UNIVERSITATEA „ALEXANDRU IOAN CUZA”
FACULTATEA DE INFORMATICĂ
DEPARTAMENTUL DE ÎNVĂȚĂMÂNT LA DISTANȚĂ

Gheorghe Grigoraş

PROIECTAREA COMPILATOARELOR

2006 – 2007

EDITURA UNIVERSITĂȚII „ALEXANDRU IOAN CUZA”
IAŞI

Adresa autorului: Universitatea Alexandru Ioan Cuza Iaşi
Facultatea de Informatică
str. Berthelot 16
700483, Iaşi
grigoras@infoiasi.ro, http://www.infoiasi.ro/~grigoras

Cuprins

1. Limbaje

Acest capitol trece în revistă chestiuni legate de limbajele de programare de nivel înalt, precum şi elemente de limbaje formale necesare abordării problemelor de analiză structurală în cadrul compilatoarelor . Este ştiut faptul că există şi sunt folosite un număr impresionant de limbaje de programare. Vom examina caracteristicile câtorva din aceste limbaje, lucru util pentru compararea lor şi, prin urmare, pentru alegerea limbajului potrivit pentru o aplicaţie anume. De asemenea, caracteristicile limbajelor de programare sunt importante pentru proiectarea şi implementarea compilatoarelor.

Limbajul este instrumentul principal al comunicării. Gradul de succes sau de eşec al comunicării este influenţat de caracteristicile limbajului folosit. Comunicarea între subiecţi de o anume specialitate se poate face prin intermediul unui limbaj ce incorporează o terminologie specială care face ca multe părţi ale comunicării să fie neinteligibile pentru un neiniţiat. Tot aşa, comunicarea cu computerul presupune utilizarea unui vocabular şi a unei gramatici specializate în care regulile – chiar dacă sunt mai uşor de specificat decât pentru limbajele naturale – trebuiesc cunoscute de ambele părţi. Oamenii comunică cu computerul prin limbaje specializate; există numeroase astfel de limbaje şi se proiectează mereu altele noi. Aceste limbaje sunt numite, unele, limbaje de programare, altele limbaje de comandă sau limbaje de interogare a bazelor de date etc.

1.1 Limbaje de programare

Limbajele de programare sunt instrumente folosite pentru a construi descrieri formale ale algoritmilor. Un limbaj de programare trebuie să conţină, pe lângă operaţiile de bază ce transformă o *stare iniţială* dată într-o *stare finală*, şi modul în care paşii ce conţin aceste operaţii sunt înlănţuiţi pentru a rezolva o anume problemă. Pentru a răspunde la aceste cerinţe, un limbaj de programare trebuie să conţină trei componente:

- *tipuri de date*, *obiecte* şi *valori* împreună cu *operaţiile* ce se pot aplica acestora;
- reguli pentru stabilirea *relaţiilor cronologice* între operaţiile specificate;
- reguli ce stabilesc structura (statică) a unui *program*.

Aceste componente constituie nivelul de *abstracţie* la care putem reprezenta algoritmi în limbajul respectiv. În raport cu acest nivel de abstracţie, limbajele de programare se pot clasifica – simplificând destul de mult lucrurile - în două grupe mari: *limbaje de nivel scăzut* şi *limbaje de nivel înalt*. Limbajele de nivel scăzut sunt mai apropiate de limbajele maşină: există o corespondenţă destul de puternică între operaţiile implementate de limbaj şi operaţiile implementate de hardware-ul corespunzător. Limbajele de nivel înalt, pe de altă parte, sunt mai apropiate de limbajele folosite de oameni pentru a exprima probleme şi algoritmi. Fiecare propoziţie într-un limbaj de nivel înalt poate să fie echivalentă cu o succesiune de operaţii (mai multe propoziţii) dintr-un limbaj de nivel scăzut. În realitate, există o mulţime de limbaje între cele două clase, de nivel scăzut şi de nivel înalt, şi nu se poate face o împărţire strictă a limbajelor doar în două clase.

Abstractizările într-un limbaj de programare permit programatorului să extragă proprietăţile esenţiale necesare pentru soluţia problemei de rezolvat, ascunzând detaliile de implementare a soluţiei. Cu cât nivelul de abstractizare este mai înalt, programatorul se gândeşte mai puţin la maşina (hardware-ul) pe care va fi implementată soluţia problemei. Cu alte cuvinte, ideile de programare pot fi separate în întregime de arhitectura sistemului pe care va fi executat programul. De pildă, în limbajele de programare de nivel înalt, programatorul lucrează cu nume de variabile simbolice şi nu cu adrese numerice de memorie. Abstracţiile de date permit ca locaţiile de memorie să fie privite drept celule de memorare pentru tipuri de date de nivel înalt; programatorul nu trebuie să-şi pună problema reprezentării acestora. Abstracţiile de control permit programatorului să exprime algoritmii cu ajutorul structurilor de tipul `if`, `while` sau al procedurilor. Cum sunt implementate acestea în maşina respectivă, este un lucru care nu-l interesează, în general, pe programator.

Un program scris într-un limbaj de nivel înalt este tradus în cod maşină echivalent, cu ajutorul unui program special numit *compilator*. În această carte vom prezenta unele din tehnicile cunoscute pentru proiectarea unui compilator.

1.1.1 Descrierea sintaxei şi semanticii unui limbaj

Limbajele, formal vorbind, sunt mulţimi de şiruri de caractere dintr-un alfabet fixat. Acest lucru este valabil atât pentru limbajele naturale, cât şi pentru cele artificiale. Şirurile de caractere ale limbajului se numesc *propoziţii* (*sentences*) sau *instrucţiuni*, *afirmaţii* (*statements*). Orice limbaj are un număr de reguli sintactice care stabilesc care din şirurile de caractere ce se pot forma cu simboluri din alfabet fac parte din limbaj. Dacă pentru limbajele naturale regulile sintactice sunt în general complicate, limbajele de programare sunt relativ simple din punct de vedere sintactic. Există în fiecare limbaj de programare unităţile sintactice de nivelul cel mai mic – numite *lexeme* – care nu sunt cuprinse în descrierea formală a sintaxei limbajului. Aceste unităţi sunt cuprinse într-o specificare lexicală a limbajului care precede descrierea sintaxei. Lexemele unui limbaj de programare cuprind identificatorii, literalii, operatorii, cuvintele rezervate, semnele de punctuaţie. Un program poate fi privit ca un şir de lexeme şi nu un şir de caractere;

obţinerea şirului de lexeme din şirul de caractere (programul obiect) este sarcina *analizorului lexical*. Un *token* al unui limbaj este o categorie de lexeme. De exemplu, un identificator este un token care are ca instanţe lexeme ca: `suma`, `total`, `i`, `par5`, etc. Tokenul PLUS pentru operatorul aritmetic "+" are o singură instanţă. Instrucţiunea:

```
delta = b2 - ac
```

este compusă din lexemii şi tokenurile specificate în tabela următoare:

Lexem	Token
`delta`	IDENTIFICATOR
`=`	ASIGNARE
`b2`	IDENTIFICATOR
`-`	MINUS
`ac`	IDENTIFICATOR

