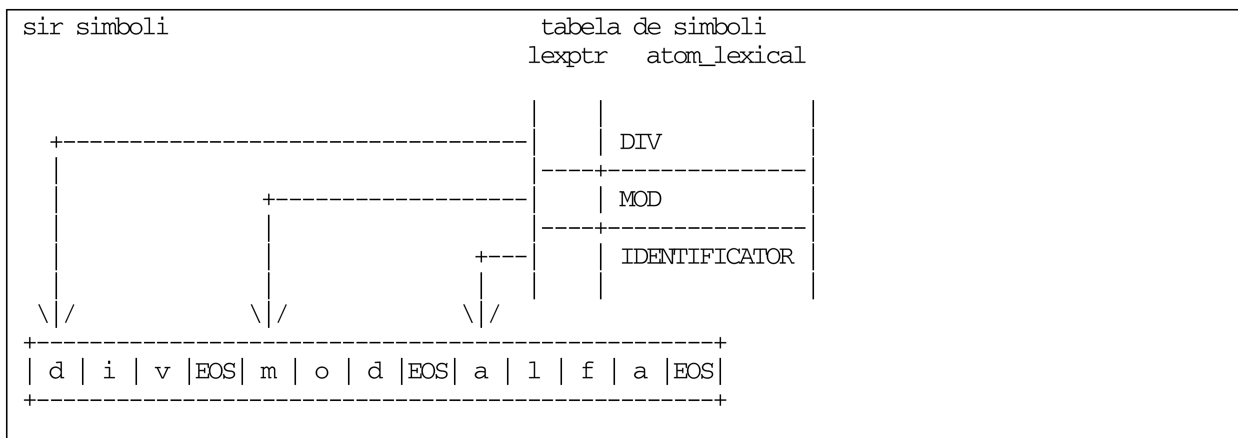
```
S              → lista eof
lista          → expresie; lista | λ
expresie       → expresie + termen | expresie - termen |
                 termen
termen         → termen * factor   | termen / factor   |
                 termen div factor | termen mod factor |
                 factor
factor         → (expresie) | identificator | număr
identificator  → litera rest_id
rest_id        → litera rest_id | cifra rest_id | λ
număr          → cifra număr | cifra
litera         → A | B | ... z
cifra          → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Se observă ca nivelul analizei lexicale printr-o transformare corespunzătoare presupune recunoaşterea următorilor atomi lexicali :

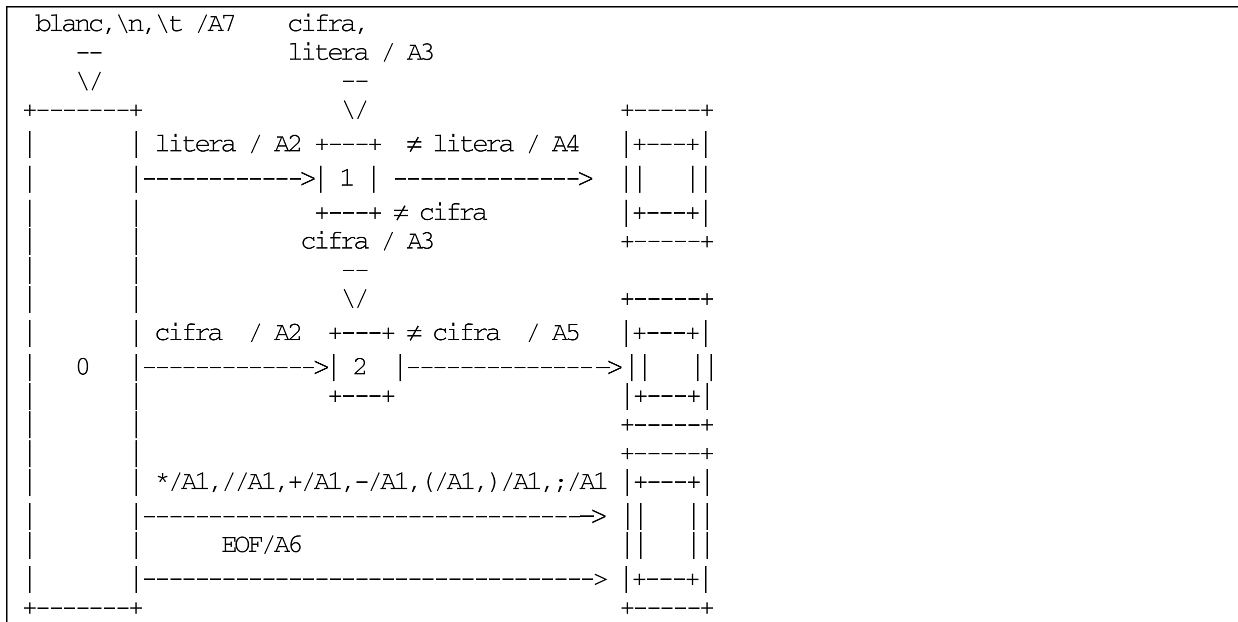
atom lexical	actiune
număr	calcul valoare
identificator	introduce în tabela de simboli
div	
mod	
(	
)	
*	
/	
+	
-	
;	
eof	

În afară de recunoaşterea acestor atomi analizorul lexical realizează şi o eliminare a blancurilor şi a unor caractere speciale de tip '\n' sau '\t'. Având în vedere tipul atomilor lexicali ce trebuie să fie recunoscuţi nu este necesară utilizarea unei interfeţe sofisticate cu sistemul de operare.

Tabela de simboli trebuie să memoreze pentru fiecare identificator (cuvânt cheie) - şirul de caractere respectiv, codul asociat (prin intermediul căruia va fi tratat de către analizorul lexical) şi eventual alte atribute semnificative pe care le poate adăuga analiza sintactică. Deoarece lungimea şirurilor care formează identificatorii poate să fie foarte diferită pentru a se asigura o bună utilizare a memoriei se va utiliza o tabelă de simboli formată dintr-un vector în care se înregistrează şirurile corespunzătoare cuvintelor cheie şi identificatorilor recunoscuţi în textul sursă şi o tabelă cu două sau mai multe colane. În tabelă, fiecare intrare este asociată unui simbol, şi conţine adresa în vectorul care memorează şirurile şi codul asociat simbolului respectiv.



Analizorul lexical poate să fie descris de următorul graf :



În graf au fost evidențiate caracterele şi acțiunile corespunzătoare acestora.

Interfața cu analizorul sintactic se realizează prin intermediul rezultatului apelului analizorului lexical (valoarea funcției anlex()) şi prin variabila tokenval. Valoarea întoarsă de funcția anlex() este codul atomului lexical recunoscut. Variabila tokenval primeşte o valoare care depinde de tipul atomului lexical recunoscut. Astfel pentru un număr această variabilă va conține valoarea numărului iar pentru un identificator adresa în tabela de simboli (la care a fost introdus sau regăsit identificatorul respectiv).

Atomii lexicali care presupun asignarea unor coduri distincte sunt : NUMĂR(256), DIV(257), MOD(258), IDENTIFICATOR(259), GATA(260).

Pentru ceilalți atomi lexicali codul, corespunde codului ASCII al caracterului ce reprezintă atomul lexical respectiv.

Fiecare identificator este asamblat într-o zonă tampon auxiliară buffer_intrare. În momentul în care se încheie construirea unui identificator (se identifică sfârşitul şirului) se poate face căutarea acestuia în tabela de simboli, eventual se va introduce simbolul respectiv în tabela de simboli.

În legatura cu tabela de simboli programul conține trei proceduri :

- init() - realizează introducerea în tabela de simboli a uvintelor cheie
- lookup(s) - caută în tabela de simboli şirul de caractere transmis ca argument. Rezultatul intors de această procedură este adresa relativă în tabela de simboli.
- insert(s,t) - introduce un simbol nou la sfârşitul tabelei de simboli. Al doilea argument de apel este codul atomului lexical reprezentat de şirul respectiv.

Acțiunile ce trebuie să fie executate în legatură cu execuția analizorului lexical sunt următoarele:

```
    A1 : tokenval = NIMIC;
         return caracter;

    A2 : b = 0 ;
```

```
A3 : buffer_intrare[b++] = caracter;
     caracter = getchar();
     if (b >= BSIZE)
       eroare("sir prea lung");

A4 : buffer_intrare[b] = EOS;
     if (caracter != EOF)
       ungetc(caracter, stdin);
     p = lookup(buffer_intrare);
     if (p == 0)
       p = insert(buffer_intrare, IDENTIFICATOR);
     tokenval = p;
     return tabela_simboli [p].atom_lexical;

A5 : buffer_intrare[b] = EOS;
     if (caracter != EOF)
       ungetc(caracter, stdin);
     tokenval = atoi(buffer_intrare);
     return NUMĂR;

A6 : tokenval = NIMIC;
     return GATA;

A7 : if (caracter == ' '|| caracter == '\t')
        ;
       else if (caracter == '\n')
               lineno++;
```

Corespunzător va rezulta următorul program care citeşte şi afişează codurile atomilor lexicali intilniţi.

```
#include "stdlib.h"
#include "stdio.h"
#include "ctype.h"
#include "string.h"

#define BSIZE 128
#define NIMIC -1
#define EOS '\0'

#define NUMAR 256
#define DIV 257
#define MOD 258
#define IDENTIFICATOR 259
#define GATA 260

#define MAXSIR 999
#define MAXSIM 100

/*
        variabile globale
*/

static int  ultimul_caracter = -1;       /* ultimul caracter citit */
static int  ultima_intrare = 0;
static int  tokenval = NIMIC;             /* cod atom lexical */
static int  lineno = 1;                   /* numar linie cod */
static char buffer_intrare[BSIZE];
static int  caracter, p, b;

/*
        tabela de simboli
*/

static char sir_simboli [MAXSIR];       /* sir de nume identificatori */
struct intrare { char *lexptr;
                 int atom_lexical;
               };
static struct intrare tabela_simboli [MAXSIM];

static struct intrare cuvinte_cheie[] = {
        "div", DIV,
        "mod", MOD,
        0,     0
};

/*
    prototipuri functii
*/

static int    lookup(char s[]);
static int    insert(char s[],int tok);
static void   init(void);
static void   a3(void);
static void   eroare (char *m);
static int    lexan(void);

/*
  cauta în tabela de simboli
*/
```

```
static int lookup(char s[])
 {
   int p;
   for (p = ultima_intrare; p > 0; p--)
      if (strcmp(tabela_simboli[p].lexptr, s) == 0)
        return p;
   return 0;
 }

/*
  introducere în tabela de simboli
*/

static int insert(char s[], int tok)
 {
  int len;
  len = strlen(s);
  if (ultima_intrare + 1 >= MAXSIM)
        eroare("s-a depasit dimensiunea tabelei de simboli");
  if (ultimul_caracter + len + 1 >= MAXSIR)
        eroare("s-a depasit dimensiunea tabelei de simboli");
  tabela_simboli[++ultima_intrare].atom_lexical = tok;
  tabela_simboli[ultima_intrare].lexptr =
                  &sir_simboli[ultimul_caracter + 1];
  ultimul_caracter = ultimul_caracter+len +1;
  strcpy(tabela_simboli[ultima_intrare].lexptr,s);
  return ultima_intrare;
 }

/*
  initializarea tabelei de simboli
*/

static void init()
{
   struct intrare *p;
   for (p = cuvinte_cheie; p -> atom_lexical;p++)
     insert(p->lexptr, p->atom_lexical);
}

/*
  tratarea erorilor
*/

static void eroare (char *m)
 {
   printf("line %d: %s\n", lineno, m);
   exit(1);
 }

/*
  analizorul lexical
*/
static void a3()

{ buffer_intrare[b++] = caracter;
  caracter = getchar();
  if (b >= BSIZE)
  eroare("şir prea lung");
```

```
}

static int lexan()
{ while(1)
    { caracter = getchar();
      /* A7 */
      if (caracter == ' '|| caracter == '\t')
           ;                    /* eliminare blancuri si tab-uri */
      else if (caracter == '\n')
             lineno++;
      else if (isdigit(caracter))    /* caracter este cifra */
             { /* A2 */
               b = 0;
               while (isdigit(caracter))
                 /* A3 */
                 a3();
                 /* A5 */
                 buffer_intrare[b] = EOS;
                 if (caracter != EOF)
                   ungetc(caracter, stdin);
                 tokenval = atoi(buffer_intrare);
                 return NUMĂR;
             }
      else if (isalpha(caracter))    /* caracter este litera */
             { /* A2 */
               b = 0;
               while  (isalnum(caracter))  /* litera  sau cifra */
                 /* A3 */
                 a3();
                 /* A4 */
                 buffer_intrare[b] = EOS;
                 if (caracter != EOF)
                   ungetc(caracter, stdin);
                 p = lookup(buffer_intrare);
                 if (p == 0)
                   p = insert(buffer_intrare, IDENTIFICATOR);
                 tokenval = p;
                 return tabela_simboli [p].atom_lexical;
             }
          else if (caracter == EOF)
                 { /* A6 */
                   tokenval = NIMIC;
                   return GATA;
                 }
           else
             { /* A1 */
               tokenval = NIMIC;
               return caracter;
             }
    }
}
void main()
{ int simbol_curent;
  init();
  simbol_curent = lexan();
  while (simbol_curent != GATA)
  { printf(" %d\n",simbol_curent);
    simbol_curent = lexan();
  }
}
```

De fapt orice analizor lexical are aceaşi formă generală, şi construirea unui analizor lexical parcurge întotdeauna aceleaşi etape:

1. construirea expresiilor regulate care descriu atomii lexicali care urmează să fie recunoscuți de către analizorul lexical (nu se poate automatiza)
2. construirea automatului finit nedeterminist care acceptă expresiile regulate (se poate face în mod automat)
3. construirea automatului finit determinist echivalent cu cel nedeterminist construit în pasul anterior (se poate face în mod automat)
4. implementarea unui simulator pentru un automat finit determinist. Acest simulator poate să fie general (adică să poată să fie utilizat pentru orice automat finit determinist), sau poate să fie construit special pentru automatul respectiv.

De fapt în pasul trei se construieşte tabela de tranziție a automatului. Dacă simulatorul este general (adică este capabil să simuleze orice automat specificat prin tabela de tranziții) se observă că începând din pasul al doilea, toate operațiile se pot face în mod automat adică se poate construi un program care pornind de la expresiile regulate şi acțiunile asociate să genereze un analizor lexical. Cel mai cunoscut program de acest tip este **lex**. Inițial acest program a fost utilizat numai sub UNIX. În prezent există variante şi pentru MS-DOS.

Specificațiile **lex** pentru analizorul lexical simplu considerat anterior sunt:

```
%{
#define NOTHING -1
#define NUMBER 256
#define DIV 257
#define MOD 258
#define IDENTIFIER 259
#define FINISH 260

#define MAXSIR 999
#define MAXSYM 100

/*
   global variables
*/

 int  last_character = -1;       /* last character */
 int  last_entry = 0;
 int  tokenval = NOTHING;        /* current token */
 int  lineno = 1;
 int  p;

/*
   symbol table

*/
 char string_array [MAXSIR];       /* string of names */
 struct entry { char *lexptr;
          int token;
         };
 struct entry symbol_table [MAXSYM];

 struct entry reserved_words [] =  {
   "div", DIV,
   "mod", MOD,
   0,     0
};

 void error (char *m)
 { printf("line %d: %s\n", lineno, m);
   exit(1);
 }

/*
  lookup in symbol table
*/

 int lookup(char s[])
 { int p;
   for (p = last_entry; p > 0; p--)
      if (strcmp(symbol_table[p].lexptr, s) == 0)
    return p;
   return 0;
 }

/*
  inserting in symbol table
*/

 int insert(char s[], int tok)
 { int len;
```

```
   len = strlen(s);
   if (last_entry + 1 >= MAXSYM)
    error("too many symbols");
   if (last_character + len + 1 >= MAXSIR)
    error("too many or too long strings");
   symbol_table[++last_entry].token = tok;
   symbol_table[last_entry].lexptr =
             &string_array[last_character + 1];
   last_character = last_character+len +1;
   strcpy(symbol_table[last_entry].lexptr,s);
   return last_entry;
 }

/*
  initialization for symbol table
*/

 void init()
{
   struct entry *p;
   for (p =  reserved_words; p-> token;p++)
     insert(p->lexptr, p->token);
}

%}

/*
  the scanner
*/

delim      [ \t]
nl  [\n]
ws  {delim}+
letter     [A-Za-z]
digit      [0-9]
letgit  [A-Za-z0-9]

%%
div            return DIV;
mod            return MOD;
{letter}{letgit}*   { p = lookup(yytext);
                  if (p == 0)
                        p = insert(yytext, IDENTIFIER);
                 tokenval = p;
                 return symbol_table [p].token;
                }

{digit}+             { tokenval = atoi(yytext);
                  return NUMBER;
                }

{nl}                   lineno++; return NOTHING;
{delim}+             return NOTHING;
<<EOF>>                return FINISH;
.                return yytext[0];

%%
void main()
{ int symbol_current;
  yyin = stdin;
```

```
  init();
  tokenval = NOTHING;
  symbol_current = yylex();
  while (symbol_current !=  FINISH)
   { if (symbol_current != NOTHING)
       printf(" symbol = %i, tokenval = %i\n",
                   symbol_current, tokenval);
     tokenval = NOTHING;
     symbol_current =  yylex();
   }
}
```

4 Analiza sintactică

Rolul fazei de analiza sintactica este de a construi arborii de derivare corespunzători propoziţiilor din limbajul respectiv verificând buna structurare a acestora. Termenul utilizat în literatura de specialitate pentru aceasta acţiune este "parsing".

Considerând gramatica expresiilor aritmetice :

```
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → a
```

şi şirul de intrare a * (a + a) să construim o derivare stânga :

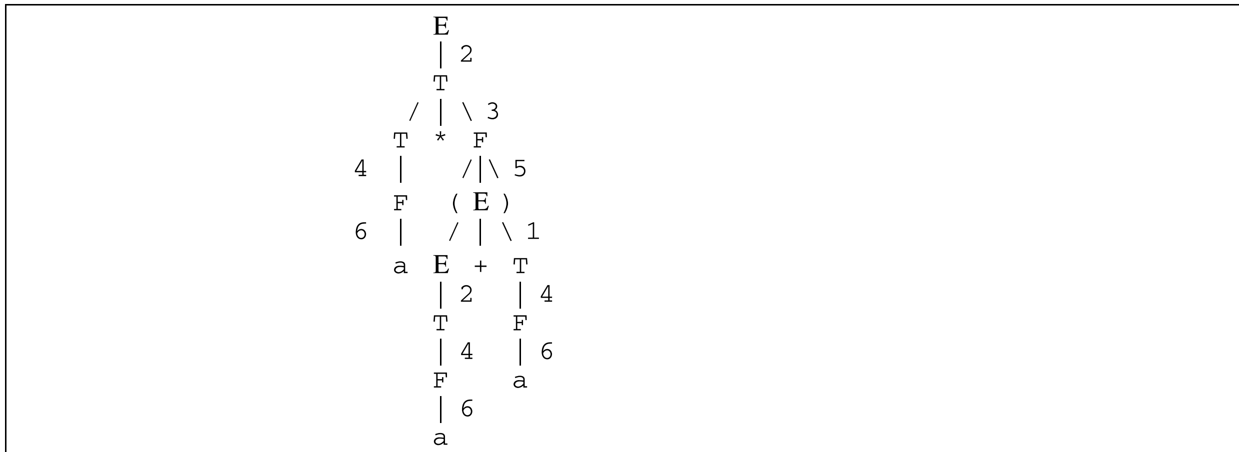
```
     2   3       4       6       5         1
   E ⇒ T ⇒ T * F ⇒ F * F ⇒ a * F ⇒ a * (E) ⇒
                   2             4              6
       a * (E + T) ⇒ a * (T + T) ⇒ a * (F + T) ⇒
                   4             6
       a * (a + T) ⇒ a * (a + F) ⇒ a * (a + a)
```

Lista producţiilor aplicate este 23465124646. Pentru derivarea dreapta se obţine:

```
  2   3     5       1           4
E ⇒ T ⇒ T * F ⇒ T * (E) ⇒ T * (E + T) ⇒
            6           2           4
    T * (E + F) ⇒ T * (E + a) ⇒ T * (T + a) ⇒
            6           4           6
    T * (F + a) ⇒ T * (a + a) ⇒ F * (a + a) ⇒

    a * (a + a)
```

În acest caz lista producţiilor care trebuie să fie aplicate pentru sintetizarea neterminalului de start al gramaticii este 64642641532. Ambele liste de producţii descriu acelaşi arbore de derivare.



Se observă că pentru derivarea stânga este vorba de o parcurgere RSD, iar pentru derivarea drapta parcurgerea arborelui este SDR. Evident dându-se o parcurgere şi producţiile gramaticii se poate reconstrui arborele de derivare.

În general pentru o gramatică independentă de context se poate construi un automat de tip push down care să producă pentru un şir de intrare dat lista de producţii care duce la acceptarea şirului respectiv. În acest scop se defineste notiunea de translator push down. Un astfel de translator se construieşte adaugând o ieşire unui automat push down. La fiecare mişcare a automatului push down, translatorul va emite un şir finit de simboli. Deci definiţia este:

```
PDT = (Q, T, Z, X, m, q0, z0, F)
```

unde X este un alfabet finit de ieşire, iar funcţia m este definită ca $m : Q \times (T \cup \{\lambda\}) \times Z \rightarrow P(Q \times Z^* \times X^*)$. În acest caz o configuraţie a automatului are patru componente : $(q, \alpha, \beta, \gamma)$, $q \in Q$, $\alpha \in T^*$, $\beta \in Z^*$, $\gamma \in X^*$, unde γ reprezintă şirul de simboli generat până la momentul curent de către automat. Se spune că automatul realizează traducerea şirului $\alpha \in T^*$ în şirul $\gamma \in X^*$ dacă este indeplinită condiţia :

$$(q0, \alpha, z0, \lambda) \mid\!-^* (q, \lambda, w, \gamma) \quad q \in F, w \in Z^*$$

În general traducerile realizate de către translator sunt :

$$\{ (\alpha, \gamma) \mid (q0, \alpha, z0, \lambda) \mid\!-^* (q, \lambda, \beta, \gamma) \quad q \in F, \beta \in Z^*, \alpha \in T^*, \gamma \in X^* \}$$

Similar definitţiilor referitoare la automatele push down există noţiunile de translatare prin stivă goală şi translator push down extins. Pentru o gramatică independentă de context putem să construim un translator push down care produce lista producţiiilor corespunzătoare derivărilor stânga şi dreapta.

Să considerăm din nou gramatica expresilor aritmetice. Rezultă translatorul care produce lista producţiilor corespunzătoare derivărilor stânga :

$$(\{q\},\{a,+,*,(,)\},\{E, T, F,a,+,*,(,)\}, \{1, 2, ..., 6\}, m, q, E, \emptyset)$$

cu

```
m(q, λ, E) = {(q, ∈ + T, 1), (q, T, 2)}
m(q, λ, T) = {(q, T * F, 3), (q, F, 4)}
m(q, λ, F) = {(q, (E), 5), (q,a,6)}
m(q, x, x) = {(q, λ, λ)} x ∈ T
```

De exemplu pentru şirul a * (a + a) se obţine evoluţia :

```
(q, a * (a + a), E, λ) |- (q, a * (a + a), T, 2)
                       |- (q, a * (a + a), T * F, 23)
                       |- (q, a * (a + a), F * F, 234)
                       |- (q, a * (a + a), a * F, 2346)
                       |- (q,   * (a + a), * F, 2346)
                       |- (q,     (a + a), F, 2346)
                       |- (q,     (a + a), (E), 23465)
                       |-+ (q, λ, λ, 23465124646)
```

În cazul derivării dreapta se obţine translatorul push down:

```
({q}, {a,+,*,(,)}, {E,T,F,a,+,*,(,),$}, {1,2,...,6}, m, q, $, ∅)
```

cu

```
m(q, λ, E + T) = {(q, E, 1)}
m(q, λ, T)     = {(q, E, 2)}
m(q, λ, T * F) = {(q, T, 3)}
m(q, λ, F)     = {(q, T, 4)}
m(q, λ, (E))   = {(q, F, 5)}
m(q, λ, a)     = {(q, F, 6)}
m(q, x, λ)     = {(q, x, λ)}   ∀ x ∈ T
m(q, λ, $∈)}   = {(q, λ, λ)}
```

Pentru acelaşi şir se obţine secvenţa de mişcări:

```
(q, a * (a + a), $, λ) |- (q, * (a + a), $a, λ)
                       |- (q, * (a + a), $F, 6)
                       |- (q, * (a + a), $T, 64)
                       |- (q, (a + a), $T *, 64)
                       |- (q, a + a), $T * (, 64)
                       |- (q, + a), $T * (a, 64)
                       |- (q, + a), $T * (F, 646)
                       |- (q, + a), $T * (T, 6464)
                       |- (q, + a), $T * (E, 64642)
                       |- (q, a), $T * (E +, 64642)
                       |- (q, ), $T * (E + a, 64642)
                       |- (q, ), $T * (E + F, 646426)
                       |- (q, ), $T * (E + T, 6464264)
                       |- (q, ), $T * (E, 64642641)
                       |- (q, λ, $T * (E), 64642641)
                       |- (q, λ, $T * F, 646426415)
                       |- (q, λ, $T, 6464264153)
                       |- (q, λ, $∈, 64642641532)
                       |- (q, λ, λ,  64642641532)
```

Se observă că cele două translatoare construite sunt nedeterministe. Se pune problema dacă întotdeauna un astfel de translator poate să fie simulat determinist deoarece ceea ce ne interesează este să obţinem analizoare sintactice respectiv compilatoare cât mai eficiente şi cu o comportare deterministă.

Există gramatici pentru care aceasta condiţie nu poate să fie îndeplinită. De exemplu, pentru o gramatică recursivă stânga nu se poate construi un translator care să producă şirul producţiilor dintr-o derivare stânga şi să poata să fie simulat determinist.

Pentru fiecare tip de analizor sintactic(ascendentă sau descendentă) există câte o clasa de gramatici pentru care se poate construi un translator determinist dacă translatorul are acces la următorii k simboli de pe şirul de intrare. Cele doua clase se numesc LL(k) respectiv LR(k). În denumire prima literă (L) reprezintă tipul de parcurgere al şirului - de la stânga (Left) la dreapta. A doua litera (L sau R) specifică tipul derivarii - stânga (Left) sau dreapta (Right). Mişcările translatorului se fac ţinând seama de starea curentă (starea unităţii de control şi conţinut stivă) şi de următorii k simboli de pe şirul de intrare. Să considerăm de exemplu următoarea gramatică:

```
(1) S → BAb
(2) S → CAc
(3) A → BA
(4) A → a
(5) B → a
(6) C → a
```

Se observă că $L(G) = aa^{+}b + aa^{+}c$. Gramatica nu este nici LR(k) şi nici LL(k) deoarece nu se ştie dacă primul a dintr-un şir din L(G) este generat utilizând producţiile 1 şi 5 sau 2 şi 6. Aceasta informaţie este aflată abia când se ajunge la sfârşitul şirului şi se citeşte ultimul simbol. Se observă că de fapt problema este că nu se ştie câte caractere trebuie să fie cercetate în avans pentru a lua o decizie corectă.

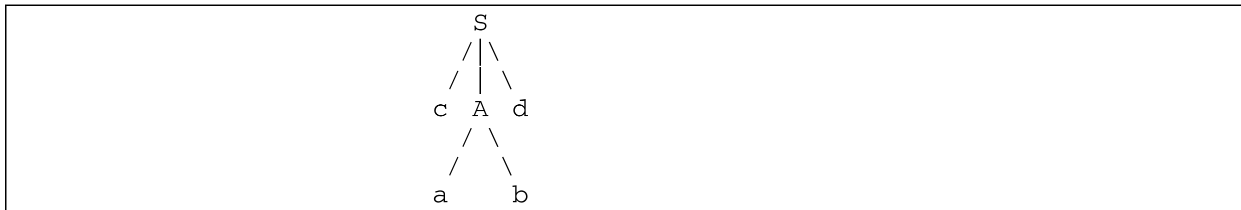
Se poate arăta că dacă o gramatică este LL(k) ea este şi LR(k) dar există gramatici LR(k) care nu sunt LL(k). Cu alte cuvinte gramaticile LL(k) sunt un subset al gramaticilor LR(k).

4.1 Analiza sintactica top - down

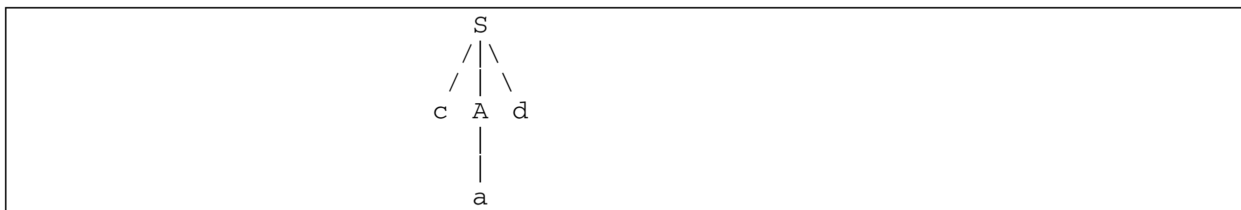
Analiza sintactică top-down poate să fie interpretată ca operaţia de construire a arborilor de derivare pornind de la rădăcină şi adaugând subarbori de derivare într-o ordine prestabilită.

Să considerăm de exemplu gramatica S → cAd, A→ ab | a şi şirul de intrare cad. Construirea arborelui de derivare pentru acest şir porneşte de la simbolul de start al gramaticii S cu capul de citire pe şirul de intrare poziţionat pe caracterul c. Pentru S se utilizează producţia S → cAd şi se obţine arborele de derivare :

În acest moment se observă că frunza cea mai din stânga corespunde cu primul caracter din şirul de intrare. În acest caz se avansează cu o poziţie pe şirul de intrare şi în arborele de derivare. Frunza curentă din arborele de derivare este acum A şi se poate aplica una dintre producţiile corespunzătoare acestui neterminal :



Se poate avansa pe şirul de intrare şi în arbore deoarece avem din nou coincidenţa simboliilor terminali. În acest moment se ajunge la compararea simbolului d din şirul de intrare cu simbolul b din arbore, corespunzător trebuie să se revină cautându-se o nouă variantă pentru A. Rezultă că capul de citire trebuie să fie readus în poziţia simbolului a. Dacă acum se utilizează producţia A → a se obţine arborele :



În continuare se va ajunge la acceptarea şirului de intrare.

Din cele prezentate anterior rezultă două observaţii :

- în general implementarea presupune utilizarea unor tehnici cu revenire (backtracking);
  dacă gramatica este recursivă stânga algoritmul poate conduce la apariţia unui ciclu infinit.

4.1.1 Analiza sintactica predictiva (descendent recursiva)

Dezavantajul abordării anterioare constă în necesitatea revenirilor pe şirul de intrare ceea ce conduce la complicarea deosebită a algoritmilor şi mai ales a structurilor de date implicate deoarece automatul cu stivă corespunzător cazului general trebuie să fie nedeterminist. Există însă gramatici pentru care dacă se utilizeaza transformări, prin eliminarea recursivitatii stânga şi prin factorizare se obțin gramatici care pot să fie utilizate pentru analiza top-down fără reveniri. În acest caz dându-se un simbol de intrare curent şi un neterminal există o singură alternativă de producție prin care din neterminalul respectiv să se deriveze un şir care începe cu simbolul de intrare care urmează.

Pentru proiectarea analizoarelor predictive se utilizează diagrame de tranziție. O diagramă de tranziție este un graf care prezintă producțiile corespunzătoare unui neterminal. Etichetele arcelor reprezintă atomi lexicali sau simboli neterminali. Fiecare arc etichetat cu un atom lexical indică tranziția care urmează să se execute dacă se recunoaşte la intrare atomul lexical respectiv. Pentru arcele corespunzătoare neterminalelor se vor activa procedurile corespunzătoare neterminalelor respective.

Pentru a construi o diagrama de tranziție pentru un neterminal A într-o gramatica în care nu există recursivitate stânga şi în care s-a făcut factorizare stânga se procedează în modul următor :

1. se crează două stări: inițială şi finală;
2. pentru fiecare producție A → X1 X2 ... Xn se va crea o cale formată din arce între starea inițială şi starea finală cu etichetele X1 X2 ... Xn.

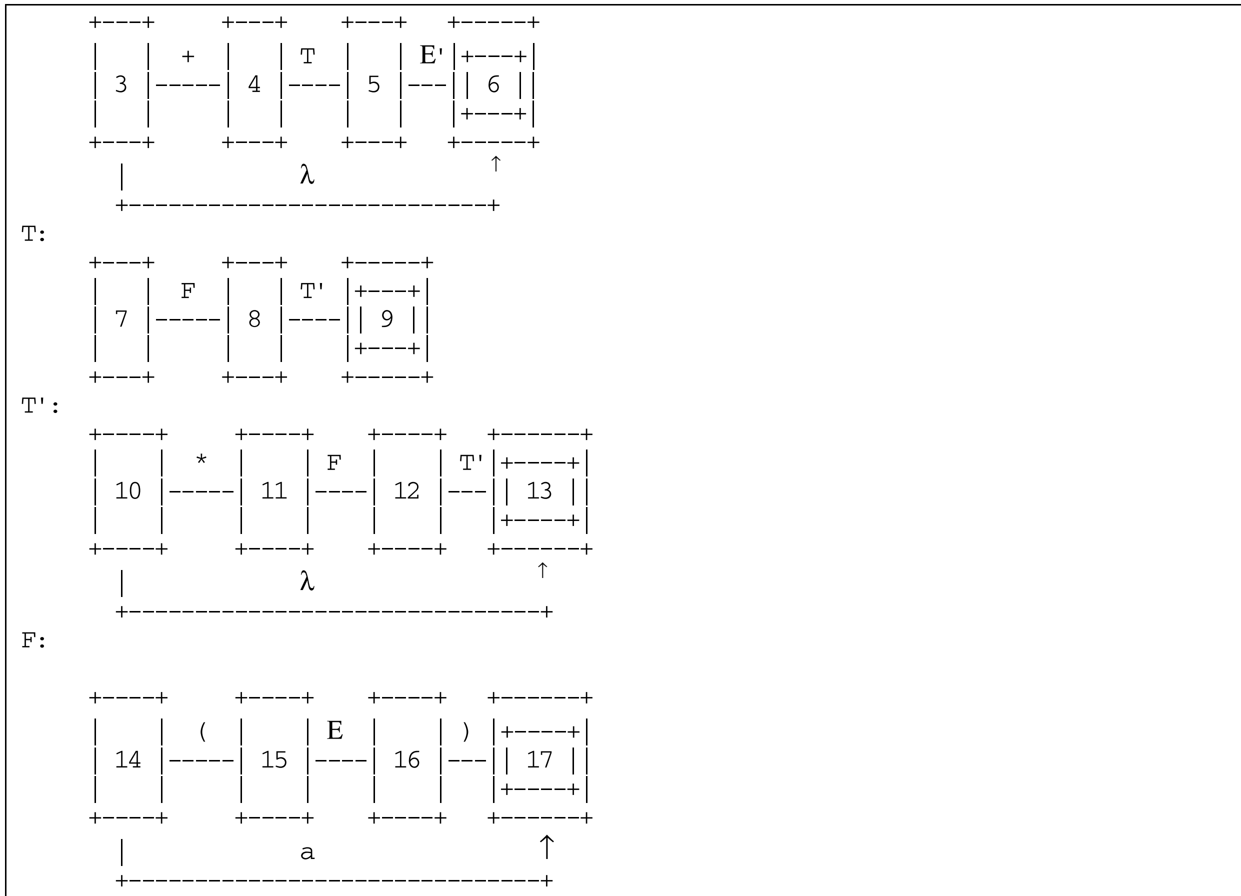
Analizorul predictiv care lucrează asupra diagramei de tranziții funcționează în modul următor. Starea inițială din care se începe analiza este starea corespunzătoare simbolului de start al gramaticii. Dacă la un moment dat analizorul se găseşte într-o stare s din care există un arc etichetat cu a spre o stare t şi simbolul curent de pe şirul de intrare este a ∈ T atunci analizorul se va deplasa pe şirul de intrare cu o poziție la dreapta şi va trece în starea t. Dacă din starea s există un arc etichetat cu un neterminal A spre starea t atunci analizorul trece în starea inițială corespunzătoare neterminalului A fără să avanseze pe şirul de intrare. Dacă se reuşeşte să se ajungă la starea finală pentru A se va trece în starea t (se observă că în acest caz se consideră că s-a "citit" A din şirul de intrare). Dacă din starea s există o λ-tranziție spre starea t atunci analizorul poate trece direct în stare t fără să se faca nici o deplasare pe şirul de intrare.

Ideea analizei sintactice predictive constă din identificarea a simbolului curent de pe banda de intrare cu începutul unei producții pentru simbolul neterminal curent. Acest tip de abordare funcționează dacă diagrama care reprezintă producțiile gramaticii este deterministă.

Să considerăm de exemplu gramatica pentru expresii aritmetice fără recursivitate stânga : E → TE', E' → +TE' | λ, T → FT', T' → *FT' | λ, F → (E) | a.

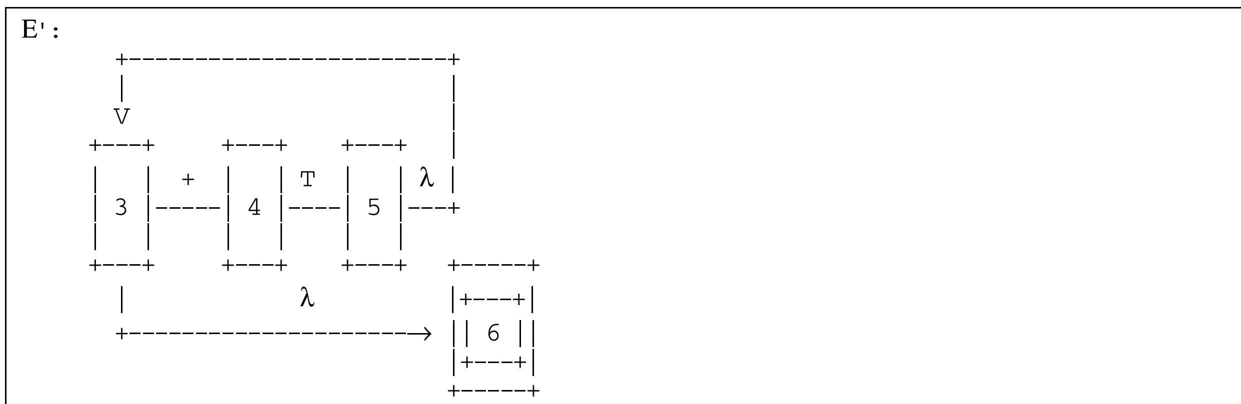
Diagramele de tranziție corespunzătoare sunt :

E':



Acest grup de diagrame de tranziție prezintă aparent dezavantajul existenței λ - tranzițiilor care pare să confere un caracter nedeterminist analizorului sintactic predictiv corespunzător acestor diagrame, dar acest aspect poate să fie rezolvat simplu. De exemplu pentru E' se va alege în implementare o λ-tranziție dacă şi numai dacă simbolul care urmează la intrare nu este +, etc.

Diagramele de tranziție obținute direct din gramatica pot să fie în continuare simplificate prin câteva transformări simple. De exemplu observând ca în diagrama de tranziție pentru E' apare un arc etichetat cu E' se pot face următoarele transformări :



şi apoi