Un limbaj de programare poate fi descris, formal, prin aşa–zisele mecanisme de generare a şirurilor din limbaj. Aceste mecanisme se numesc gramatici şi au fost descrise de Chomsky în anii 1956 – 1959. Nu mult după ce Chomsky a descoperit cele patru clase de limbaje formale, grupul ACM – GAMM a proiectat limbajul Algol 58 iar la o conferinţă internaţională John Backus a prezentat din partea grupului acest limbaj folosind o metodă formală nouă de descriere a sintaxei. Această nouă notaţie a fost modificată de Peter Naur la descrierea limbajului Algol 60. Metoda de descriere a sintaxei limbajelor de programare aşa cum a fost folosită de Naur poartă numele de **forma Backus - Naur** sau simplu **BNF**. De remarcat faptul că BNF este aproape identică cu mecanismul generativ al lui Chomsky – gramatica independentă de context. BNF este un metalimbaj. Acesta foloseşte abstracţiile pentru a descrie structurile sintactice. O abstracţie este scrisă între paranteze unghiulare, ca de exemplu: <asignare>, <variabilă>, <expresie>, <if_stm>, <while_stm> etc. Definiţia unei abstracţii este dată printr-o *regulă* sau *producţie* care este formată din:

- partea stângă a regulii ce conţine abstracţia care se defineşte.
- o săgeată → (sau caracterele ::=) care desparte partea stângă a regulii de partea dreaptă.
- partea dreaptă a regulii care este un şir format din abstracţii, tokenuri, lexeme.

De exemplu, regula:

<asignare> → <variabilă> = <expresie>

defineşte sintaxa unei asignări: abstracţia <asignare> este o instanţă a abstracţiei <variabilă> urmată de lexemul = urmat de o instanţă a abstracţiei <expresie>. O propoziţie care are structura sintactică descrisă de această regulă este:

```
total = suma1 + suma2
```

Într-o regulă BNF abstracţiile se numesc *simboluri neterminale* sau simplu *neterminali*, lexemii şi token-urile se numesc *simboluri terminale* sau simplu *terminali*. O descriere BNF sau o gramatică (independentă de context) este o colecţie de reguli.

Dacă se întâmplă ca un neterminal să aibă mai multe definiţii acestea se pot insera într-o singură regulă în care partea dreaptă conţine părţile drepte ale definiţiilor sale despărţite prin simbolul |. De exemplu, definiţiile:

<if_stm> → `if` <expr_logică> `then` <stm>
<if_stm> → `if` <expr_logică> `then` <stm> `else` <stm>

se scriu:

<if_stm> → `if` <expr_logică> `then` <stm>
| `if` <expr_logică> `then` <stm> `else` <stm>

Iată un exemplu de gramatică (BNF) ce descrie un minilimbaj de programare:

<program> → `begin` <lista_de_instructiuni> `end`
<lista_de_instructiuni>→ <instructiune>

| <lista_de_instructiuni>; <instructiune>

<instructiune> → <variabila> = <expresie>

<variabila> → LITERA

<expresie> → <expresie> + <expresie>
| <expresie> - <expresie>
| <variabila>
| NUMAR

Aici apar două tokenuri: LITERA şi NUMAR. Acestea pot la rândul lor să fie descrise prin acelaşi mecanism:

LITERA → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

NUMAR → <cifra> | NUMAR <cifra>

<cifra> → 0|1|2|3|4|5|6|7|8|9

Se observă că de data aceasta gramatica este de tip 3 (sau gramatică regulată) ceea ce înseamnă că tokenurile se pot descrie folosind expresiile regulate iar recunoaşterea lor (*analiza lexicală*) se poate face cu automate finite deterministe.

BNF a fost extinsă în diverse moduri mai ales din raţiuni de implementare a unui generator de analizor sintactic. Orice extensie a BNF este numită EBNF (Extended BNF). Extensiile BNF nu măresc puterea descriptivă a mecanismului ci reprezintă facilităţi privind citirea sau scrierea unităţilor sintactice. Extensiile ce apar frecvent sunt:

- includerea, în partea dreaptă a unei reguli, a unei părţi opţionale ce se scrie între paranteze drepte. De exemplu, instrucţiunea if ar putea fi descrisă astfel:
  <if_stm> → `if` <expr_logică> `then` <stm> [`else` <stm>]
- includerea în partea dreaptă a unei reguli a unei părţi ce se repetă de zero sau mai multe ori. Această parte se scrie între acolade şi poate fi folosită pentru a descrie recursia:
  <lista_de_instructiuni> → <instructiune> {; <instructiune>}
- includerea în partea dreaptă a unei reguli a opţiunii de a alege o variantă dintr-un grup. Acest lucru se face punând variante în paranteze rotunde, despărţite prin |. De exemplu, o instrucţiune `for` se poate defini astfel:

<for_stm> → `for` <var> = <expr> (`to` | `downto`) <expr> `do` <stm>

Trebuie precizat că parantezele de orice fel din descrierea EBNF fac parte din metalimbaj, ele nu sunt terminali ci doar instrumente de a nota o anume convenţie. În cazul în care unele din aceste simboluri sunt simboluri terminale în limbajul descris, acestea vor fi puse între apostrof : '[' sau '{' etc.

EBNF nu face altceva decât să simplifice într-un anume fel definiţiile sintactice. Spre exemplificare iată cum se descrie expresia aritmetică folosind cele două mecanisme:

BNF:
<expresie> → <expresie> + <termen>
| <expresie> - <termen>
| <termen>
<termen> → <termen> * <factor>
| <termen> / <factor>
| <factor>
<factor> → '(' <expresie> ')'
| <identificator>

EBNF:
<expresie> → <termen> {(+| -) <termen>}
<termen> → <factor> {(* | /) <factor>}
<factor> → '('<expresie> ')'| <identificator>

Un exemplu de program în minilimbajul de programare descris mai sus este următorul:

```
begin a = b + c ; d = 5 - a end
```

O *derivare* a acestui program în gramatica dată este:

<program> ⇒ `begin` <lista_de_instructiuni> `end`
⇒ `begin` <lista_de_instructiuni>; <instructiune> `end`
⇒ `begin` <instructiune>; <instructiune> `end`
⇒ `begin` <variabila>=<expresie>; <instructiune> `end`
⇒ `begin LITERA=`<expresie>; <instructiune> `end`
⇒ `begin LITERA=`<expresie>+<expresie>; <instructiune> `end`
⇒ `begin LITERA=LITERA+`<expresie>; <instructiune> `end`
⇒ `begin LITERA=LITERA+LITERA`; <instructiune> `end`
⇒ `begin LITERA=LITERA+LITERA;` <variabila>=<expresie>`end`
⇒ `begin LITERA=LITERA+LITERA;LITERA=`<expresie>`end`
⇒ `begin LITERA=LITERA+LITERA;LITERA=`<expresie>-<expresie> `end`
⇒ `begin LITERA=LITERA+LITERA;LITERA=NUMAR-`<expresie> `end`
⇒ `begin LITERA=LITERA+LITERA;LITERA=NUMAR-LITERA end`

Arborele de derivare pentru acest program este dat în figura 1.1. Se observă o strânsă legătură între derivările dintr-o gramatică şi arborii de parsare.