```
E':
        +----------------+
        |                |
        V                |
      +---+    +---+     |
      |   | +  |   | T   |
      | 3 |----| 4 |-----+
      |   |    |   |
      +---+    +---+           +-----+
        |           λ          |+---+|
        +--------------------->|| 6 ||
                               |+---+|
                               +-----+
```

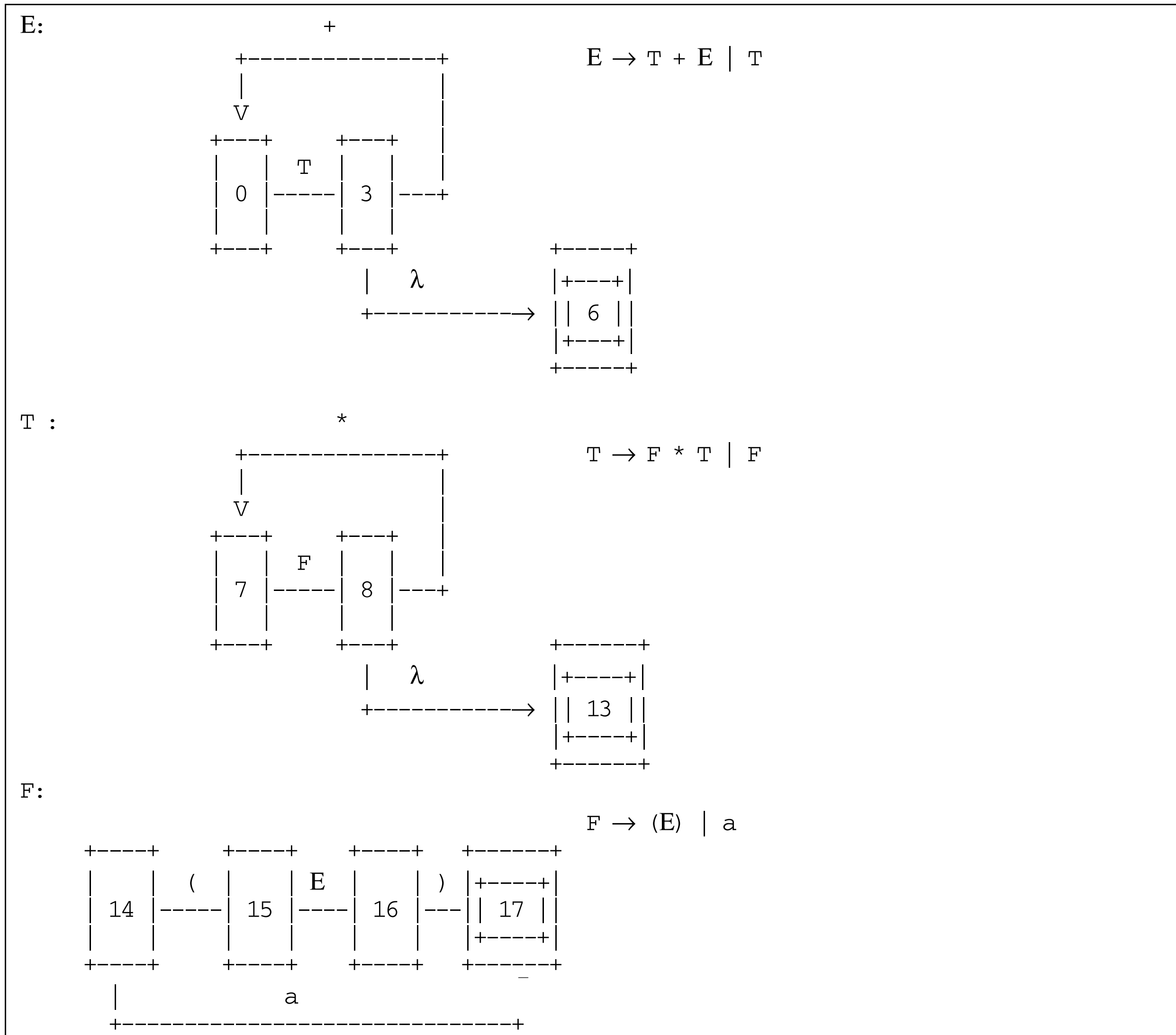
Din

```
E:
      +---+     +---+     +-----+
      |   | T   |   | E'  |+---+|
      | 0 |-----| 1 |-----|| 2 ||
      |   |     |   |     |+---+|
      +---+     +---+     +-----+
```

se obține :

```
E :                     T
                +----------------+
                |                |
                V                |
      +---+    +---+    +---+    |
      |   | T  |   | +  |   |    |
      | 0 |----| 3 |----| 4 |----+
      |   |    |   |    |   |
      +---+    +---+    +---+          +-----+
                 |           λ         |+---+|
                 +-------------------->|| 6 ||
                                       |+---+|
                                       +-----+
```

şi apoi pentru că stările 0 şi 4 sunt echivalente se obține :



Analizorul sintactic corespunzător acestor diagrame de tranziție este :

```
#include "stdio.h"
#include "ctype.h"

int caracter_curent;

/*
        prototipuri functii
*/

static void gasit(int t);
static void expr(void);
static void termen(void);
static void factor(void);


static void gasit(int t)
{
    if (caracter_curent == t)
     { putchar(caracter_curent);
       caracter_curent = getchar();
     }
   else
     { printf("\neroare \n");
       exit(1);
     }
}



/*
   E → TE', E' → +TE' | λ, E → E + T | T
*/

static void expr()
{
  termen();
  while (1)
    if (caracter_curent == '+')
      { gasit('+');
        termen();
      }
    else break;
}

/*
   T → FT', T' → *FT' | λ, T -> T * F | T
*/

static void termen()
{
  factor();
  while (1)
    if (caracter_curent == '*')
      { gasit('*');
        factor();
      }
    else break;
}

/*
```

```
  F → (E) | a
*/

static void factor()
{
  if (caracter_curent == 'a')
    gasit('a');
  else
    { gasit('(');
      expr();
      gasit(')');
    }
}


main()

{ caracter_curent = getchar();
  expr();
  putchar('\n');
}
```

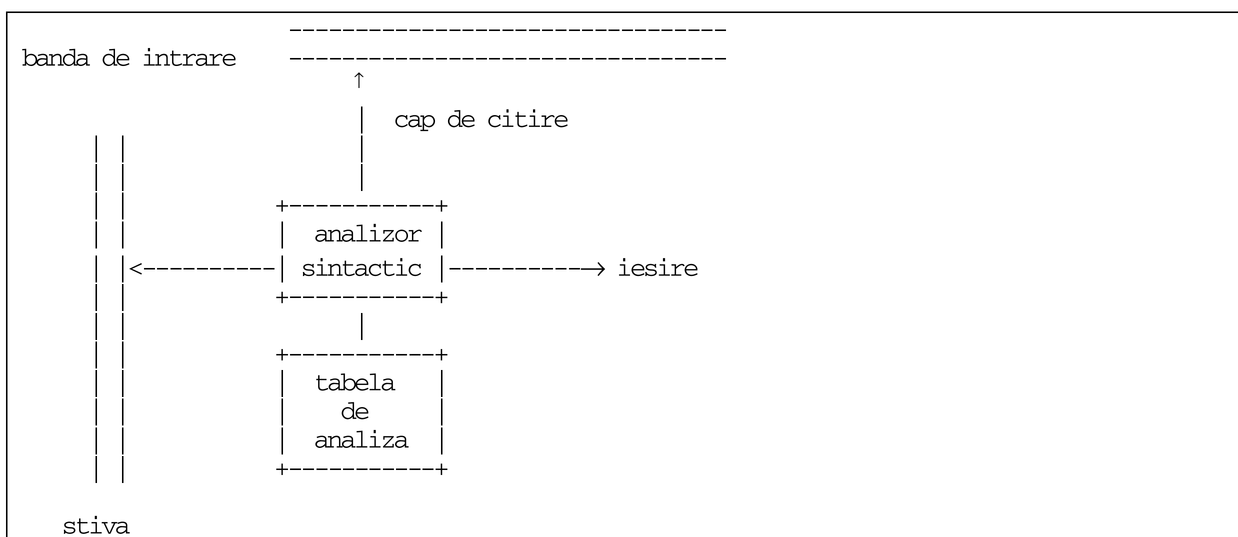
Programul afişează în ecou caracterele care sunt identificate drept corecte. În momentul în care se gaseşte un caracter care nu mai corespunde se afişează mesajul de eroare.

Se observă că abordarea anterioară presupune utilizarea unor proceduri recursive. Se poate însă implementa o abordare nerecursivă prin utilizarea unei stive. Adică, prin simularea explicită a unui automat cu stivă. Problema fundamentală în timpul analizei predictive constă din determinarea producției care va fi utilizată pentru neterminalul curent.

Dezavantajul analizei predictive constă din faptul că prin transformarea gramaticilor şi apoi prin transformarea diagramelor se poate pierde semnificația producților ceea ce poate complica până la imposibil procesul de generare de cod.

4.1.1.1 Gramatici LL(1)

În cazul analizei LL(1) modelul analizorului este :



Este vorba de fapt de un automat cu stivă pentru care funcţia de transfer **m** este memorată într-o tabelă de analiză. Pentru a favoriza construirea unui automat determinist se consideră că pe banda de intrare şirul analizat este urmat de un simbol special care nu aparţine gramaticii, pe care îl vom nota cu **$**. Existând un mecanism de identificare a sfârşitului şirului se extinde clasa imbajelor care pot să fie acceptate de către automat. Stiva conţine iniţial simbolii **S** şi **$** unde **S** este simbolul de start al gramaticii memorat în vârful stivei. În orice moment stiva conţine o secvenţă de simboli terminali şi neterminali. Tabela de analiza este o matrice **M[A,a]** unde **A** este un neterminal iar **a** este un terminal sau simbolul **$**. Programul care simulează funcţionarea analizorului sintactic funcţionează în modul următor. Fie **X** simbolul din vârful stivei şi **a** simbolul curent pe banda de intrare. Există trei posibilităţi :

1. dacă X = a = $ analiza s-a terminat cu succes;
2. dacă X = a ≠ $ analizorul extrage simbolul a din vârful stivei şi deplasează capul de citire pe banda de intrare cu o poziţie la dreapta;
3. dacă X este un neterminal, analizorul va cerceta valoarea M[X,a]. Valoarea din intrarea respectiva este fie o producţie pentru neterminalul X fie o informaţie de eroare. Dacă de exemplu M[X,a] = $\{X \rightarrow UVW\}$, simbolul X din vârful stivei va fi inlocuit cu UVW (U este noul vârf al stivei).

Ca algoritm funcţionarea analizorului poate să fie descrisă de următorul algoritm de simulare a unui automat push down determinist:

Intrare un şir w şi o tabela de analiză M pentru gramatica G.
Iesire dacă $w \in L(G)$, derivarea stângă, altfel un mesaj de eroare.

Iniţial configuraţia automatului este **w$** pe banda de intrare şi **S$** în stivă. Analizorul va afişa producţiile aplicate pe masură ce evoluează. Având în vedere că nu este necesară decât o singură stare pentru unitatea de control a automatului aceasta nu mai este considerată explicit.

```
initializeaza vârful stivei
intitializeaza pozitie curenta pe banda de intrare
repeta
  fie  X simbolul din vârful stivei si a  caracterul  curent de pe banda de
       intrare
  daca X este un simbol terminal sau $
     atunci
       daca X = a
         atunci
            descarca stiva
            actualizeaza pozitia curenta pe banda de intrare
      altfel
            eroare
            exit
       □
     altfel /* X este un simbol neterminal */
       daca M[X,a] contine X → Y1 Y2 ... Yk
         atunci
          inlocuieste X din varful stivei cu Y1 Y2 ... Yk
          afiseaza productia X → Y1 Y2 ... Yk
         altfel
          eroare
          exit
       □
  □
pana X = $ /* stiva este goala */
```

Să reluăm exemplul gramaticii expresiilor aritmetice. Tabela de analiză corespunzătoare acestei gramatici este :

neterminal	a	+	*	(	)	$
E	E -> TE'			E -> TE'		
E'		E'->+TE'			E' → λ	E' → λ
T	T -> FT'			T->FT'		
T'		T'→ λ	T'->*FT'		T' → λ	T' → λ
F	F → a			F->(E)		

Pentru şirul a + a * a rezultă următoarele mişcări (stiva este reprezentata cu vârful la stânga):

Stiva	Intrare	Iesire
E$	a + a * a $	
TE'$	a + a * a $	E → TE'
FT'E'$	a + a * a $	T → FT'
aT'E'$	a + a * a $	F → a
T'E'$	+ a * a $	
E'$	+ a * a $	T' → λ
+TE'$	+ a * a $	E' → +TE'
TE'$	a * a $	
FT'E'$	a * a $	T → FT'
aT'E'$	a * a $	F → a
T'E'$	* a $	
*FT'E'$	* a $	T'→ *FT'
FT'E'$	a $	
aT'E'$	a $	F → a
T'E'$	$	
E'$	$	T' → λ
$	$	∈' → λ

Construcţia tabelei de analiza este realizată cu ajutorul a doua funcţii **FIRST** şi **FOLLOW**.

FIRST : $(N \cup T)^* \rightarrow \mathcal{P}(T \cup \{\lambda\})$
FOLLOW : $N \rightarrow \mathcal{P}(T \cup \{\$\})$

Dacă w este un şir de simboli $w \in (N \cup T)^*$, FIRST(w) reprezintă setul terminalelor cu care încep şirurile derivate din w. Dacă $w \Rightarrow^* \lambda$ atunci $\lambda \in$ FIRST(w).

Pentru un neterminal A, FOLLOW(A) reprezintă mulţimea terminalelor a care pot să apară imediat dupa A într-o formă propoziţională; adica dacă există o derivare $S \Rightarrow^* \beta A a \gamma$ atunci $a \in$ FOLLOW(A).

Pentru a calcula FIRST(X) pentru toţi simbolii X din gramatică se aplică următoarele reguli până când nu se mai pot adăuga simboli terminali sau λ pentru nici o mulţime FIRST(X).

1. Dacă X este un terminal atunci FIRST(X) = {X};
2. Dacă există o producţie $X \rightarrow \lambda$ atunci FIRST(X) = FIRST(X) $\cup \{\lambda\}$

3. Dacă există o producţie $X \rightarrow Y1\ Y2\ ...\ Yk$ şi $a \in FIRST(Yi)$, $\lambda \in FIRST(Y1), ..., \lambda \in FIRST(Yi-1)$, ($Y1\ Y2\ ...\ Yi-1 \Rightarrow^* \lambda$) atunci $a \in FIRST(X)$. Dacă $\lambda \in FIRST(Yj)$, $1 \leq j \leq k$, atunci $\lambda \in FIRST(X)$. (orice simbol terminal din FIRST(Y1) este inclus în FIRST(X). Dacă $Y1 \Rightarrow^* \lambda$ atunci şi orice simbol terminal din FIRST(Y2) este inclus în FIRST(X), etc.).

Putem să calculăm funcţia FIRST pentru orice şir X1 X2 ... Xm în modul următor. Se adaugă la FIRST(X1 X2 ... Xm) toţi simbolii diferiti de λ din FIRST(X1). Se adaugă apoi toţi simbolii din FIRST(X2) diferiti de λ dacă $\lambda \in FIRST(X1)$, etc. În final dacă λ apare în FIRST(Xi), $1 \leq i \leq m$, se adauga λ la FIRST(X1 X2 ... Xm).

Pentru a calcula FOLLOW(A) pentru $A \in N$ se aplică următoarele reguli în mod repetat până când nimic nu se mai poate adauga la mulţimile FOLLOW:

1. $ face parte din FOLLOW(S) (S este simbolul de start al gramaticii);
2. Dacă există o producţie $A \rightarrow wB\beta$ atunci $FIRST(\beta) \setminus \{\lambda\} \subseteq FOLLOW(B)$;
3. Dacă există o producţie de forma $A \rightarrow w\ B$ sau o producţie $A \rightarrow wB\beta$ şi $\lambda \in FIRST(\beta)$ ($\beta \Rightarrow^* \lambda$) atunci $FOLLOW(A) \subseteq FOLLOW(B)$ (B poate să apară şi în alte contexte).

Să considerăm de exemplu din nou gramatica fără recursivitate stânga pentru expresii aritmetice:

```
E → TE', E' → +TE' | λ, T → FT', T' → *FT' | λ,
F → (E) | a.

FIRST(E) = FIRST(T) = FIRST(F) = {(,a}
FIRST(E') = {+,λ}
FIRST(T') = {*,λ}

FOLLOW(E) = FOLLOW(E') = {),$}
FOLLOW(T) = FOLLOW(T') = {+,),$}
FOLLOW(F) = {+,*,),$}
```

Pentru construirea tabelei de analiză se utilizează următoarele reguli:

- Dacă **A → w** este o producţie şi **a** ∈ FIRST(w) atunci dacă simbolul pe banda de intrare este **a** analizorul trebuie să înlocuiască în stivă pe **A** cu **w**.
- Dacă **w ⇒* λ** atunci A este înlocuit în stivă cu **w** numai dacă $a \in FOLLOW(A)$ (în particular simbolul a poate să fie $).

Rezultă următorul algoritm pentru construcţia tabelei :

```
initializare matrice M
pentru fiecare p = A → w ∈ P executa
   pentru fiecare a ∈ FIRST(w) ∩ T executa
          M[A,a] = M[A,a] ∪ {A → w}
   □


   daca λ ∈ FIRST(w)
    atunci
      pentru fiecare b ∈ FOLLOW(A) executa /* b ∈ T ∪ {$} */
              M[A,b] = M[A,b] ∪ {A → w}
          □
   □
□
```

Aplicând acest algoritm asupra gramaticii expresiilor aritmetice va rezulta tabela cu care a fost ilustrată funcționarea analizei predictive.

Să considerăm însă şi următoarea gramatică : S → i E t S S' | a, S' → e S | λ, E → b. (este vorba de gramatica care descrie instrucțiunea if). Pentru această gramatică rezultă următoarea tabelă de analiză:

neterminal	a	b	e	i	t	$
S	S -> a			S->iEtSS'		
S'			S' -> λ S' -> eS			S' -> λ
E		E -> b				

M[S',e] = {S' → eS, S' → λ} deoarece FOLLOW(S') = {e, $}. Se ştie că în această formă gramatica este ambiguă şi ambiguitatea se manifestă prin alegerea ce trebuie să fie făcută când se întâlneşte simbolul e (else). Soluția corectă este de a alege producția S' → eS (această alegere corespunde asocierii cuvintului else cu cel mai recent then). Se observă că alegerea permanentă a producției S' → λ ar impiedica aducerea terminalului e (else) în vârful stivei ceea ce nu poate conduce la o evoluție corectă.

O gramatică pentru care în tabela de analiză nu există intrări cu mai multe alternative se spune că este o gramatică LL(1) (primul L se refera la parcurgerea şirului de intrare de la stânga (Left) la dreapta, al doilea L pentru utilizarea derivării stânga (Left), iar 1 se referă la utilizarea unui terminal pentru a adopta o decizie de analiza).

Gramaticile LL(1) sunt gramatici neambigue, nerecursive stânga şi factorizate. Se poate arăta că o gramatică G este LL(1) dacă şi numai dacă pentru oricare doua producții de forma $A \to \alpha$, $A \to \beta$, cu $\alpha \neq \beta$ sunt satisfacute următoarele condiții :

1. $FIRST(\alpha) \cap FIRST(\beta) = \emptyset$
2. dacă $\beta \Rightarrow^* \lambda$ atunci $FIRST(\alpha) \cap FOLLOW(A) = \emptyset$ iar dacă $\alpha \Rightarrow^* \lambda$ atunci $FIRST(\beta) \cap FOLLOW(A) = \emptyset$.

Gramatica pentru expresii aritmetice (fără recursivitate stânga) este LL(1) în timp ce gramatica considerată pentru instructiunea if nu este LL(1).

Se pune problema cum se poate realiza o analiză predictivă pentru un limbaj descris de o gramatică care nu este LL(1). O solutie este de a transforma gramatica astfel încât să devină LL(1). În anumite cazuri (ca de xemplu în cazul anterior al gramaticii pentru limbajul corespunzător instrucţiunilor if) se poate face o "potrivire" a tabelei M, dar nu există o regulă generală de transformare a matricii de analiză astfel încât intrările multiple să poată să fie înlocuite cu intrări simple.

Transformarea suferită de o gramatică pentru a deveni LL(1) o poate face însă dificil de recunoscut. De obicei analiza predictivă este utilizată pentru instrucţiuni (nu pentru expresii).

4.1.1.2 Tratarea erorilor în analiza predictivă

Existenţa în stivă a terminalelor şi neterminalelor pe care analizorul se aşteaptă să le găsească pe banda de intrare simplifică mult problema diagnosticării erorii. O eroare este detectată în timpul analizei dacă în vârful stivei se găseşte un terminal care nu coincide cu terminalul curent de pe banda de intrare, sau dacă pentru neterminalul din vârful stivei (fie el A) şi terminalul curent de pe banda de intrare (fie el a) în tabela M nu există o producţie ($M[A,a] = \emptyset$).

Problema cea mai dificilă nu este însă identificarea erorii ci stabilirea modului în care analiza trebuie să continue după identificarea unei erori. Dacă o astfel de continuare nu ar fi posibilă atunci la prima eroare întâlnită compilatorul trebuie să abandoneze analiza. O metodă posibilă de continuare în caz de eroare constă din ignorarea de pe banda de intrare a simbolilor care urmează până la întâlnirea unui simbol dintr-o mulţime numită mulţime de sincronizare. Această mulţime se alege astfel încât analizorul să poată să decidă cât mai rapid modul de continuare pentru erorile cele mai frecvente. Se utilizează în acest scop câteva reguli cu caracter euristic ca de exemplu :

1. Pentru un neterminal A aflat în vârful stivei mulţimea de sincronizare va conţine simbolii din FOLLOW(A). Dacă la întâlnirea unui astfel de simbol pe banda de intrare se descarcă stiva, există o bună şansă pentru continuarea analizei, ca şi cum s-ar fi reuşit identificarea simbolului A;
2. Pentru limbaje de programare care au caracter de terminare a instrucţiunilor (ca de exemplu ; pentru limbajul C) acestea vor fi considerate în mulţimile de sincronizare pentru neterminalele care corespund instrucţiunilor. Pentru orice instrucţiune cuvintele cheie care nu fac parte din setul FOLLOW(instrucţiune) pot să fie introduse în mulţimea de sincronizare. În acest caz dacă de exemplu după o instrucţiune de atribuire lipseşte terminalul care indică terminarea acesteia în analiza se poate trece la cuvintul cheie care reprezintă începutul instrucţiunii următoare, etc. Pe de altă parte în general pentru o gramatică care descrie un limbaj de programare există o ierarhie a construcţilor sintactice. De exemplu expresiile apar în instrucţiuni care apar în blocuri care apar în funcţii, etc. Rezultă ca mulţimea de sincronizare corespunzătoare unei structuri sintactice va conţine simbolii corespunzători structurilor superioare. Astfel, se vor adăuga cuvintele cheie cu care încep instrucţiunile la mulţimea de sincronizare pentru neterminalele care generează expresii.
3. Dacă se adaugă simbolii din FIRST(A) la mulţimea de sincronizare pentru A, atunci se poate relua analiza corespunzătoare neterminalului A cu recunoaşterea unui terminal din FIRST(A) la intrare. Utilizând în caz de eroare dacă este posibil pentru neterminalul aflat în vârful stivei o λ-producţie se poate amâna diagnosticarea erorii.
4. Dacă pentru terminalul din vârful stivei nu există o corespondenţă în şirul de intrare se poate afişa un mesaj corespunzător renunţându-se în continuare la simbolul respectiv.

Utilizând funcţiile FIRST şi FOLLOW se poate completa tabela M cu informaţiile referitoare la mulţimea simbolilor de sincronizare. De exemplu pentru tabela de analiză corespunzătoare gramaticii expresiilor aritmetice rezultă:

	a	+	*	(	)	$	sinc
E	E -> TE'			E->TE'	sinc	sinc	) $
E'		E'->+TE'			E' → λ	E' → λ	) $
T	T -> FT'	sinc		T->FT'	sinc	sinc	+) $
T'		T'-> λ	T'->*FT'		T' -> λ	T' -> λ	+) $
F	F -> a	sinc	sinc	F->(E)	sinc	sinc	+*)$

Mulțimile sinc sunt construite pornind de la funcția FOLLOW. Cu ajutorul unei tabele de acest tip analiza se va executa în modul următor . Dacă M[A,a] nu conține nici o valoare atunci se avansează peste simbolul curent (a). Dacă M[A,a] = sinc atunci se descarcă vârful stivei pentru a continua analiza cu ce urmează după simbolul din vârful stivei. Dacă terminalul din vârful stivei nu se potriveşte cu terminalul de la intrare se descarcă terminalul din vârful stivei. De exemplu pentru secvență de intrare :)a * + a se obține :

Stiva	Intrare	Iesire
E$	) a * + a $	eroare,) ∉ FIRST(E)
E$	a * + a $	
TE'$	a * + a $	E → TE'
FT'E'$	a * + a $	T → FT'
aT'E'$	a * + a $	F → a
T'E'$	* + a $	
*FT'E'$	* + a $	T' → *FT'
FT'E'$	+ a $	eroare
T'E'$	+ a $	T' → λ
E'$	+ a $	E' → +TE'
+T'E'$	+ a $	
T'E'$	a $	T'→ λ
E'$	$	E' → λ
$	$	

Se observă că pentru început se caută un simbol din FIRST(E) pentru a începe recunoaşterea ignorindu-se astfel terminalul) şi nu se descarcă stiva. Pentru M[F,+] = sinc se va descărca stiva.

4.1.2 Analiza sintactica bottom-up

4.1.2.1 Analiza sintactica de tip deplaseaza şi reduce

Analiza sintactica de tip bottom-up încearcă să construiască un arbore de derivare pentru un şir de intrare dat pornind de la frunze spre rădăcina aplicând metoda "handle pruning". Adică trebuie să se construiască în stivă partea dreaptă a unei producţii. Selecţia producţiei construite se face pe baza "începuturilor" care reprezintă începuturi posibile de şiruri derivate conform producţiei respective.

4.1.2.2 Implementarea analizei sintactice bottom-up deplasează şi reduce

Pentru a utiliza această metodă trebuie să fie rezolvate două probleme :

1. localizarea începutului;
2. alegerea producţiei (dacă există mai multe producţii cu aceeaşi parte dreapta).

Algoritmul de analiză sintactică de tip deplasează şi reduce realizează simularea unui automat cu stivă. Din şirul de intrare se mută simboli terminali în stiva până când în vârful stivei se obţine un

început care este redus la partea stânga a uneia dintre producțiile corespunzătoare gramaticii. Dacă inițial în stivă se găseşte numai simbolul $ iar pe banda de intrare se găseşte şirul w$ în final dacă w aparține limbajului acceptat de analizor, în stivă se va găsi $S (cu S vârful stivei, S - simbolul de start al gramaticii) iar pe banda de intrare capul de citire este poziționat pe simbolul $. Să considerăm de exemplu acțiunile necesare pentru recunoaşterea şirului a + a * a pentru gramatica expresiilor aritmetice : $E \rightarrow E + E \mid E * E \mid a$.

stiva	intrare	actiune
$	a + a * a $	deplaseaza
$a	+ a * a $	reduce E → a
$∈	+ a * a $	deplaseaza
$E +	a * a $	deplaseaza
$E + a	* a $	reduce E → a
$E + E	* a $	deplaseaza
$E + E *	a $	deplaseaza
$E + E * a	$	reduce E → a
$E + E * E	$	reduce E → E * E
$E + E	$	reduce E → E + E
$E	$	succes

Au fost utilizate 4 tipuri de operații :

1. **deplasare** - presupune mutarea simbolului terminal curent de pe banda de intrare în vârful stivei;
2. **reduce** - în vârful stivei se găseşte un început care va fi înlocuit cu partea stânga a producției respective;
3. **succes** - s-a ajuns la sfârşitul şirului şi conținutul stivei este $S;
4. **eroare**.

Avantajul utilizării stivei constă din faptul că un început se formează întotdeauna în vârful stivei (şi deci dacă nu s-a format în vârful stivei un început atunci acesta nu trebuie să fie căutat în altă parte). Să considerăm două situații care pot să apară în cazul derivărilor dreapta :

1. $S \Rightarrow^* \alpha\ A\ z \Rightarrow^* \alpha\ \beta\ B\ y\ z \Rightarrow^* \alpha\ \beta\ \mu\ y\ z$
2. $S \Rightarrow^* \alpha\ B\ x\ A\ z \Rightarrow^* \alpha\ B\ x\ y\ z \Rightarrow^* \alpha\ \mu\ x\ y\ z$

cu $A \rightarrow \beta\ B\ \gamma$, $A \rightarrow y$, $B \rightarrow \mu$, $x,y,z \in T^*$, $\alpha,\beta,\mu \in (T \cup N)+$

În primul caz ($S \Rightarrow * \alpha\ A\ z$) reducerea inseamna :

```
stiva        intrare

$αβμ          yz$    se reduce μ
$αβB          yz$    se copiaza y
$αβBy          z$    se reduce βBy
$αA            z$    se copiaza z
$αAz            $    se reduce aAz
$S
```

În al doilea caz (S ⇒* a B x A z).