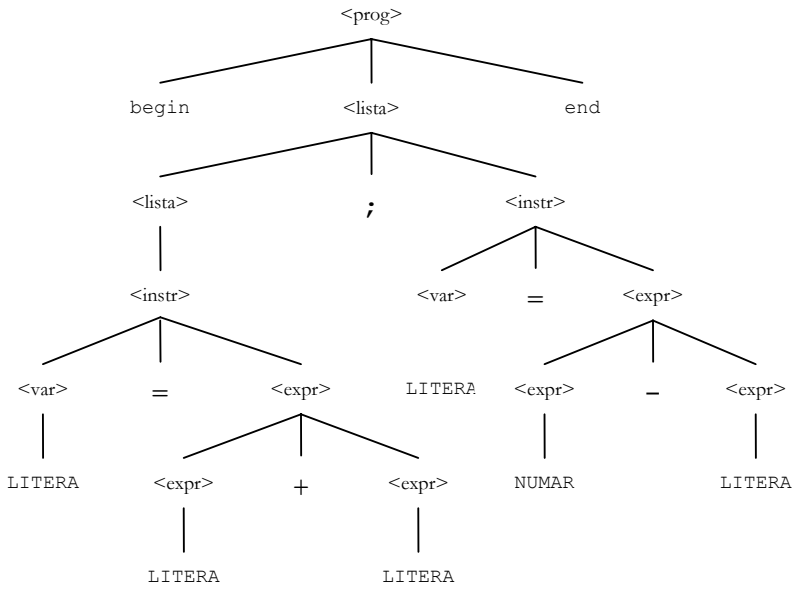


Figura 1.1

Există reguli ale limbajelor de programare care nu pot fi descrise utilizând BNF. De pildă faptul că o variabilă nu poate fi utilizată înainte de a fi declarată este o caracteristică ce nu poate fi descrisă prin BNF. Reguli de acest fel fac parte din *semantica statică* a limbajului: reguli ce ţin indirect de înţelesul unui program relativ la execuţie. Semantica statică este denumită aşa pentru că poate fi verificată înainte de execuţie, la momentul compilării. Un mecanism care descrie semantica statică a fost introdus de Knuth în 1968 şi este denumit gramatică cu atribute. O *gramatică cu atribute* este o gramatică independentă de context, care descrie sintaxa, la care s-au adăugat următoarele:

- fiecărui simbol al gramaticii i se ataşează o mulţime de *atribute*, unele dintre ele sunt *sintetizate* (valorile lor depind de valorile atributelor descendenţilor), altele *moştenite* (valorile lor depind de valorile atributelor părinţilor şi fraţilor);
- fiecărei reguli din gramatică i se asociază o mulţime de *funcţii semantice* care determină valorile atributelor sintetizate ale părţii stângi şi valorile atributelor moştenite ale simbolurilor din partea dreaptă.

Un arbore de derivare într-o gramatică cu atribute este arborele de derivare din gramatica ce descrie sintaxa împreună cu mulţimea valorilor atributelor ataşate fiecărui nod al arborelui. Pentru a evita *circularitatea* în obţinerea valorilor atributelor, deseori se pune restricţia ca un atribut moştenit să depindă doar de valorile atributelor părţii stângi a regulii şi de atributele simbolurilor din stânga sa în şirul din partea dreaptă a regulii. Această restricţie permite proiectarea unui evaluator care parcurge arborele de derivare în preordine. Nodurile frunze din arborele de derivare au atribute sintetizate ale căror valori se determină în afara arborelui de derivare; acestea se numesc *atribute intrinseci*. Spre exemplu, tipul unei instanţe a unei variabile într-un program se obţine din tabela simbolurilor care este construită la analiza lexicală şi conţine numele variabilelor, tipul lor etc.. În capitolul *Analiza semantică* vor fi date exemple de gramatici cu atribute şi modul de evaluare a atributelor.
Descrierea înţelesului unui program înseamnă *semantica dinamică*. Este vorba despre înţelesul expresiilor, a instrucţiunilor, a unităţilor de program. Dacă pentru descrierea sintaxei formalismul BNF este universal acceptat, nu există încă o notaţie universal acceptată pentru semantică (în continuare semantica dinamică o să fie numită semantică). Şi aceasta în condiţiile în care fiecare programator trebuie să cunoască atunci când foloseşte un limbaj, ce transformări face fiecare din instrucţiunile sale şi cum sunt obţinute aceste transformări. De obicei ei "învaţă" semantica citind text în limbaj natural care descrie fiecare instrucţiune. Or se ştie că, din multe motive, aceste explicaţii sunt imprecise şi / sau incomplete. Pe de altă parte, cei care implementează limbajele deseori folosesc explicaţiile în limbaj natural atunci când proiectează compilatoarele. Utilizarea semanticii în limbaj natural atât de către programatori, cât şi de către implementatori se poate explica prin aceea că mecanismele formale de descriere a semanticii sunt în general complexe şi greu de utilizat practic. De aceea domeniul acesta este încă în atenţia multor cercetători.

1.1.2 Implementarea limbajelor

Două din componentele primare ale unui computer sunt *memoria internă* şi *procesorul*. Memoria internă este folosită pentru păstrarea programelor şi a datelor. Procesorul este o colecţie de circuite care implementează operaţii sau instrucţiuni maşină. În absenţa altor componente software un computer înţelege doar propriul limbaj maşină. Pentru a fi folosit, computerul are nevoie de un sistem de operare - care reprezintă interfaţa pentru alte sisteme software - şi de un număr de limbaje de programare care să faciliteze rezolvarea de probleme diverse.
Limbajele de programare pot fi implementate prin trei metode generale: compilare, interpretare şi sisteme de implementare hibride.
Un *compilator* este un sistem software care traduce programele scrise într-un limbaj de programare (*programe sursă*) în programe scrise în limbaj maşină (*programe obiect*) ce pot fi executate direct de către computer. Avantajul acestei metode de implementare este acela că execuţia programelor este foarte rapidă. Procesul de compilare, în cele mai multe cazuri, se desfăşoară după schema din figura 1.2. *Analizorul lexical* identifică în textul sursă *unităţile lexicale* (tokenurile) şi transformă acest text într-un şir de unităţi lexicale,

ignorând spaţiile albe şi comentariile. Unităţi lexicale într-un limbaj de programare sunt identificatorii, operatorii, cuvintele rezervate, semnele speciale.

Analizorul sintactic preia şirul de unităţi lexicale de la analizorul lexical şi construieşte arborele de derivare (de parsare) al acestui şir în gramatica ce descrie sintaxa limbajului de programare. De fapt, de cele mai multe ori, se obţine o reprezentare a acestui arbore de derivare, un set de informaţii din care poate fi construit acest arbore.

Tabela de simboluri, construită de cele două analizoare amintite mai sus, cuprinde informaţii privind numele, tipul, atributele tuturor identificatorilor definiţi de utilizator în program. Aceste informaţii sunt necesare analizorului semantic şi generatorului de cod.

Analizorul semantic este de regulă un generator de cod intermediar care are rolul de a verifica erorile ce nu pot fi detectate de analizorul sintactic (de exemplu cele referitoare la tipul variabilelor) şi de a obţine din arborele de derivare un program echivalent cu cel iniţial, dar într-un limbaj intermediar între limbajul sursă şi limbajul maşină. O parte opţională a procesului de compilare este optimizarea. Aceasta are rolul de a îmbunătăţi programul obţinut, în ceea ce priveşte mărimea sa, prin eliminarea unor instrucţiuni sau în ceea ce priveşte viteza de calcul. Optimizarea, atunci când se face, este mai uşor de aplicat codului intermediar decât codului maşină.