```
stiva        intrare

$αμ          xyz$   se reduce μ
$αB          xyz$   se copiaza x
$αBx          yz$   se copiaza y
$αBxy          z$   se reduce y
$αBxA          z$   se copiaza z
$αBxAz          $   se reduce aBxAz
$S
```

Se observă că întotdeauna începutul utilizat pentru reducere este în vârful stivei.

Setul prefixelor derivărilor dreapta ce pot să apara în vârful stivei în analiza unui analizor de tip deplasează reduce se numeşte setul prefixelor viabile pentru gramatica respectivă.

Există gramatici independente de context pentru care nu se poate utiliza o analiză de tip deplasează şi reduce, deoarece se poate ajunge fie în situaţii în care nu se ştie dacă trebuie să se efectueze o deplasare sau o reducere (conflict reducere / deplasare) respectiv nu se ştie care dintre variantele de reducere posibile trebuie să fie luată în considerare (conflict reduce / reduce). Să considerăm de exemplu din nou gramatica pentru instrucţiuni, în forma ambiguă:

```
instr → if expresie then instr |
        if expresie then instr else instr |
        alte_instr
```

Dacă se ajunge în configuraţia :

```
stiva                      intrare

...if expresie then instr  else ... $
```

nu putem să hotărâm dacă **if expresie then inst**r este un început indiferent de ce mai conţine stiva. Să observa ca în acest caz avem un conflict deplaseaza / reduce. Această gramatică şi în general orice gramatică ambiguă nu permite o analiză sintactică ascendentă deterministă. Pentru acest caz particular dacă hotarîm că în situaţia unui conflict deplaseaza / reduce are prioritate deplasarea atunci se va ajunge la o execuţie corectă a analizei. Să considerăm şi o gramatica care descrie instrucţiuni de atribuire şi apeluri de funcţii. În expresiile care apar în instrucţiunea de atribuire şi în lista de parametri actuali ai unei proceduri pot să apară variabile simple sau elemente de variabilele indexate pentru care se utilizează paranteze obişnuite. Să considerăm că analizorul lexical produce pentru atomul lexical identificator simbolul terminal **id**. Se obţine următoarea gramatică :

```
instrucţiune → id(lista_par) | expresie := expresie
lista_par → lista_par, id | id
expresie → id(lista_expr) | id
lista_expr → lista_expr, expresie | expresie
```

Să considerăm o instrucţiune care începe cu A(i,j) şi deci care este de forma id(id,id). Se observă că se ajunge în configuraţia:

```
   stiva              intrare

 id ( id              , id ) ...
```

Se observă că nu se ştie care este producţia cu care se face reducerea pentru că simbolul neterminal la care se face reducerea depinde de tipul variabilei A (variabila indexata sau funcţie). Rezultă ca a apărut un conflict reduce-reduce. Aceasta informaţie poate să fie obţinută din tabela de simboli. O soluţie constă din modificarea analizorului lexical care poate să furnizeze terminale diferite pentru variabile indexate şi nume de funcţii. În acest caz alegerea reducerii se va face în funcţie de tipul identificatorului care precede paranteza.

4.1.2.3 Analiza sintactica de tip LR(k)

LR(k) este o metodă de analiză ascendentă al cărui nume are următoarea semnificație : primul L indică sensul parcurgerii - stânga, dreapta, litera R indică faptul că derivarea este dreapta iar k indică numărul de atomi lexicali necesari pentru o alegere corecta a unui început stivă. Avantajele metodei LR(k) sunt :

- se pot construi analizoare de tip LR pentru toate construcțiile din limbajele de programare care pot să fie descrise de gramatici independente de context;
- clasa limbajelor care pot să fie analizate descendent în mod determinist este o submulțime proprie a limbajelor care pot să fie analizate prin tehnici LR;
- detectarea erorilor sintactice se face foarte aproape de atomul lexical în legătura cu care a aparut eroarea.

Dezavantajul major al acestei metode - volumul mare de calcul necesar pentru implementarea unui astfel de analizor fără aportul unui generator automat de analizoare sintactice.

Să considerăm din nou structura unui automat cu stivă.

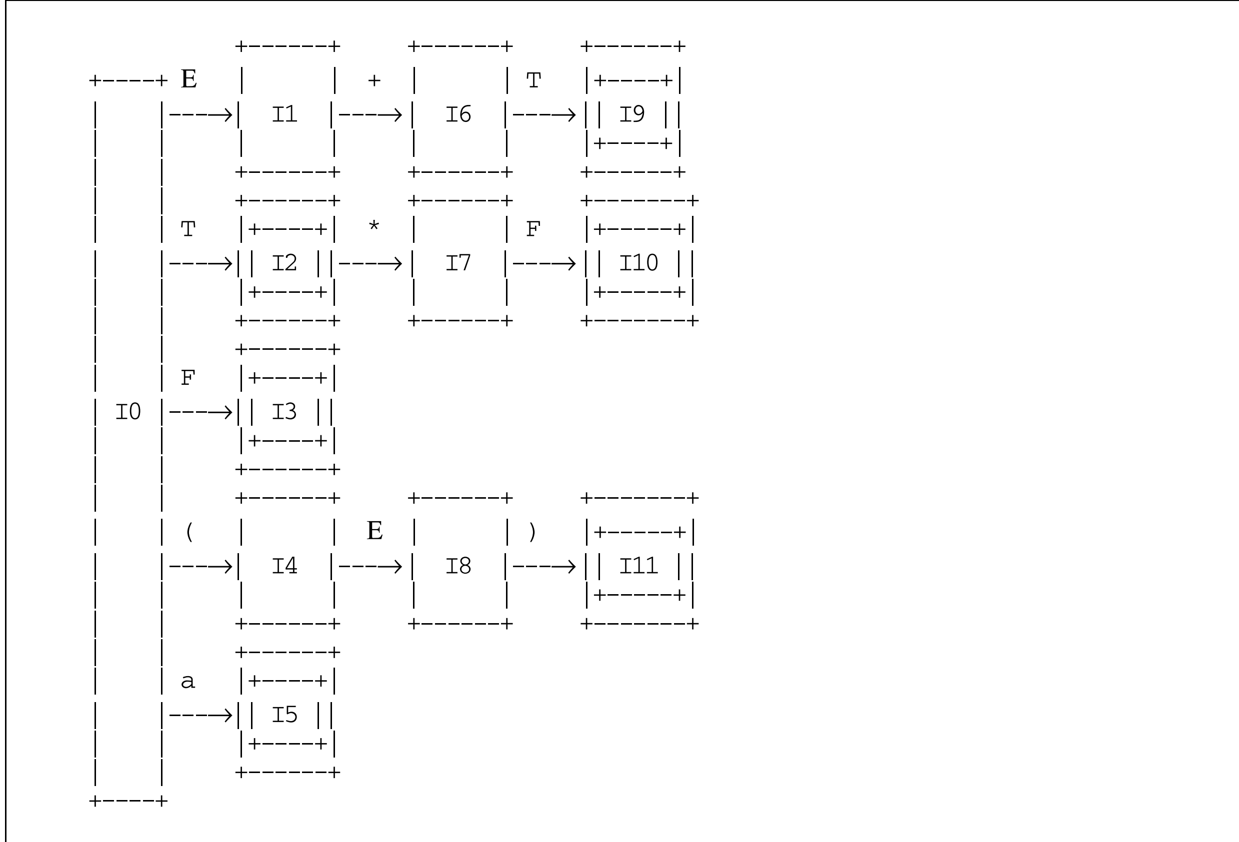
Ca şi în cazul analizei LL particularizarea limbajului se face prin intermediul tebelei de analiză care conține funcția m.

Înainte de a începe discuția referitoare la modul de construire a tabelelor de analiza este bine să reamintim câteva definiții. şi anume se numeşte început al unei forme propoziționale partea dreaptă a unei producții care dacă este înlocuită cu partea stânga produce forma propozițională anterioară în şirul formelor propoziționale obținute într-o derivare dreapta. Un prefix viabil este un subşir cu care poate începe o formă propozițională.

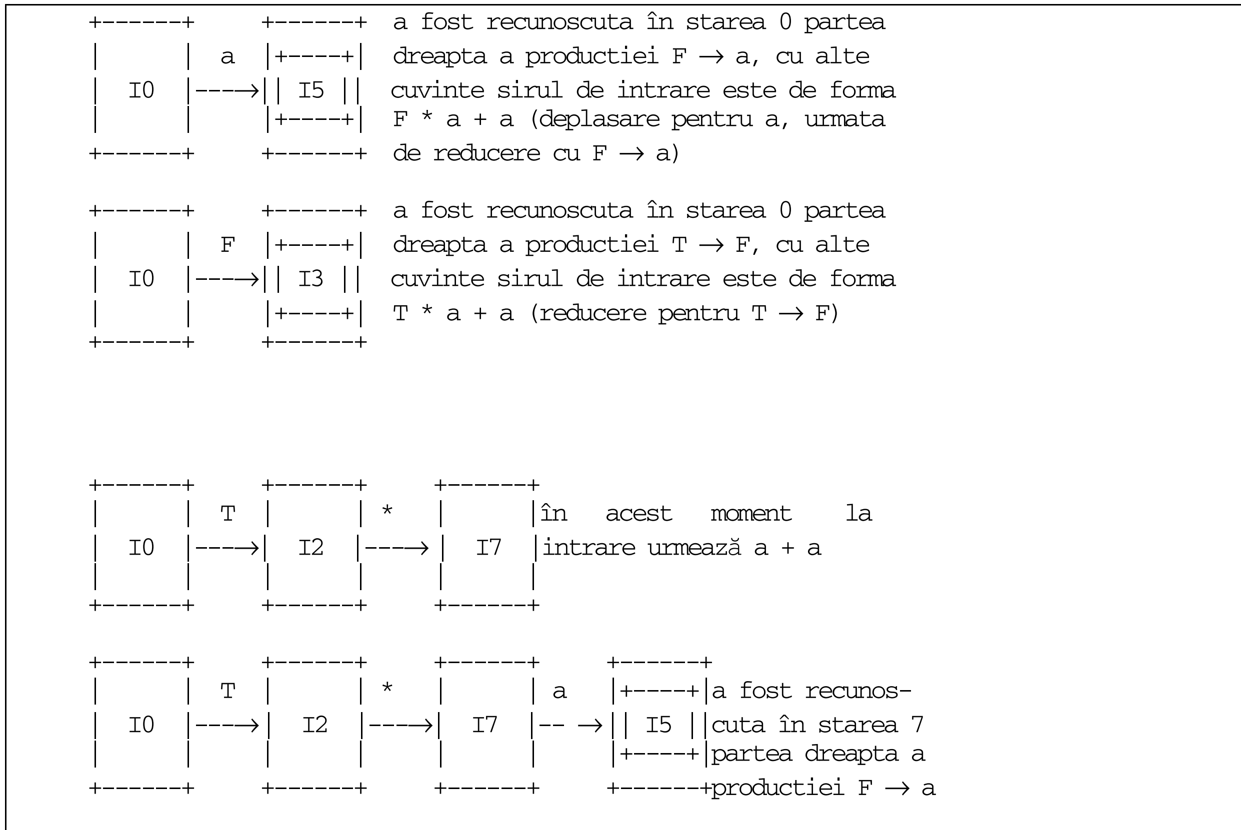
Problema centrală în execuția algoritmului constă din identificarea momentului în care în vârful stivei a fost obținut un început. Desigur, o variantă posibilă este explorarea completă a stivei la fiecare pas. Evident, o astfel de abordare este ineficientă. În cazul analizei de tip LR(k) se utilizează pentru identificarea capetelor un dispozitiv similar cu un automat finit. Rolul acestui automat este de a controla activitatea de recunoaştere a începuturilor. Să considerăm de exemplu din nou gramatica expresiilor aritmetice :

```
(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → a
```

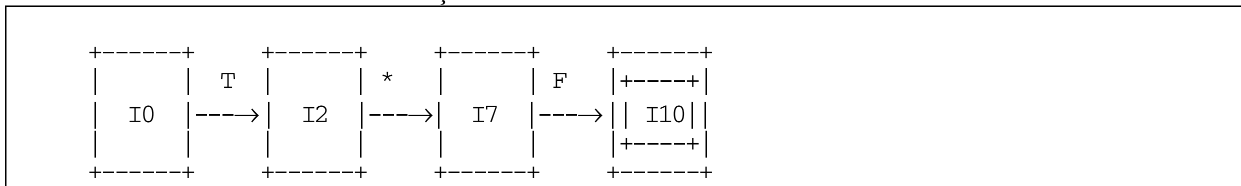
Putem să considerăm diagrama de tranziţii corespunzătoare acestor producţii:



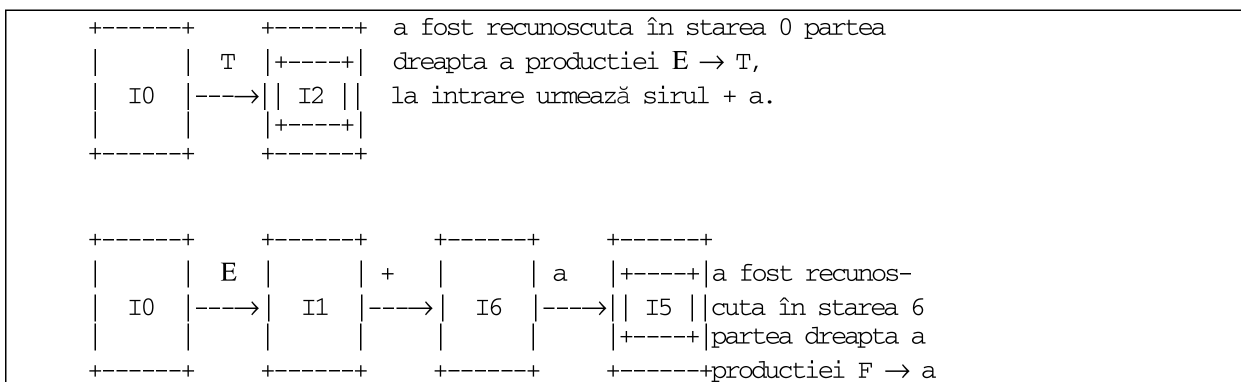
Să urmărim secvenţa de stări prin care se trece pentru şirul de intrare a * a + a.



Cu alte cuvinte în starea 7 urmează şirul de intrare F + a

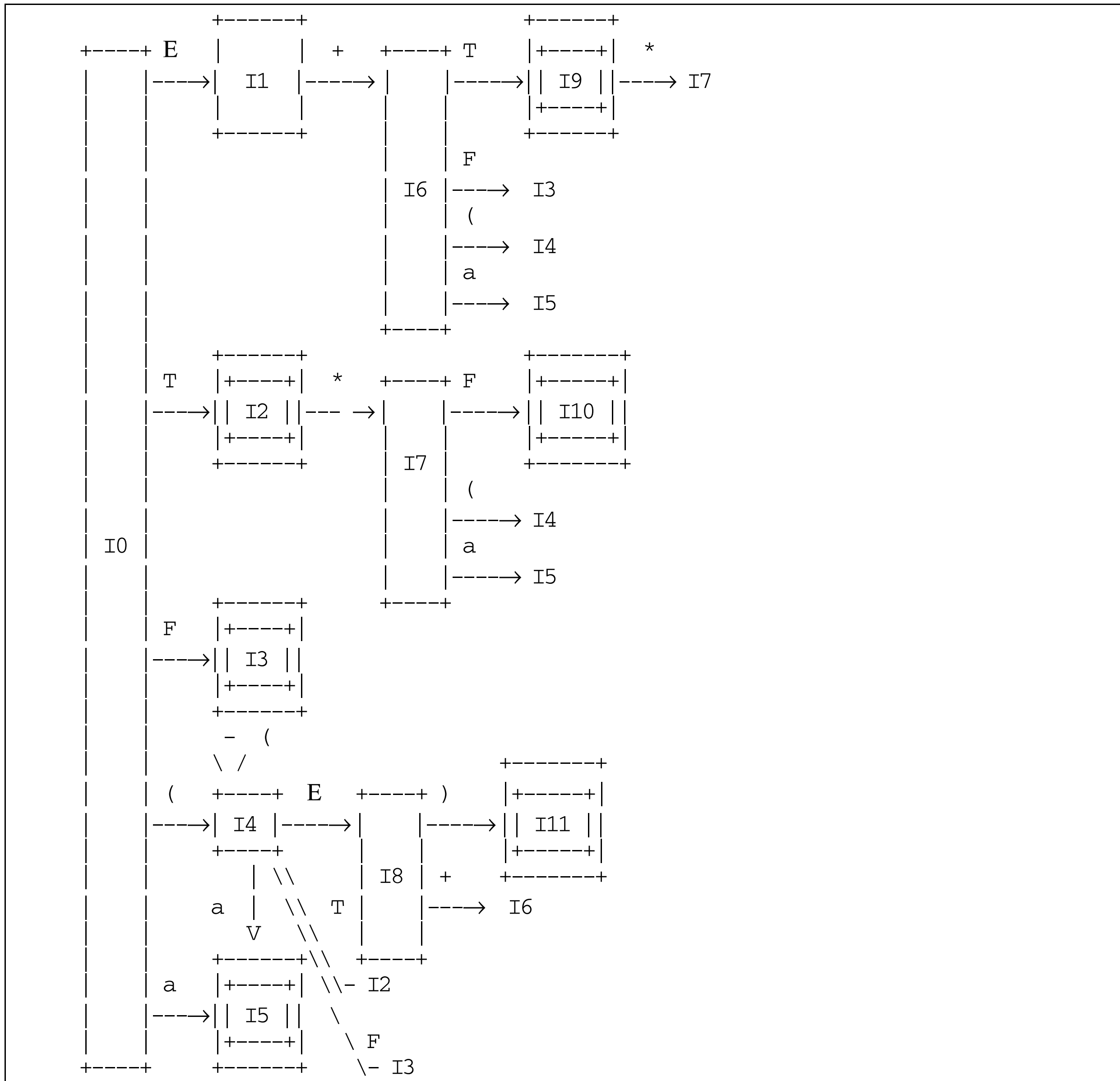


A fost recunoscută în starea 0 partea dreaptă a producţiei T → T * F (reducere).

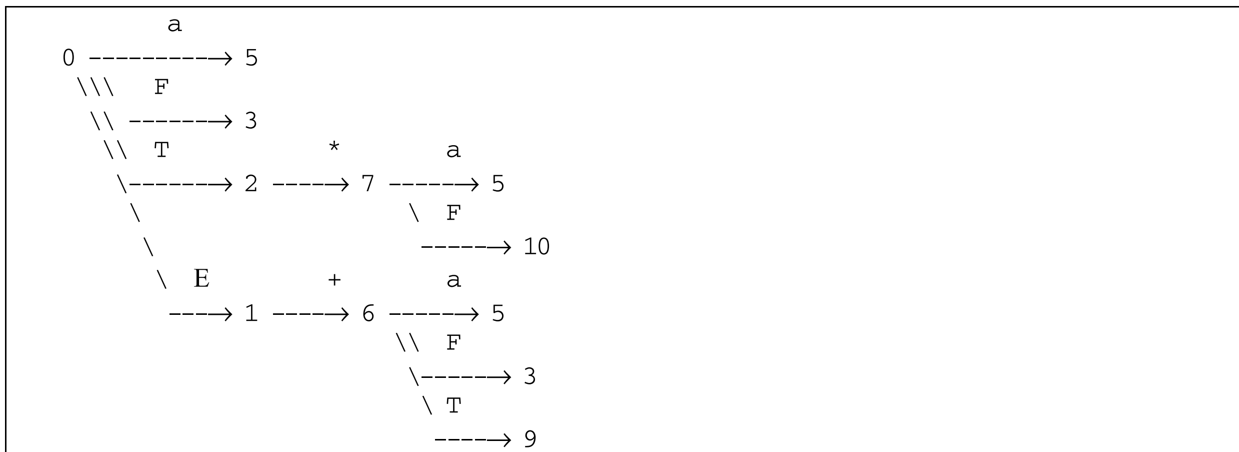


Evoluţia continuă până la obţinerea situaţiei :

Având în vedere aceasta evolutie diagrama de tranziții poate să fie reprezentată şi sub forma următoare (în care se evidențiază toate începuturile posibile) :



Se utilizează convenţia că la fiecare reducere se revine în starea anterioară lanţului de arce corespunzătoare începutului identificat executându-se o altă tranziţie. Desenul anterior poate să fie interpretat ca reprezentând un automat finit care acceptă începuturi. Să urmărim secvenţă de stări prin care se poate trece pentru şirul de intrare a * a + a :



Se observă că pentru primul simbol a se ajunge în starea 5 în care s-a recunoscut partea dreaptă a producției F $\rightarrow$ a, este ca şi cum s-ar fi aplicat la intrare şirul F * a + a, similar în acest caz se poate considera că s-a recunoscut partea dreaptă a producției T $\rightarrow$ F. După T se găseşte la intrare * apoi a, ajungându-se în starea 5 în care se recunoaşte din nou partea dreapta a producției F $\rightarrow$ a. În acest moment în starea 10 s-a citit la intrare şirul T * F deci se poate aplica pentru reducere producția T $\rightarrow$ T * F. Un analizor sintactic LR realizează simularea acestor recunoaşteri de începuturi memorând în stivă stările prin care se trece într-o recunoaştere ca cea făcută anterior.

Conținutul stivei la un moment dat este de forma s0 X1 s1 X2 s2 ... Xn sn, unde vârful stivei este sn (de fapt în stivă este suficient să se memoreze numai starile nu şi simboli Xi). Simbolii **Xi** reprezintă simboli din gramatica iar **si** reprezintă stări ale automatului care recunoaşte capete. Pe baza informației specificate de starea din vârful stivei şi a simbolului curent de pe banda de intrare se determină mişcarea următoare efectuată de către analizor.

Tabela de analiza este formata din doua parti : tabela de acțiuni şi tabela goto. Pentru o combinatie stare , simbol de intrare : sn, a, actiune[sn,a] poate avea următoarele valori :

- **deplasare în starea s** (se copiază terminalul curent de pe banda de intrare în stivă şi se trece în starea s).
- **reduce** pe baza productiei A $\rightarrow$ α, noua stare find determinată din tabela goto.
- **succes**
- **eroare**

Pentru o combinţtie (si, Xi), Xi $\in$ N, goto[si, Xi] este starea automatului de recunoaştere a începuturilor în care se ajunge dacă în starea **si** un început a fost redus la **Xi.**

Configurația unui analizor LR este dată de conținutul stivei şi cel al benzii de intrare. Deoarece pe tot parcursul analizei automatul cu stivă se găseşte în aceaşi stare, aceasta nu mai este prezentata explicit. Deci o configurație este de forma :

```
(s0 X1 s1 X2 ... Xm sm, ai ai+1 ... an$)
```

unde X1 X2 ... Xm ai ai+1 ... an reprezintă o formă propozițională obținuta dintr-o derivare dreapta dacă şirul aflat inițial pe banda de intrare face parte din limbaj. Următoarea mişcare a analizorului depinde de ai şi sm şi este descrisă de actiune[sm,ai]. Modificările posibile pentru configurația automatului pot să fie :

- dacă acțiune[sm,ai] = deplasare s, se obține noua configurație :

(s0 X1 s1 X2 ... Xm sm ai s, ai+1 ... an$)

- dacă acțiune[sm,ai] = reduce pe baza $A \rightarrow \alpha$, se obține configurația :

  (s0 X1 s1 X2 ... Xm-r sm-r A s , ai ai+1 ... an$)

  unde s = goto[sm-r,A] iar r este lungimea capatului α utilizat pentru reducere. Se observă ca s-au eliminat 2 * r simboli din stivă, starea nouă din vârful stivei fiind sm-r. Se introduce apoi în stiva neterminalul A (partea stânga a producției) şi starea obținuta pentru sm-r şi A din tabela goto (s = goto[sm-r, A]).

- dacă acțiune[sm,ai] = acc atunci analiza se încheie.

dacă acțiune[sm,ai] = eroare înseamnă că s-a diagnosticat o eroare.

Algoritmul general de analiza LR este :

Intrare un şir de intrare w şi o tabelă de analiză LR.

Iesire dacă $w \in L(G)$ este o propoziție derivata în G atunci se va obține lista producțiilor aplicate într-o derivare dreapta, altfel se obține un mesaj de eroare corespunzător.

```
pozitioneaza ip pe inceputul sirului w$ si s0 în stiva.
repeta
  fie  s  simbolul  din vârful stivei  si  a  simbolului  de
  intrare curent
  dacă actiune[s,a] = deplaseaza s'
     atunci
        memoreaza a si s' în stiva
        ip++
     altfel
        dacă actiune[s,a] = reduce A → α
          atunci
             extrage 2 * |α| simboli din stiva
             fie s' noul vârf al stivei
             memoreaza A în stiva
             memoreaza goto[s',A] în stiva
             afiseaza producția A → α
          altfel
            dacă actiune[s,a] = acc
               atunci return
               altfel eroare
            □
        □
  □
până false
```

Să considerăm de exemplu gramatica expresiilor artimetice :

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → a

Să considerăm că vom utiliza tabela de analiză:

	actiune						goto		
stare	a	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s6				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

şi - reprezintă deplasare si starea următoare i
rj - reprezintă reducerea cu producția j
acc - reprezintă succes
blanc - reprezintă eroare

Tabela goto[X,a] este conținuta în tabela actiune[X,a] pentru a ∈ T. Fie şirul de intrare a * a + a. Evoluția algoritmului de analiză este:

stiva	intrare	actiune
0	a * a + a $	deplasare
0 a 5	* a + a $	reduce cu F → a (6), goto[0,F]=3
0 F 3	* a + a $	reduce cu T → F (4), goto[0,T]=2
0 T 2	* a + a $	deplaseaza
0 T 2 * 7	a + a $	deplaseaza
0 T 2 * 7 a 5	+ a $	reduce cu F → a (6),goto[7,F]=10
0 T 2 * 7 F 10	+ a $	reduce cu T → T * F (3)
0 T 2	+ a $	reduce cu E → T (2), goto[0,E]=1
0 E 1	+ a $	deplaseaza
0 E 1 + 6	a $	deplaseaza
0 E 1 + 6 a 5	$	reduce cu F → a (6), goto[6,F]=3
0 E 1 + 6 F 3	$	reduce cu T → F (3), goto[6,T]=9
0 E 1 + 6 T 9	$	reduce cu E → E + T (0)
0 E 1	$	succes

Problema cea mai dificilă este desigur modul în care se construiesc tabelele de analiză. Se poate face următoarea observaţie - evoluţia analizorului urmăreşte recunoaşterea începutului din vârful stivei automatului. Dar o astfel de recunoaştere poate să fie realizată de către un automat finit puţin modificat. Simbolii de stare din stivă reprezintă de fapt stările unui astfel de automat. Informaţia din vârful stivei codifică de fapt tot ce se găseşte în stivă din punctul de vedere al începutului construit. Se observă că deosebirea fundamentală între analiza LL şi analiza LR este că pentru analiza LL(k) pe baza a k atomi lexicali trebuie "ghicita" producţia care se poate aplica pentru un simbol neterminal dat, pentru analiza LR(k) este cunoscută partea dreapta a productiei şi trebuie "ghicita" producţia pe baza următorilor k atomi lexicali care urmează.

În cele ce urmează prezentam câteva metode de construire a tabelelor de analiza pentru analiza LR(1).

4.1.2.3.1 Analiza SLR

Cea mai simplă metodă de construire a tabelelor de analiza LR este metoda SLR (simple LR). Un element LR(0) pentru o gramatică G este o producţie având un punct intercalat între

simbolii părţii dreapta ai producţiei. Astfel din producţia $A \rightarrow XYZ$ rezultă patru elemente : $[A \rightarrow .XYZ]$, $[A \rightarrow X.YZ]$, $[A \rightarrow XY.Z]$, $[A \rightarrow XYZ.]$. Ideea utilizării acestor elemente este că pentru a se recunoaşte începutul XYZ se vor aduce pe rând în vârful stivei prefixele X, XY, XYZ. Producţia $A \rightarrow \lambda$ produce un singur element $[A \rightarrow .]$. Un element poate să fie reprezentat de o

pereche de numere. Primul reprezintă numărul productiei, al doilea număr este poziţia punctului. Semnificaţia unui element este cât s-a recunoscut (văzut) din partea dreapta a unei producţii. Ideea

de baza este construirea unui automat finit determinist care să stie să recunoască elemente LR. Se grupează elementele în mulţimi care reprezintă stările unui automat finit nedeterminist care recunoaşte prefixe viabile. Gruparea acestor elemente se face pe baza unui algoritm de construcţie a subseturilor de tipul celor utilizate pentru construirea automatului finit determinist echivalent unui automat finit nedeterminist dat.

O colecţie a mulţimilor de elemente LR(0) numită mulţime canonică de elemente LR(0) reprezintă punctul de plecare pentru construcţia tabelelor de analiza. Pentru aceasta construcţie se utilizează o gramatică modificată G', doua funcţii : închidere şi goto şi o procedură numită elemente. Gramatica G' se obţine din G prin utilizarea unui alt simbol de start al gramaticii S' şi a unei producţii S' → S care permite recunoaşterea stării de acceptare prin utilizarea în reducere a acestei producţii.

Dacă I este o mulţime de elemente pentru o gramatica G, atunci inchidere(I) este o mulţime de elemente construite din I utilizând următoarea funcţie :

```
functia inchidere(I) este
  C = I
  repeta
     C' = C
     pentru fiecare A → a . B B ∈ C executa /* B ∈ N */
        fie B → α o productie pentru B
        C = C ∪ { [B → .α]}
     □
  pâna C = C'
  rezultat C
□
```

Dacă A → α . B β face parte din inchidere(I) la un moment dat în procesul de analiza, urmează să gasim la intrare un şir derivat din Bβ. Corespunzător dacă B → γ este o producţie, atunci în şirul de intrare ne aşteptăm să găsim un şir derivat din γ. De exemplu pentru gramatica modificată a expresiilor aritmetice :

```
E' → E
E  → E + T | T
T  → T * F | F
F  → (E) | a
```

pornind de la setul I = {[E' → .E]} se obţine inchidere(I) :

```
E' → .E, E → .E + T, E → .T, T → .T * F, T → .F
F → .(E), F → .a
```

În general pentru o mulţime T de elemente se poate face împărţirea acestora în doua clase :

1. elemente care formează nucleul, din care se pot genera alte elemente. În aceasta clasa intră elementul S' → .S şi toate celelalte elemente din I care nu au punct pe prima poziţie a părtii dreapta;
2. elemente care nu fac parte din nucleu şi care au punct pe prima poziţie a părţii dreapta (cu excepţia elementului S' →.S).

Se observă că orice element care nu face parte din nucleu poate să fie generat prin operația de închidere dintr-un element din nucleu. Corespunzător este suficienta memorarea numai a elementelor din nucleu (celelalte elemente putind să fie generate prin operatia de inchidere).