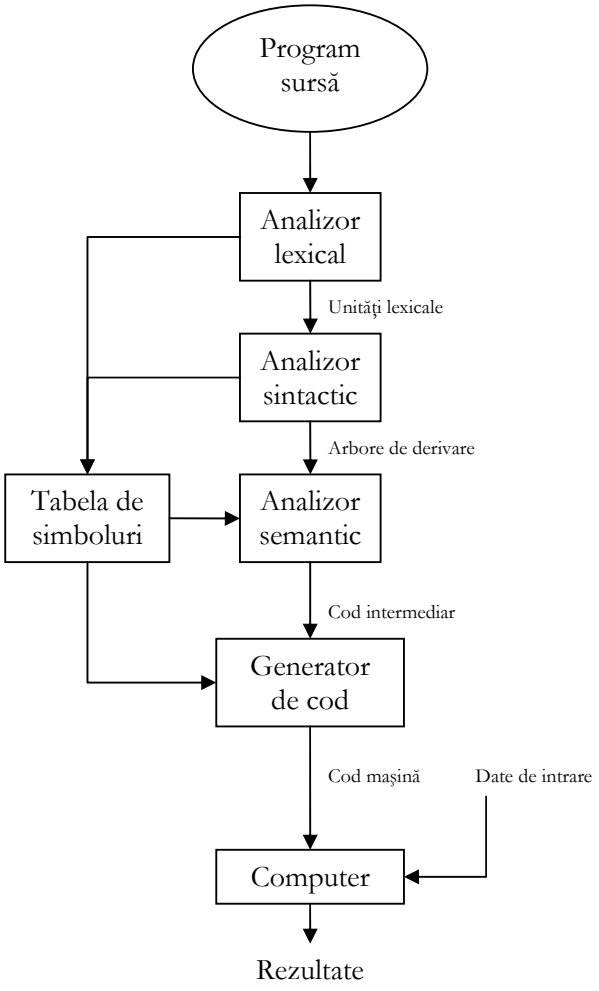


Figura 1.2

Generatorul de cod traduce codul intermediar (optimizat) într-un program echivalent scris în cod maşină. Acest program cod maşină are nevoie, de cele mai multe ori, pentru a fi executat, de module program oferite de sistemul de operare (cum ar fi programe de introducere sau extragere a datelor în/din memorie). Procesul de colectare a programelor sistem necesare execuţiei programului utilizator compilat şi de legare a lor la acesta se face de către un program special numit *linker*.

O tehnică destul de folosită în implementarea limbajelor de programare este *interpretarea* sau *interpretarea pură* cum mai este cunoscută în literatura de specialitate. Programele scrise în limbajul sursă sunt interpretate de către un alt program, numit *interpreter*, care acţionează ca un simulator software al unei maşini ce execută instrucţiune cu instrucţiune programul scris în limbajul sursă fără ca acesta să fie tradus în limbaj maşină. Simulatorul software oferă o maşină virtuală pentru limbajul sursă în cauză.

Avantajul interpretării este acela că permite implementarea uşoară a operaţiilor de depanare a programelor, pentru că erorile de execuţie se referă la unităţi ale limbajului sursă şi nu ţin de limbajul maşină. Din păcate, dezavantajele sunt mai numeroase: timpul de execuţie este serios diminuat în cazul interpretării faţă de compilare (un program interpretat se execută de 10 până la 100 ori mai lent decât unul compilat); spaţiul folosit de un program interpretat este mai mare; în cazul interpretării, semantica unei construcţii trebuie determinată din programul sursă la momentul execuţiei. Aceste dezavantaje fac ca doar limbajele cu structuri simple să fie implementate prin interpretare (comenzi sistem în UNIX, APL, LISP). Există şi excepţii: cu toate că nu este un limbaj simplu, JavaScript este interpretat.

Sistemele de *implementare hibridă* folosesc atât compilarea, cât şi interpretarea pentru implementarea limbajelor. Această tehnică se rezumă la obţinerea unui cod intermediar pentru programul scris în limbaj de nivel înalt (compilare) şi apoi interpretarea acestui cod intermediar. Desigur, se alege limbajul intermediar aşa fel ca el să poată fi uşor interpretat. Avantajul implementărilor hibride rezultă din aceea că obţinerea codului intermediar se poate face automat iar execuţia este mai rapidă decât în cazul interpretării pure. Ca exemple de implementări hibride consemnăm limbajul Perl şi implementările de început ale limbajului Java care este tradus în cod intermediar numit "byte code" şi poate fi portat pe orice maşină care dispune de un interpreter de "byte code" asociat cu un sistem de execuţie, în fapt ceea ce se numeşte *maşină virtuala Java*.

1.1.3 Analiza lexicală şi analiza sintactică

Sintaxa unui limbaj de programare se descrie folosind formalismul gramaticilor independente de context sau aşa zisa BNF – Backus Naur Form. Utilizarea BNF în descrierea sintaxei este avantajoasă pentru că este un formalism clar şi concis şi poate fi uşor transcris în sistemele software care implementează limbajul respectiv. În aproape toate compilatoarele, verificarea corectitudinii sintactice se face în două etape distincte: *analiza lexicală* şi *analiza sintactică*. *Analiza lexicală* se ocupă de construcţiile simple ale limbajului precum: identificatori, literali numerici, operatori, cuvinte rezervate, semne speciale. *Analiza sintactică* se ocupă de construcţiile evoluate din limbaj precum expresiile, instrucţiunile, unităţile de program. Tehnicile pentru analiza lexicală sunt mult mai puţin complexe faţă de cele ale analizei sintactice. Asta face ca implementarea separată a analizei lexicale să ducă la obţinerea unor module relativ mici şi uşor de întreţinut pentru fiecare din cele două faze. În esenţă un analizor lexical este un program pentru recunoaşterea subşirurilor de caractere dint-un şir dat, subşiruri ce se potrivesc cu un anume şablon (problema este cunoscută sub numele de pattern matching şi se întâlneşte frecvent în editoarele de text). Analizorul lexical citeşte textul sursă şi colectează caracterele în grupări logice care se numesc *lexeme*. Acestor grupări logice li se asociază coduri interne care se numesc *tokenuri*. De pildă, textul următor:

```
alpha = beta + 734;
```

în urma analizei lexicale se prezintă astfel:

Lexem	Token
alpha	IDENT
=	ASIGN
beta	IDENT
+	PLUS
734	INT_LIT
;	PV

IDENT, ASIGN etc. sunt nişte nume pentru nişte coduri numerice care vor fi returnate de către analizor atunci când grupul de caractere citit îndeplineşte condiţia de a fi identificator, operator de asignare etc. Pentru că spaţiile albe şi comentariile nu sunt relevante pentru execuţia unui program, acestea sunt ignorate de către analizorul lexical. Rolul analizorului lexical, în contextul compilării unui program, este de a citi textul sursă şi de a oferi analizorului sintactic tokenurile.