```
functia goto(I,X) este
  fie J = {[A → aX.B] | [A → a.XB] ∈ I}
  rezultat inchidere(J)
□
```

Funcția goto(I,X) unde I este o mulțime de elemente (deci o stare a automatului finit utilizat în recunoaşterea prefixelor viabile) şi $X \in N \cup T$, se defineşte ca fiind închiderea tuturor elementelor $[A \to a\ X.\ B]$ pentru care $[A \to a\ .\ X\ B]$ este în I. Intuitiv, dacă I este o mulțime de elemente utilizate pentru prefixe de forma t, atunci goto(I,X) este mulțimea de elemente utilizate pentru prefixul tX.

De exemplu dacă I = {[E' → E.], [E → E. + T]} atunci goto(I,+) va conține {[E → E + .T], [T → .T * F], [T → .F], [F → .(E)], [F → .a]}. Se calculează goto(I,+) examinând mulțimea I pentru elemente având simbolul + după punct. De exemplu elementul [E' → E.] nu este un astfel de element în schimb se observă că elementul [E → E. + T] satisface condiția. Se va muta punctul peste simbolul + şi se va obține elementul [E → E +. T] şi se face închiderea mulțimii care conține acest element.

Se poate construi mulțimea canonică a mulțimilor de elemente LR(0) pentru o gramatică G' modificată în modul următor :

```
procedura elemente
  C = {inchidere ({[S' → .S]})}
  repeta
     C' = C
     pentru fiecare multime I ∈ C si fiecare X ∈ N ∪ T executa
          daca  goto(I,X) ≠ ∅ & goto(I,X) ∉ C
            atunci C = C ∪ {goto (I,X)}
          □
     □
  pana cand C = C'
□
```

Pentru gramatica expresiilor aritmetice să considerăm din nou diagrama de tranziții pentru capetele care pot să apară în stivă:

```
+-----+ E   +-----+  +   +-----+ T    +-----+  *
|     |---->| I1  |----->|     |----->| I9  |---> I7
|     |     +-----+      |     |      +-----+
|     |                  |     | F
|     |                  | I6  |--->  I3
|     |                  |     | (
|     |                  |     |--->  I4
|     |                  |     | a
|     |                  |     |--->  I5
|     |                  +-----+
|     | T   +-----+  *   +-----+ F    +------+
|     |---->| I2  |----->|     |----->| I10  |
|     |     +-----+      |     |      +------+
|     |                  | I7  | (
|     |                  |     |-----> I4
| I0  |                  |     | a
|     |                  |     |-----> I5
|     |                  +-----+
|     | F   +-----+
|     |---->| I3  |
|     |     +-----+
|     |      -  (
|     |     \ /
|     | (   +-----+  E   +-----+ )    +------+
|     |---->| I4  |----->|     |----->| I11  |
|     |     +-----+      |     |      +------+
|     |        | \\  T   | I8  | +
|     |     a  |  \\     |     |--->  I6
|     |        V   \\    +-----+
|     | a   +-----+  \\- I2
|     |---->| I5  |   \
|     |     +-----+    \ F
+-----+                 \- I3
```

Dacă fiecare stare cu exceptia stării I0 este o stare finală se observă că această diagramă poate reprezenta automatul finit nedeterminist care acceptă prefixe viabile. Dacă considerăm automatul finit nedeterminist pentru care starile sunt elemente, se observă că va exista o tranziție de la nodul corespunzător elementului $[A \rightarrow a.X\ \beta]$ la nodul corespunzător elementului $A \rightarrow a\ X\ .\ \beta$. Aceasta tranziție (arc) va fi etichetată cu X. Există o tranziție de la nodul corespunzător elementului $[A \rightarrow a.\ B\ \beta]$ la nodul corespunzător elementului $[B \rightarrow .\gamma]$ dacă $B \rightarrow \gamma$ este o producție. Acest arc va fi etichetat cu λ. Operația de închidere este de fapt operația de construire a automatului finit determinist echivalent unui automat nedeterminist dat. În acest mod funcția goto(I,X) va indica tranziția din starea I pe baza simbolului X în automatul finit determinist. Rezultă următoarea colecție canonica pentru G :

```
I0 : E' → .E          I1 : E' → E.          I5 : F → a.
     E  → .E + T           E  → E. + T      I6 : E → E + .T
     E  → .T          I2:  E  → T.               T → .T * F
     T  → .T * F           T  → T. * F           T → .F
     T  → .F          I3 : T  → F.               F → .(E)
     F  → .(E)        I4 : F  → (.E)             F → .a
     F  → .a               E  → .E + T      I7 : T → T *. F
I9 : E  → E + T.           E  → .T               F → .(E)
     T  → T. * F           T  → .T * F           F → .a
I10: T  → T * F.           T  → .F          I8 : F → (E.)
I11: F  → (E).             F  → .(E)             E → E. + T
                           F  → .a
```

Spunem că elementul [A → β1 . β2] este valid pentru un prefix viabil a β1 dacă există o derivare dreapta S ⇒* a A w ⇒* a β1 β2 w. În general un element este valid pentru mai multe prefixe viabile. Dacă A → β1 . β2 este valid pentru a β1 şi β2 ≠ λ atunci nu s-a găsit încă un capat deci următoarea operaţie trebuie să fie o deplasare. Dacă β2 = λ atunci se poate aplica pentru reducere producţia A → β1. Problema este că pentru acelaşi prefix se pot găsi mai multe elemente valide, deci în general poate să apară un conflict care eventual poate să fie soluţionat pe baza următorului simbol de la intrare (LR(1)).

Se poate demonstra că mulţimea elementelor valide pentru un prefix viabil γ este mulţimea elementelor care sunt parcurse pornind din starea iniţială de a lungul unei căi etichetate cu γ în automatul finit determinist construit din mulţimea canonică de mulţimi de elemente cu tranziţii date de funcţia goto. De fapt aceste elemente valide conţin toate informaţiile necesare referitoare la conţinutul stivei.

Să considerăm din nou gramatica expresiilor aritmetice. Se observă că E + T * este un prefix viabil. Automatul se va gasi în starea I7 dupa parcurgerea acestui prefix. Starea I7 conţine elementele : [T → T *. F], [F →.(E)], [F → .a]. Să considerăm de exemplu urmatoarele derivari dreapta :

```
E' ⇒ E               E' ⇒ E              E' ⇒ E
   ⇒ E + T              ⇒ E + T             ⇒ E + T
   ⇒ E + T * F          ⇒ E + T * F         ⇒ E + T * F
   ⇒ E + T * a          ⇒ E + T * (E)       ⇒ E + T * a
   ⇒ E + T * F * a
```

Pentru prima derivare se va folosi elementul T → T *. F, pentru a doua F → .(E) iar pentru ultima F → .a pentru prefixul viabil E + T *.

Construirea tabelelor de analiza SLR se face pornind de la automatul finit care acceptă prefixe viabile. Algoritmul care va fi prezentat în continuare nu produce în mod necesar tabele cu intrări conţinând un singur termen pentru orice gramatică care descrie un limbaj de programare dar poate să fie utilizat cu succes pentru majoritatea construcţiilor din limbajelor de programare. Dându-se o gramatica G se utilizează gramatica G' (obţinută prin înlocuirea simbolului de start al gramaticii S cu un simbol S' şi adăugarea producţiei S' → S) pentru a construi C, mulţimea canonică de mulţimi de elemente pentru G'. Algoritmul care produce tabelele de actiune şi goto utilizează următorul algoritm :

Intrare O gramatica G'
Iesire Tabele de analiza pentru G'

Algoritmul parcurge urmatoarele etape :

```
1. se construieste C = {I0, I1, ..., In} multimea canonică a multimilor de elemente
   LR(0) pentru G'.
2. starea i este corespunzatoare multimii Ii. Intrarile în tabela actiune pentru starea i se
   obtin în modul urmator :

   a) daca [A → α . a β] ∈ Ii si goto(Ii,a) = Ij
        atunci actiune[i,a] = "deplaseaza j" /* a ∈ T */
      □
   b) daca [A → α.] ∈ Ii & A ≠ S'
       atunci
         pentru fiecare a ∈ FOLLOW(A) executa
            actiune[i,a] = "reduce A → a"
         □
      □
   c) daca [S' → S.] ∈ Ii
       atunci actiune [i,$] = "succes"
      □
```

Dacă în construirea tabelei actiune apar conflicte gramatica nu este SLR(1) şi trebuie să fie utilizata o alta metoda.

```
1. Intrarile în tabela goto pentru starea i se obzin în modul urmator :

     daca goto (Ii,A) = Ij     /* tranzitia din starea Ii în
                                  starea Ij se face pentru
                                  simbolul A */

       atunci goto[i,A] = j
     □

   4. Toate intrarile necompletate prin regulile 2 si 3 sunt marcate cu eroare
   5. Starea initiala a analizorului este cea care corespunde multimii de
      elemente care contine elementul [S' → .S].
```

Tabelele de analiză construite conform regulilor anterioare formeaza tabela SLR(1) pentru G. Un analizor LR care lucreaza cu tabele SLR(1) este un analizor SLR(1).

Să construim de exemplu analizorul SLR(1) pentru gramatica expresiilor aritmetice. Să considerăm întâi mulțimea I0 :

```
E' → .E, E → .E + T, E → .T, T → .T * F, T → .F,
F → .(E), F → .a.
```

Din elementul F → .(E) se obține actiune[0,(] = "deplasare 4", din F → .a se obține actiune[0,a] = "deplasare 5". Celelalte elemente nu produc acțiuni. Pentru I1: E' → E., E → E. + T se obține actiune[1,$] = "succes", actiune[1,+] = "deplasare 6". I2 este format din E → T., T → T. * F. FOLLOW(E) = {$,+,)}, rezultă actiune[2,$] = actiune[2,+] = actiune[2,)] = "reduce E → T". Pentru T → T. * F se obține actiune[2,*] = "deplasare 7". Conținând se obține tabela utilizată pentru a ilustra funcționarea algoritmului de analiza LR.

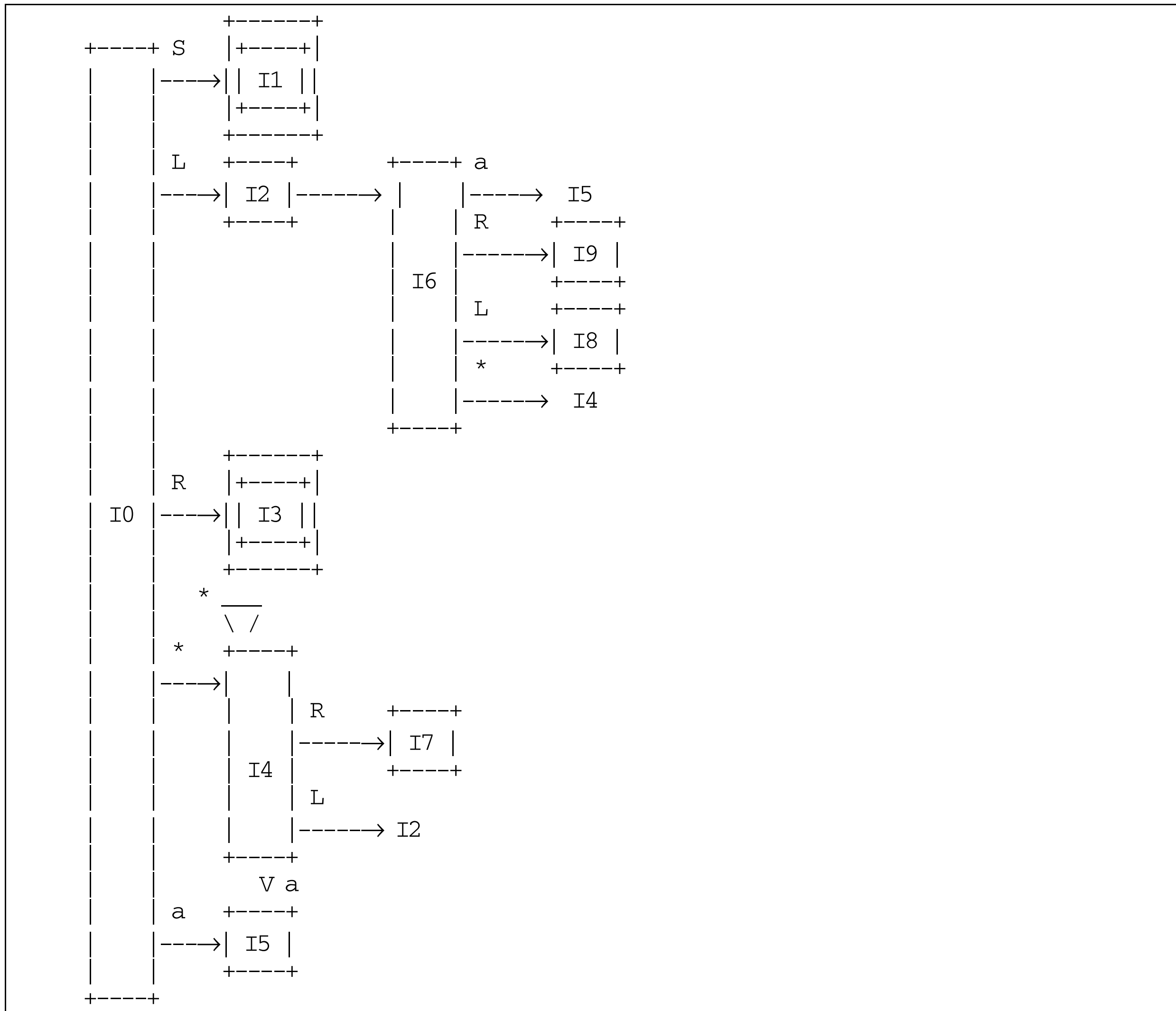
Orice gramatica SLR este neambigua. Reciproca nu este adevărată. Să considerăm de exemplu gramatica :

(1) S → L = R
(2) S → R
(3) L → *R
(4) L → a
(5) R → L

Mulțimea canonică de mulțimi de elemente este :

```
 I0 : S' → .S              I5: L → a.
      S → .L = R           I6: S → L = .R
      S → .R                  R → .L
      L → .* R                L → .*R
      L → .a                  L → .a
      R → .L               I7: L → *R.
I1:   S' → S.              I8: R → L.
I2:   S → L. = R           I9: S → L = R.
      R → L.
I3:   S → R.
I4:   L → *.R
      R → .L
      L → .*R
      L → .a
```

Diagramele de tranziție pentru prefixele viabile sunt :



Să considerăm elementul I2 pentru care se obține actiune[2,=] = "deplasare 6". FOLLOW(R) ⊆ FOLLOW(L) ⊆ FOLLOW(R) deci FOLLOW(R) = FOLLOW(L), = ∈ FOLLOW(L), deci = ∈ FOLLOW(R) şi deci actiune[2,=] = "reduce R → L. Se observa că deşi gramatica nu este ambiguă a apărut un conflict deoarece metoda folosita nu este suficient de puternica neputând să memoreze suficienta informație pe baza căreia pentru terminalul = să se stabileasca ce acțiune trebuie să se execute pentru un şir care poate să fie redus la L.

4.1.2.3.2 Analiza canonica LR

Prezentăm în continuare metoda cea mai generala de construire a tabelelor LR(1) pentru o gramatică dată. Pentru metoda considerata anterior în starea **i** se face reducere cu producția $A \to \alpha$ dacă mulțimea Ii conține elementul $[A \to \alpha.]$ şi a ∈ FOLLOW(A) urmează în şirul de intrare. Există situatii, în care dacă **i** apare în vârful stivei, prefixul $\beta\alpha$ din stiva este astfel încât βA nu poate să fie urmat de a într-o derivare dreapta şi deci nu se poate face reducerea $A \to \alpha$. Reluând ultimul exemplu pentru starea I2 şi elementul [R → L.], cu = pe banda de intrare, s-a obținut = ∈ FOLLOW(R) şi deci se pune problema aplicării reducerii R → L (vârful stivei conține L). Dar nu există nici o derivare dreapta care să înceapa cu R = ... Deci nu trebuie aplicată reducerea. Se observă că utilizarea

condiției a $\in$ FOLLOW(A) este prea slabă în cazul general. O soluție este transformarea gramaticii astfel încât pentru automatul care rezultă să fie bine precizat ce simbol de intrare poate să urmeze unui capăt pentru care există posibilitatea unei reduceri. Altă soluție constă din adăugarea unei informații pentru fiecare element obtinându-se elementele LR(1). Informația suplimentară este incorporată în stare prin redefinirea elementelor astfel încât fiecare element să conțină ca a doua componentă un simbol terminal. Forma generală pentru un element este: $[A \to \alpha \,.\, \beta, a]$ cu $A \to \alpha\beta$ o producție şi $a \in T \cup \{\$\}$. Un astfel de obiect se numeste element LR(1). 1 se referă la numărul de simboli terminali luați în considerare pentru a se decide e4fectuarea unei reduceri. Pentru un element de forma $[A \to \alpha\,.,\ a]$ se va face o reducere utilizând producția $A \to \alpha$ dacă şi numai dacă următorul simbol de intrare este a. Dacă un element este de forma $[A \to \alpha \,.\, \beta, a]$ şi $\beta \neq \lambda$ atunci a nu oferă o informație suplimentara. Pentru elementele $[A \to \alpha., a]$ este clar că $a \in$ FOLLOW(A) dar nu toate elementele din FOLLOW(A) apar în elementele LR(1) de forma $[A \to \alpha.,a]$. Se spune ca elementul LR(1) $[A \to \alpha \,.\, \beta,a]$ este valid pentru un prefix viabil γ dacă

există o derivare : $S \Rightarrow^* \sigma Aw \Rightarrow \sigma\alpha\beta w$, unde

1. $\gamma = \sigma\alpha$ şi
2. fie a este primul simbol din w, fie $w = \lambda$ şi $a = \$$.

Se considerăm de exemplu gramatica $S \to BB$, $B \to aB \mid b$. Fie derivarea $S \Rightarrow^* aaBab \Rightarrow aaaBab$. Elementul $[B \to a.B,a]$ este valid pentru prefixul viabil aaa considerând σ = aa, A = B w = ab, α = a, β = B în definiția anterioară. De asemenea să considerăm şi derivarea $S \Rightarrow^+ BaB \Rightarrow BaaB$. Pentru aceasta derivare elementul $[B \to a \,.\, B, \$]$ este valid pentru prefixul Baa.

Metoda de construire a mulțimii de elemente LR(1) valide este aceaşi cu cea pentru construirea mulțimii canonice de mulțimi de elemente LR(0) cu modificări adecvate ale procedurii inchidere şi goto.

Să considerăm un element de forma $[A \to \alpha \,.\, B\, \beta, a]$, A , B $\in$ N, $\alpha, \beta \in (N \cup T)^*$, $a \in T$. Acest element este valid pentru un prefix γ, dacă există o derivare dreapta $S \Rightarrow^* \sigma Aax \Rightarrow \sigma\alpha B\beta ax$ cu $\gamma = \sigma\,\alpha$. Să presupunem că $\beta\, \alpha\, x \Rightarrow^* by$, $b \in T$, $y \in T^*$. Atunci pentru orice producție $B \to \rho$, vom avea derivarile $S \Rightarrow^* \gamma Bby \Rightarrow \gamma\rho by$. Deci, $[B \to .\rho, b]$ este valid pentru $\gamma\rho$. Se poate observa că b este primul simbol terminal derivat din β, sau dacă $\beta \Rightarrow^* \lambda$, $\beta\, \alpha\, x \Rightarrow^* by$ şi deci b = a. Altfel spus $b \in$ FIRST(βax) (FIRST(βax) = FIRST(βa), $a \neq \lambda$). Rezultă urmatorul algoritm de construire a elementelor LR(1) :

Intrare O gramatica G'
Ieşire mulțimile de elemente LR(1) valide pentru unul sau mai multe prefixe din G'.

Algoritmul utilizează următoarele proceduri auxiliare :

```
functia inchidere(I) este
  repeta
     I' = I
     pentru fiecare element [A → α . B β, a] ∈ I,
            fiecare productie B → γ,
            fiecare b ∈ FIRST(βa) executa
               dacă [B → .γ,b] ∉ I
                 atunci
                    I = I ∪ {[B → .γ,b]}
               □
     □
  pana I = I'
  rezultat I
□

functie goto(I,X) este
  fie J = {[A → αX.β,a] | [A → α.Xβ,a] ∈ I}
  rezultat inchidere(J)
□
procedura elemente(G')
  C = {inchidere ( { [S' → .S,$]})}.
  repeta
     C' = C
     pentru fiecare I ∈ C, fiecare X ∈ N ∪ T executa
       daca goto(I,X) ≠ ∅ si goto(I,X) ∉ C
          atunci C = C ∪ {goto(I,X)}
       □
     □
  pana C' = C
```

Să considerăm de exemplu gramatica $S' \rightarrow S$, $S \rightarrow CC$, $C \rightarrow cC \mid d$. Limbajul generat este $c^n dc^m d$ Să calculăm întâi mulțimea ({[S' →.S, $]}. Să identificam în elementul [S' → .S,$] componentele elementului generic [A → α.Bβ,a]. Rezultă $A = S'$, $a = \lambda$, $B = S$, $\beta = \lambda$, $a = \$$. Din modul de acțiune al funcției inchidere se observă că se adaugă termeni de forma [B → .τ, b] pentru fiecare producție de forma $B \rightarrow \tau$ şi fiecare $b \in FIRST(\beta a)$. Singura producție care se potriveste este $S \rightarrow CC$, pentru care $\beta a = \$$, pe post de b se obține $. Rezultă adăugarea elementului [S → .CC, $] la mulțimea inchidere ([{[S' → .S,$]}. Să considerăm acum producțiile care au C în partea stângă pentru care trebuie să se adauge elementele [C →.γ,b] pentru $b \in FIRST(C\$)$. Comparând din nou elementul [S →.CC,$] cu [A → α.Bβ,a], de data aceasta $A = S$, $\alpha = \lambda$, $B = C$, $\beta = C$, $a = \$$. Deoarece $C \Rightarrow^* \lambda$, $FIRST(C\$) = FIRST(C) = \{c, d\}$. Se vor adauga deci elementele [C →.cC, c], [C →.cC,d], [C → .d,c], [C → .d, d]. Se observă că nu mai există elemente cu un neterminal la dreapta punctului deci s-a obținut :

```
I0 : S' → .S,$
     S  → .CC,$
     C  → .cC,c/d
     C  → .d, c/d
```

Să calculăm goto(I0,X) pentru diferitele valori posibile pentru X. Dacă X = S se va calcula inchiderea elementului [S' → S., $]. Rezultă :

```
I1 : S' → S.,$
```

Dacă X = C vom calcula închiderea pentru S → C . C, $:

```
I2 : S → C.C,$
     C → .cC,$
     C → .d,$
```

Dacă X = c, se calculează închiderea pentru {[C → c.C, c/d]}.

```
 I3 : C → c.C, c/d
      C → .cC, c/d
      C → .d,c/d
```

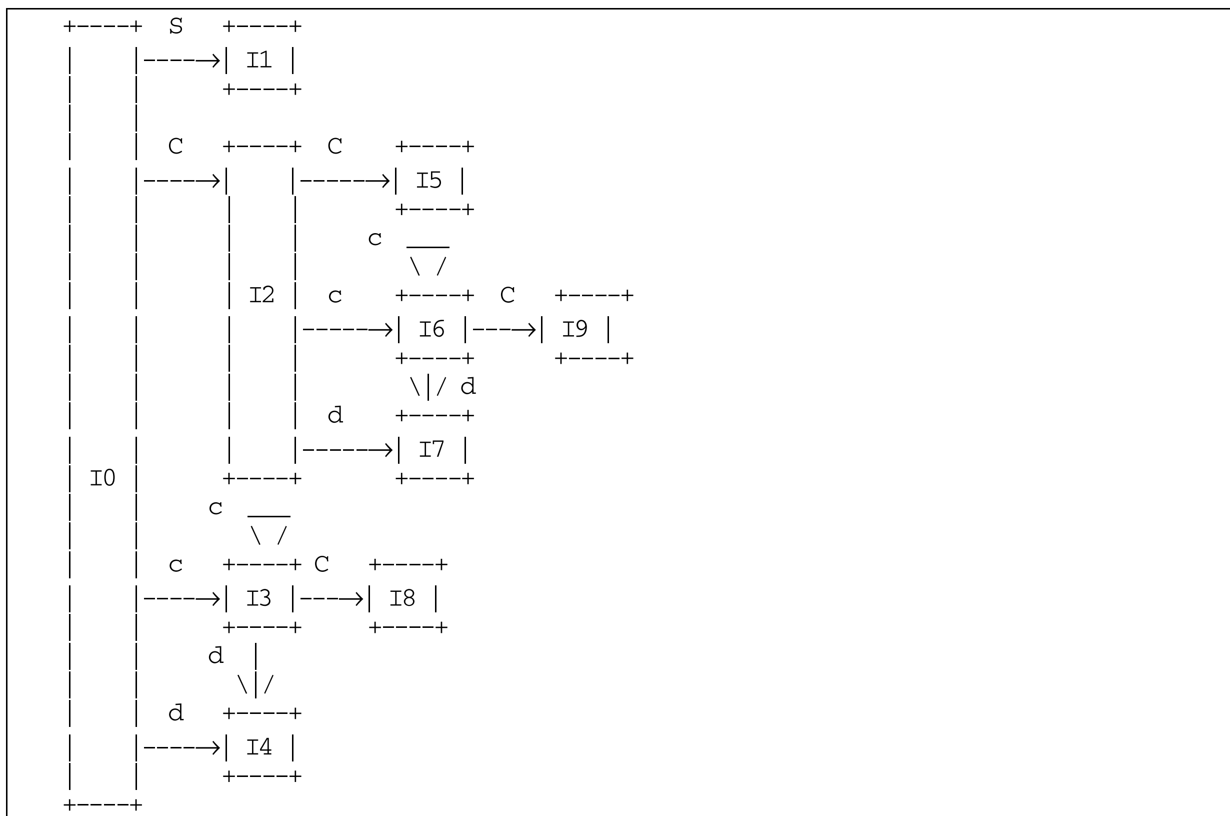
Pentru X = d

```
I4 : C → d., c/d
```

În acest mod s-a terminat calculul pentru inchidere (I0), urmează:

```
I5: S → CC.,$           I7: C → d.,$
I6: C → c.C,$           I8: C → cC., c/d
    C → .cC,$           I9: C → cC.,$
    C → .d, $
```

Rezultă următorul graf al automatului :



Pe baza descrierii obţinute pentru automatul finit, descriere ce s-a construit utilizând funcţiile goto şi inchidere, rezultă urmatorul algoritm de construire a tabelelor de analiză LR pentru G'.

Intrare o gramatica G'
Iesire Tabelele de analiza actiune şi goto pentru G'

Algoritmul parcurge urmatoarele etape :

```
1. se construieste C = {I0, .., In} multimea de multimi  canonice de elemente LR(1) pentru
G'.
2. pentru fiecare stare i reprezentata prin multimea Ii executa
      dacă [A → α.a β, b] ∈ Ii si goto(Ii,a) = Ij  /* a ∈ T */
        atunci actiune[i,a] = "deplaseaza j"
      □
      dacă [A → a., a] ∈ Ii, A ≠ S'
        atunci actiune[i,a] = "reduce A → a"
      □
      daca [S' → S., $] ∈ Ii
        atunci actiune[i,$] = "succes"
      □
  □
```

Dacă din cele regulile anterioare rezultă un conflict atunci înseamnă că gramatica nu este LR(1) şi execuţia algoritmului se opreşte.

```
3. pentru fiecare Ii ∈ C si fiecare A ∈ N executa
     daca goto(Ii,A) = Ij
       atunci goto [i,A] = j
     □
   □
4. Toate intrarile necompletate corespund unor situatii de eroare
5. Starea  initiala  a analizorului se obtine din  multimea  care contine elementul [S' →
.S,$].
```

Aplicând acest algoritm pentru gramatica care are producțiile :

(1) S → CC
(2) C → cC
(3) C → d

se obțin tabelele de analiza :

stare	actiune			goto	
	c	d	$	S	C
0	s3	s4		1	2
1	succes				
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

O gramatică SLR(1) este evident o gramatica LR(0), dar tabela de analiză LR(1) pentru aceaşi gramatică are mai multe stări decât tabela de analiza SLR.

Să reluăm şi exemplul care "nu a funcționat" corect pentru algoritmul SLR:

(1) S → L = R
(2) S → R
(3) L → *R
(4) L → a
(5) R → L

Mulțimea canonică de mulțimi de elemente este :

```
 I0 : S' → .S, $              I5: L → a., =/$
      S  → .L = R, $          I6: S → L = .R, $
      S → .R, $                   R → .L, $
      L → .* R, =/$               L → .*R, $
      L → .a, =/$                 L → .a, $
      R → .L, $               I7: L → *R., =/$
I1:   S' → S., $              I8: R → L., =/$
I2:   S → L. = R , $          I9: S  → L = R. , $
      R → L., $               I10: R → L., $
I3:   S → R., $               I11: L → *.R, $
I4:   L → *.R, =/$                 R → .L, $
      R → .L, =/$                  L → .*R, $
      L → .*R, =/$                 L → .a, $
      L → .a, =/$            I12:L → a., $
```

Se observă că pentru I2 se va face reducere dacă urmează sfârşitul şirului respectiv se va face deplasare dacă urmează =.

4.1.2.3.3 Analiza sintactică LALR

Numele LALR a fost ales pornind de la termenul "lookahead" - LR. Tabelele construite utilizând aceasta metoda sunt mai mici (există mai puţine stări) decât tabelele canonice LR şi de asemenea majoritatea construcţiilor din limbajele de programare pot să fie exprimate convenabil utilizând gramatici LALR.

Pentru un limbaj cum este PASCAL-ul tabelele de analiza LALR şi SLR au sute de intrari în timp ce tabelele canonice au câteva mii. Să reluăm din nou gramatica G' : S' → S, S → CC, C → cC | d şi să analizăm mulţimile canonice I4 şi I7 :

```
I4 : C → d., c/d , I7 : C → d., $
```

Pentru I4 decizia de reducere este pentru simbolii terminali c şi d, pentru I7 decizia de reducere se face pentru $. Gramatica pe care am considerat-o generează şiruri de forma c*dc*d. Deci analizorul va deplasa iniţial şirul de c şi va intra în starea I4 după întâlnirea simbolului d, după care se va face o reducere C → d dacă se întâlneşte un c sau un d. Reducerea este corectă deoarece c sau d pot să înceapă şirul c*d care trebuie să urmeze. Dacă însă apare $ înseamnă că şirul s-a terminat prea devreme (de exemplu este de forma ccd). După d se intră în starea 7 după care trebuie să urmeze $ sau şirul de intrare nu este corect. Să considerăm ca reunim cele doua stări sub forma unui element [C → d., c/d/$]. Trimiterile în I4 pentru d din I0, I2, I3 şi I6 vor fi spre I47. Actiunea din I47 va fi de reducere pentru orice intrare. Deci există situaţii în care în loc de semnalarea unei erori se va face reducerea C→ d. De exemplu pentru şirurile ccd sau cdcdc. Eroarea se va depista înainte de a se mai considera un alt caracter de intrare.

În general se cauta mulţimi de elemente LR(1) având acelaşi nucleu (aceeaşi mulţime a primelor componente pentru elemente) şi se face o reuniunea acestor mulţimi de elemente. Pentru exemplul considerat mulţimile I4 şi I7 au ca nucleu mulţimea {C → d.}, de asemenea mulţimile I3 şi I6 au ca nucleu mulţimea {C → c.C, C → .cC, C →.d}. În general un nucleu este o mulţime de elemente LR(0). O gramatica LR(1) poate produce mai multe mulţimi cu acelaşi nucleu.

Se observă că miezul mulţimii goto(I,X) depinde numai de miezul mulţimii I. Corespunzător se poate face reuniunea mulţimilor goto pentru doua mulţimi care au acelaşi nucleu. Valorile din tabela goto şi actiune vor reflecta aceste reuniuni prin modificări corespunzătoare. Adică tabela goto va

reflecta noile mulțimi între care se fac tranziții iar în tabela actiune se înlocuiesc o serie de intrări de eroare cu intrări de reducere.

Să presupunem ca s-a construit o gramatica LR(1) fără conflicte. Dacă înlocuim toate mulțimile cu acelaşi nucleu cu reuniunea lor s-ar putea, dar este puțin probabil să apară conflicte. Să presupunem ca există un conflict datorită unui simbol $a \in T$ deoarece există un element $[A \to \alpha., a]$ care indică o reducere cu $A \to \alpha$ şi există un alt element $[B \to \beta. \alpha\gamma, b]$ care indică o deplasare. Se observă că în acest caz dintre elementele pentru care s-a facut reuniunea există şi elementele $[A \to \alpha., a]$ şi $[B \to \beta.\alpha\gamma, b]$ pentru un anumit b. Inseamnă că aceste elemente prezintă acelaşi conflict deplaseaza / reduce pentru terminalul a şi gramatica nu ar mai fi LR(1) asa cum s-a presupus. Deci un conflict deplaseaza / reduce nu poate să apară prin reuniunea mulțimilor de elemente cu acelaşi nucleu, dacă un astfel de conflict nu a apărut într-una din stările inițiale deoarece deplasările depind numai de nucleu, nu şi de simbolii care urmează pe banda de intrare. Dar poate să apară un conflict reduce / reduce. Să considerăm de exemplu gramatica : $S' \to S$, $S \to aAd \mid bBd \mid aBe \mid bAe$, $A \to c$, $B \to c$. Această gramatică generează şirurile : acd, ace, bcd, bce. Construind mulțimea elementelor LR(1) se va obține mulțimea $\{ [A \to c.,d], [B \to c., e]\}$ ale carui elemente sunt valide pentru prefixul ac şi mulțimea $\{[A \to c., e], B \to c.,d]\}$ ale cărui elemente sunt valide pentru prefixul bc. Dacă se construieşte reuniunea acestor mulțimi se obține mulțimea $\{[A \to c., d/e], [B \to c., d/e]$ pentru care se observă că există un conflict reduce/reduce.

Acest conflict semnifică faptul că gramatica nu este LALR(1). De obicei transformări asupra gramaticii rezolva problema.

Bibliografie

- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers Principles, Techniques, and Tools, Addison Wesley, 1986
- Alfred V. Aho, Jeffrey D. Ullman, The Theory of Parsing, Translation and Compiling, Prentice-Hall
- William M. Waite, Gerhard Goos, Compiler Construction, Springer Verlang
- Harry R. Lewis, Christos H. Papadimitriou, Elements of the Theory of Computation, Prentice Hall, 1981
- Michael A. Harrison, Introduction to Formal Language Theory