Partea de analiză a textului sursă care se referă la unităţile sintactice precum expresii, instrucţiuni, blocuri etc. poartă numele de *analiză sintactică* sau *parsare*. Rolul unui analizor sintactic este dublu. În primul rând, este acela de a verifica dacă programul este corect sintactic. Dacă se întâlneşte o eroare atunci analizorul produce un diagnostic şi analizează mai departe textul astfel încât să fie semnalate cât mai multe din erorile de sintaxă existente în text. Al doilea rol este acela de a construi arborele de derivare al programului respectiv în gramatica ce descrie limbajul. De fapt, sunt obţinute informaţiile din care se poate construi arborele de derivare: derivarea extrem stângă sau derivarea extrem dreaptă corespunzătoare, care însemna moduri de traversare a arborilor de derivare. Sunt cunoscute două tipuri de parsere, în funcţie de direcţia în care este construit arborele de derivare: *parsere top – down* (*descendente*) care construiesc arborele de la rădăcină către frunze şi *parsere bottom – up* (*ascendente*) care construiesc arborele de la frunze către rădăcină. Un parser *top – down* trasează derivarea extrem stângă a cuvântului de analizat. Arborele de derivare este construit în preordine: se începe cu rădăcina şi apoi fiecare nod este vizitat înainte ca descendenţii săi să fie generaţi. Aceştia sunt vizitaţi în ordinea de la stânga la dreapta. În termenii derivării, acest lucru se poate sintetiza astfel: dată o formă propoziţională (un cuvânt dintr-o derivare extrem stânga), sarcina parserului este de a determina următoarea formă propoziţională în derivare. Dacă forma propoziţională curentă este uAα unde u este un cuvânt format din terminali (tokenuri), A este neterminal iar α este un cuvânt format din terminali şi neterminali, atunci sarcina parserului este de a înlocui pe A cu partea dreaptă a unei reguli ce are pe A în partea stângă. Dacă regulile corespunzătoare lui A au în partea dreaptă respectiv β_1, β_2,..., β_n şi se alege β_k, atunci noua formă propoziţională este u$\beta_k\alpha$. În general, alegerea corectă a variantei de rescriere a lui A se face prin analiza simbolului următor lui u în cuvântul de analizat uv. Dacă acest lucru este posibil, atunci gramatica se numeşte gramatică LL(1) şi metoda de parsare se numeşte parsare LL(1). Implementarea se face fie prin transformarea fiecărei reguli din gramatică (forma BNF) într-un număr de linii de cod (parsare recursiv descendentă) fie prin utilizarea unei tabele de parsare pentru implementarea regulilor BNF. În capitolul al treilea vom trata pe larg parsarea LL(1).

Un parser *bottom-up* construieşte arborele de derivare pornind de la frunze şi înaintând spre rădăcină. Acesta produce reversul unei derivări extrem drepte a cuvântului de analizat. Parsarea bottom-up se poate descrie succint astfel: dată forma propoziţională dreapta (un cuvânt din derivarea extrem dreaptă) γ parserul trebuie să găsească în α partea dreaptă a unei reguli, iar aceasta se reduce la partea stângă şi se obţine forma propoziţională precedentă. Dacă se găseşte γ = αβu şi β este partea dreaptă a regulii ce

are în stânga A, atunci precedenta formă propoziţională este αAu. Algoritmii de analiză bottom-up sunt cei bazaţi pe relaţii de precedenţă sau, cei mai utilizaţi, cei din familia LR.

1.2 Limbaje formale

În acest paragraf trecem în revistă, succint, noţiuni de limbaje formale necesare în abordarea analizei lexicale, sintactice şi semantice în cadrul procesului de compilare. Un limbaj (formal) este o mulţime de cuvinte peste un alfabet Σ. Notăm prin Σ* limbajul total: mulţimea tuturor cuvintelor peste Σ. Dacă L_1 şi L_2 sunt limbaje, atunci la fel sunt şi reuniunea lor, intersecţia, complementara lui L_1 faţă de Σ*. De asemenea, dacă vom considera produsul uv a două cuvinte u şi v ca fiind un nou cuvânt ce se obţine prin concatenarea celor două, notăm prin L_1L_2 produsul celor două limbaje care înseamnă mulţimea cuvintelor uv unde u este din L_1 iar v este din L_2. Vom nota prin ε cuvântul nul, cel fără de litere. Atunci putem vorbi de puterea L^n a unui limbaj L folosind produsul şi stabilind că L^0 este limbajul ce conţine doar ε. Considerăm şi iteraţia unui limbaj L, notată L* ca fiind reuniunea limbajelor L^n pentru n = 0, 1, 2, ... Este cunoscută ierarhia lui Chomsky în care limbajele se clasifică în *limbaje regulate*, *limbaje independente de context*, *limbaje dependente de context* şi *limbaje de tip 0*. Limbajele regulate joacă un rol important în analiza lexicală. Unităţile lexicale pot fi descrise cu ajutorul *expresiilor regulate* (algebric) sau al gramaticilor regulate (generativ) şi pot fi recunoscute cu ajutorul *automatelor finite*. În esenţă, un *analizor lexical* este implementarea unui automat finit. Vom trece în revistă pentru început chestiunile legate de expresii regulate automate finite, fără a intra în detalii. Apoi, vom aminti câte ceva despre gramatici independente de context, instrumentul generativ pentru descrierea sintaxei limbajelor de programare. Cititorul este îndemnat a parcurge lucrări precum [Juc99], [Gri86] pentru a afla amănunte, mai ales de ordin teoretic. Noi suntem interesaţi aici de utilizarea acestor modele de calcul pentru dezvoltarea de algoritmi de analiză structurală a textelor sursă încât, vom prezenta atât cât este necesar pentru acest lucru.

1.2.1 Expresii regulate

Definiţia 1.2.1 Fie Σ un alfabet, simbolurile ε, Φ, |, •, *, (,) care nu aparţin lui Σ şi E un cuvânt peste alfabetul Σ ∪ { ε, Φ, |, •, *,),(}. O *expresie regulată* peste Σ se defineşte inductiv astfel:

1. E este un *atom regulat* peste Σ dacă E este un simbol din Σ ∪ { ε, Φ} sau este de forma (E_1) unde E_1 este o expresie regulată peste Σ;
2. E este un *factor regulat* peste Σ dacă E este un atom regulat peste Σ sau este de forma E1* unde E_1 este un factor regulat peste Σ ;
3. E este un *termen regulat* peste Σ dacă E este un factor regulat peste Σ sau este de forma $E_1 \bullet E_2$ unde E1 este un termen regulat, iar E_2 este un factor regulat peste Σ;
4. E este o *expresie regulată* peste Σ dacă E este un termen regulat peste Σ sau este de forma $E_1 \mid E_2$, unde E_1 este o expresie regulată, iar E_2 este un termen regulat peste Σ.

Aici ε, Φ sunt privite ca simple simboluri fără vreo semnificaţie. Mai jos, interpretarea acestor expresii va fi desigur limbajul {ε} respectiv limbajul vid.

În definiţia de mai sus, pe lângă noţiunea de expresie regulată se dau cele de termen regulat, factor regulat, atom regulat care reflectă precedenţa şi asociativitatea operatorilor |, •, *. În cele ce urmează vom omite scrierea operatorului • aşa cum se obişnuieşte la scrierea operatorului de înmulţire.

Definiţia 1.2.2 Limbajul L(E) descris (sau notat, sau denotat) de expresia regulată E peste alfabetul Σ este definit inductiv astfel:

1. $L(\Phi) = \Phi$, $L(\varepsilon) = \{\varepsilon\}$, $L(a) = \{a\}$, pentru orice a din Σ;
2. $L((E)) = L(E)$, pentru orice expresie regulată E peste Σ;
3. $L(E^*) = (L(E))^*$, pentru orice factor regulat E peste Σ;
4. $L(E_1E_2) = L(E_1)L(E_2)$, pentru orice E_1, termen regulat, şi pentru orice E_2 factor regulat peste Σ ;
5. $L(E_1 \mid E_2) = L(E_1) \cup L(E_2)$, pentru orice E_1 expresie regulată, şi pentru orice E_2 termen regulat peste Σ.

Proprietăţile 1 şi 2 de mai sus asigură faptul că $L(E_1E_2)$ şi $L(E_1 \mid E_2)$ sunt bine definite.

Exemplul 1.2.1 Fie $E_1 = \varepsilon \mid (a \mid b)^*(ab \mid baa)$ o expresie regulată.

Atunci limbajul descris de E_1 este:

$$\begin{aligned} L(E_1) &= L(\varepsilon \mid (a \mid b)^*(ab \mid baa)) = L(\varepsilon) \cup L((a \mid b)^*(ab \mid baa)) = \\ &= \{\varepsilon\} \cup L((a \mid b)^*)\, L((ab \mid baa)) = \\ &= \{\varepsilon\} \cup (L(a \mid b))^*\, L(ab \mid baa) = \\ &= \{\varepsilon\} \cup (L(a \mid b))^*\, (L(ab) \cup L(baa)) = \\ &= \{\varepsilon\} \cup (L(a) \cup L(b))^*\, (L(a)L(b) \cup L(b)L(a)L(a)) = \\ &= \{\varepsilon\} \cup (\{a\} \cup \{b\})^*(\{a\}\{b\} \cup \{b\}\{a\}\{a\}) = \\ &= \{\varepsilon\} \cup \{a,b\}^*\{ab, baa\}. \end{aligned}$$

Analog pentru $E_2 = (a \mid b)(a \mid b)$ obţinem $L(E_2) = \{aa, ab, ba, bb\}$

Este binecunoscut faptul că mulţimea limbajelor descrise de expresiile regulate coincide cu mulţimea limbajelor regulate [Juc99]. Asta înseamnă că pentru fiecare expresie regulată E poate fi construit (efectiv) un automat finit care să recunoască exact limbajul descris de expresia E. De asemenea, pentru orice automat finit, există o expresie regulată (care poate fi obţinută efectiv) ce descrie limbajul recunoscut de el. Cum suntem interesaţi în proiectarea unui analizor lexical, parte a unui compilator, vom descrie algoritmic în paragrafele următoare trecerea de la o expresie regulată la un automat finit; algoritmii diferă substanţial de cei utilizaţi în demonstraţiile teoretice.

Definiţia 1.2.3 Două expresii regulate E_1, E_2 peste alfabetul Σ, sunt echivalente dacă $L(E_1) = L(E_2)$. Notăm acest lucru prin $E_1 = E_2$.

Exemplul 1.2.2 Se verifică uşor că au loc următoarele :

$$a \mid b = b \mid a \quad \text{şi} \quad \varepsilon \mid (a \mid b)^* = (a \mid b)^*.$$

Lema 1.2.1 (Proprietăţi algebrice ale expresiilor regulate). Fie E, E_1, E_2, E_3 expresii regulate peste alfabetul Σ. Sunt adevărate următoarele:

1. $E_1 \mid E_2 = E_2 \mid E_1$
2. $(E_1 \mid E_2) \mid E_3 = E_1 \mid (E_2 \mid E_3)$
3. $(E_1E_2)E_3 = E_1(E_2E_3)$
4. $E_1(E_2 \mid E_3) = E_1E_2 \mid E_1E_3$
5. $(E_1 \mid E_2)E_3 = E_1E_3 \mid E_2E_3$
6. $\varepsilon E = E$, $E\varepsilon = E$, $E\Phi = \Phi$, $\Phi E = \Phi$
7. $E^* = (E \mid \varepsilon)^*$, $\Phi^* = \varepsilon$
8. $(E^*)^* = E^*$
9. $(E1^*E2^*)^* = (E1 \mid E2)^*$

Demonstraţie Rezultă din definiţia anterioară şi proprietăţile operaţiilor cu mulţimi.

□

În procesul de descriere a limbajelor, un rol important îl joacă noţiunea de ambiguitate. Dacă $\mathcal{D}$ este o descriere a unui limbaj iar $L(\mathcal{D})$ este limbajul descris de $\mathcal{D}$, spunem că $\mathcal{D}$ este ambiguă dacă există măcar un cuvânt din $L(\mathcal{D})$ care este descris în două moduri diferite.

De exemplu expresia regulată E = ab | (a | b)* este ambiguă deoarece cuvântul ab în L(E) are două descrieri: una în expresia ab şi alta în (a | b) *.

Formal, neambiguitatea unei expresii regulate este definită inductiv astfel:

Definiţia 1.2.4 (Expresii regulate neambigue)

1. Expresiile regulate Φ, ε, a şi (E) sunt neambigue, pentru orice a din Σ şi oricare ar fi expresia neambiguă E peste Σ ;
2. Pentru orice factor E peste Σ, E* este neambiguă dacă E este neambiguă şi pentru orice $u \in L(E^*)$ există exact un număr $n \geq 0$ şi o secvenţă unică $(u_1, u_2, \ldots, u_n)$ de cuvinte din L(E) astfel încât $u_1u_2\ldots u_n = u$ (pentru cazul $n = 0$, $u_1u_2\ldots u_n = \varepsilon$) ;
3. Pentru orice termen E_1 şi pentru orice factor E_2 peste Σ, E_1E_2 este neambiguă dacă:
   a. $L(E_1E_2) = \Phi$, sau
   b. E_1 şi E_2 sunt neambigue şi pentru orice $u \in L(E_1E_2)$ există o pereche unică (u_1,u_2) în $L(E_1) \times L(E_2)$ astfel încât $u_1u_2 = u$.
4. Pentru orice expresie E_1 şi orice termen E_2, expresia $E_1 \mid E_2$ este neambiguă dacă E_1 şi E_2 sunt neambigue şi în plus $L(E_1) \cap L(E_2) = \Phi$.

1.2.2 Automate finite

Definiţia 1.2.5 Un automat finit este sistemul $A = (Q, \Sigma, \delta, q_0, F)$ unde Q şi Σ sunt mulţimi finite, nevide, numite mulţimea stărilor respectiv *alfabetul de intrare*, $q_0 \in Q$ este *starea iniţială*, $F \subseteq Q$ este *mulţimea stărilor finale* iar δ este o funcţie

$$\delta: Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q,$$

numită *funcţia de tranziţie* (unde prin 2^Q am notat mulţimea părţilor lui Q).

Observaţia 1.2.1 Modelul definit mai sus este cel cunoscut în literatură şi sub denumirea de *sistem tranziţional* (sau automat *nedeterminist cu ε - tranziţii*). Prin particularizări ale lui δ vom obţine modelul de *automat nedeterminist* (fără ε - tranziţii) şi cel de *automat determinist*.

Un automat finit poate fi reprezentat prin *tabela de tranziţie* (funcţia δ) sau prin *graful de tranziţie*. În reprezentarea grafului de tranziţie facem convenţia ca stările care nu sunt finale să le reprezentăm prin cercuri iar cele finale prin pătrate. De asemenea ε - tranziţiile sunt reprezentate prin arce neetichetate.

Exemplul 1.2.3 Fie $Q = \{0, 1, 2\}$, $\Sigma = \{a, b, c\}$, $F = \{2\}$, $q_0 = 0$, iar δ este dată astfel:

Tabela de tranziţie: **Graful de tranziţie:**

δ	a	b	c	ε
0	{0}	Φ	Φ	{1}
1	Φ	{1}	Φ	{2}
2	Φ	Φ	{2}	Φ

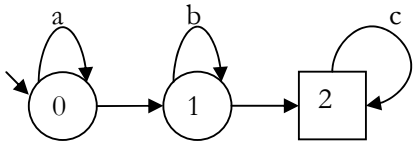


Figura 1.3

Exemplul 1.2.4 $Q = \{0, 1, 2\}$, $\Sigma = \{a, b\}$, $F = \{2\}$, $q_0 = 0$, iar δ este:

Tabela de tranziție: **Graful de tranziție:**

δ	a	b	ε
0	{0, 1}	Φ	Φ
1	{1}	{2}	Φ
2	{2}	Φ	{0, 1}

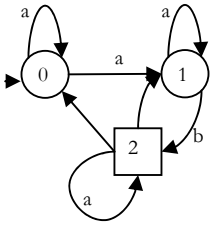


Figura 1.4

Exemplul 1.2.5 Q = {0, 1, 2, 3}, Σ = {a, b}, F = {0}, q_0 = 0, iar δ este dată în figura 1.5.

Tabela de tranziție: **Graful de tranziție:**

δ	a	b
0	1	3
1	0	2
2	3	1
3	2	0

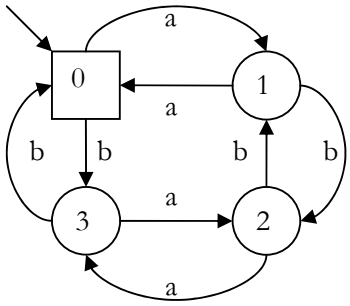


Figura 1.5

Exemplul 1.2.6 Q = {0, 1, 2,}, Σ = {a, b}, F = {2}, q_0 = 0, iar δ este:

Tabela de tranziție: **Graful de tranziție:**

δ	a	b	c
0	{0,1, 2}	{1, 2}	{2}
1	Φ	{1, 2}	{2}
2	Φ	Φ	{2}

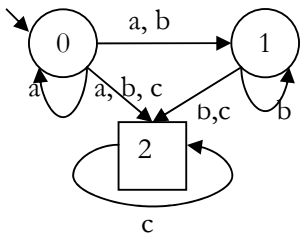


Figura 1.6

Definiția 1.2.6 Un automat A = (Q, Σ, δ, q_0, F) se numește:

1. *nedeterminist (fără ε-tranziţii),* dacă $\delta(q, \varepsilon) = \Phi$, $\forall q \in Q$
2. *determinist*, dacă $\delta(q, \varepsilon) = \Phi$, $\forall q \in Q$ şi $|\delta(q, a)| \leq 1$, $\forall q \in Q$, $\forall a \in \Sigma$

În literatura de specialitate, de obicei, un automat finit determinist are proprietatea că $|\delta(q, a)| = 1$, $\forall q \in Q$, $\forall a \in \Sigma$. Condiţia $|\delta(q, a)| \leq 1$, dată în definiţia precedentă, permite ca un automat determinist să fie parţial definit (eventual) dar nu este restrictivă relativ la puterea de calcul: putem obţine automatul total definit dacă adăugăm la automat o stare specială Φ (numită *stare moartă* sau *stare de blocare*) şi definim δ astfel:

$\delta(q, a) = \Phi$, dacă $|\delta(q,a)| < 1$ şi $\delta(\Phi, b) = \Phi$, $\forall b \in \Sigma$.

Aşadar, putem subînţelege, atunci când este nevoie, că este îndeplinită condiţia $|\delta(q,a)| = 1$ (de exemplu când un automat trebuie să „citească" în întregime orice cuvânt, chiar dacă nu-l acceptă).

Pentru definirea *limbajului acceptat (recunoscut)* de un automat finit, să extindem funcţia de tranziţie la cuvinte. Vom nota mai întâi, pentru $q \in Q$:

$Cl(q) = \{q' \mid q' \in Q$, în graful automatului A există un drum de la q la q'
de lungime $k \geq 0$ ale cărui arce sunt etichetate cu $\varepsilon\}$.

Facem precizarea că $q \in Cl(q)$ pentru că q este legat de q printr-un drum de lungime 0. Dacă $S \subseteq Q$, atunci notăm:

$Cl(S) = \bigcup_{q \in S} Cl(q)$ iar $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$.

Definiţia 1.2.7 Dacă $A = (Q, \Sigma, \delta, q_0, F)$ este un automat, atunci extensia lui δ la cuvinte este funcţia $\hat{\delta} : Q \times \Sigma^* \to 2^Q$ care se defineşte astfel:

1. $\hat{\delta}(q, \varepsilon) = Cl(q)$, $\forall q \in Q$;
2. $\hat{\delta}(q, ua) = Cl(\delta(\hat{\delta}(q, u), a))$, $\forall q \in Q$, $\forall u \in \Sigma^*$, $\forall a \in \Sigma$.

Lema 1.2.2 Fie $A = (Q, \Sigma, \delta, q_0, F)$ un automat finit, o stare $q \in Q$ şi un cuvânt $w \in \Sigma^*$, unde $w = a_1a_2\ldots a_n$. Atunci $q' \in \hat{\delta}(q, w)$ dacă şi numai dacă în graful automatului, există un drum de la q la q' de lungime $m \geq n$ cu arcele etichetate respectiv $y_1, y_2, \ldots, y_m$ (în această ordine) astfel încât $y_1y_2\ldots y_m = w$.

Demonstraţie.

1. $\Leftarrow$ Se demonstrează prin inducţie după lungimea lui w.
2. $\Rightarrow$ Se demonstrează prin inducţie după lungimea drumului de la q la q'.

□

Observaţia 1.2.2 Pentru automatele fără ε-tranziţie (în care $\delta(q, \varepsilon) = \Phi$, $\forall q \in Q$), deoarece $Cl(q) = \{q\}$ are loc $\hat{\delta}(q, a) = \delta(q, a)$, $\forall q \in Q$, $\forall a \in \Sigma$. Din acest motiv, pentru astfel de automate $\hat{\delta}$ va fi notată de asemenea cu δ.

În cazul automatelor cu ε - tranziţii vom păstra notaţia $\hat{\delta}$ pentru extensie pentru că, în general, $\hat{\delta}(q, \varepsilon) \neq \delta(q, \varepsilon)$ şi, de asemenea, $\hat{\delta}(q, a) \neq \delta(q,a)$, $a \in \Sigma$. De pildă, în Exemplul 1.2.4 avem:

$\hat{\delta}(0, a) = \{0, 1, 2\}$ iar $\delta(0, a) = \{0\}$ şi $\hat{\delta}(0, \varepsilon) = \{0, 1, 2\}$ iar $\delta(0, \varepsilon) = \{1\}$.

Lema 1.2.3 Fie $A = (Q, \Sigma, \delta, q_0, F)$ un automat finit şi cuvintele $u, v \in \Sigma^*$. Atunci are loc:

$$\hat{\delta}(q, uv) = \hat{\delta}(\hat{\delta}(q, u), v).$$

Demonstraţie. Inducţie după $|v|$.

□

Definiţia 1.2.8 *Limbajul acceptat* (*recunoscut*) de automatul $A = (Q, \Sigma, \delta, q_0, F)$ este mulţimea notată L(A) dată de expresia:

$$L(A) = \{w \mid w \in \Sigma^*, \hat{\delta}(q_0, w) \cap F \neq \Phi\}.$$

Aşadar, un cuvânt w este recunoscut de un automat A dacă, după *citirea* în întregime a cuvântului w, automatul (pornind din starea iniţială q_0) poate să ajungă într-o stare finală. În cazul automatelor finite deterministe, putem scrie:

$$L(A) = \{w \mid w \in \Sigma^*, \delta(q_0, w) \in F\}.$$

deoarece $\delta(q_0, w)$ este o mulţime formată dintr-un singur element şi identificăm această mulţime cu elementul respectiv.

Teorema 1.2.1 Pentru orice expresie regulată E peste Σ există un automat finit (cu ε - tranziţii) A, astfel încât L(E) = L(A).

Demonstraţie Procedăm prin inducţie asupra complexităţii expresiei E.

Dacă $E \in \{\Phi, \varepsilon, a\}$ ($a \in \Sigma$) atunci automatul corespunzător este respectiv:

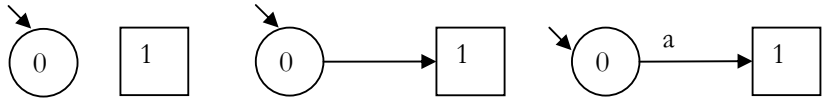


Figura 1.7

Dacă $E = E_1 \mid E_2$, $E = E_1E_2$ sau $E = E_1^*$ atunci presupunem că există automatele finite $A_1 = (Q_1, \Sigma, q_{01}, \delta_1, \{f_1\})$, $A_2 = (Q_2, \Sigma, q_{02}, \delta_2, \{f_2\})$, cu Q_1, Q_2 disjuncte şi $\delta_1(f_1, a) = \delta_2(f_2, a) = \Phi$, $\forall a \in \Sigma$ (ipoteza inductivă care are loc pentru $E = \Phi$, ε sau a), care recunosc limbajele $L(E_1)$ respectiv $L(E_2)$. Să considerăm două stări noi q_0, f (ce nu sunt din $Q_1 \cup Q_2$). Atunci, automatul $A = (Q, \Sigma, \delta, q_0, F)$ se construieşte astfel:

1. $E = E_1 \mid E_2 : Q = Q_1 \cup Q_2 \cup \{q_0, f\}$, $F = \{f\}$, $\delta(q_0, \varepsilon) = \{q_{01}, q_{02}\}$, $\delta(f_1, \varepsilon) = \delta(f_2, \varepsilon) = \{f\}$, $\delta(q, a) = \delta_i(q, a)\ \forall q \in Q_i, \forall a \in \Sigma, i = 1, 2.$

2. $E = E_1E_2 : Q = Q_1 \cup Q_2$, $q_0 = q_{01}$, $F = \{f_2\}$, $\delta(f_1, \varepsilon) = q_{02}$, $\delta(q, a) = \delta_i(q, a)\ \forall q \in Q_i, \forall a \in \Sigma, i = 1,2.$

3. $E = E_1^* : Q = Q_1 \cup \{q_0, f\}$, $\delta(q_0, \varepsilon) = \{q_{01}, f\}$, $\delta(f_1, \varepsilon) = \{q_{01}, f\}$, $\delta(q, a) = \delta_1(q, a)\ \forall q \in Q_1, \forall a \in \Sigma.$

Schematic, acest lucru se exprimă ca în figura 1.8.

Se dovedeşte uşor că automatele construite mai sus recunosc respectiv limbajele $L(E_1 \cup E_2)$, $L(E_1E_2)$, $L(E_1^*)$.

□

Clasa limbajelor recunoscute de automatele finite coincide cu clasa limbajelor descrise de expresii regulate şi această clasă este cunoscută sub numele de clasa limbajelor regulate (sau de tip 3). Această clasă conţine numai limbaje neambigue: un automat finit determinist este un mecanism neambiguu. De aici rezultă că pentru orice expresie regulată E există o expresie regulată E' neambiguă, echivalentă cu E. Teoretic vorbind, trecerea de la o expresie ambiguă E la echivalenta sa neambiguă E' se face astfel: se construieşte A, automatul finit determinist echivalent cu E (trecând prin automatul cu ε - tranziţii şi apoi prin cel nedeterminist fără ε - tranziţii) pentru ca apoi să se determine expresia regulată E' care descrie limbajul L(A); aceasta este neambiguă. Practic însă, descrierile ambigue pot să fie mai concise şi mai uşor de înţeles decât echivalentele lor neambigue (care costă şi timp substanţial pentru a fi obţinute).

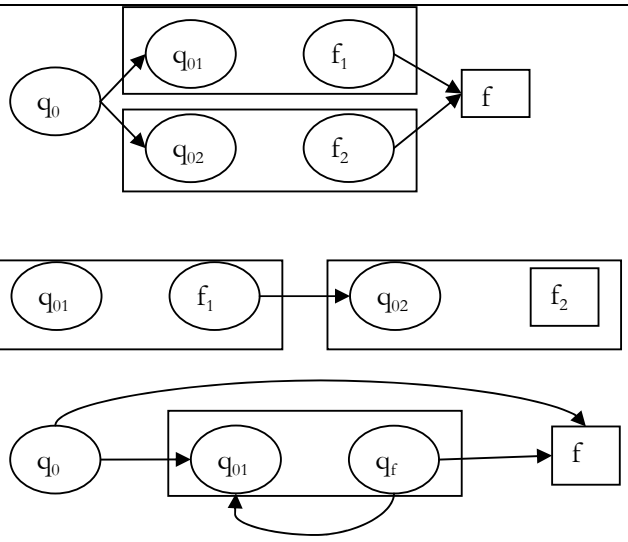


Figura 1.8

Exemplul 1.2.7 Limbajul L = {w ∈ {0,1}*|w conţine 000 sau 111} este descris de expresia ambiguă E = (0 | 1)*(000 | 111)(0 | 1)*. Expresia E' neambiguă, echivalentă cu E, în afară de faptul că se obţine greoi, nu mai exprimă în mod evident forma cuvintelor din L(E'). Cititorul se poate convinge de acest lucru printr-un exerciţiu.

1.2.3 Gramatici independente de context

Definiţia 1.2.9 O *gramatică independentă de context* este sistemul G = (V, T, S, P) unde V şi T sunt mulţimi nevide, finite, disjuncte de *neterminali* (*variabile*), respectiv *terminali*, Σ = V ∪ T se numeşte *vocabularul* gramaticii, S ∈ V este o variabilă specială numită *simbol de start* sau *axioma gramaticii*, iar P ⊆ V × (V ∪ T)* este *mulţimea regulilor de producţie (derivare)* sau simplu *mulţimea producţiilor*.

Convenţii de notaţie. În scopul simplificării expunerii, vom face câteva convenţii care vor fi păstrate de-a lungul acestei lucrări, cu excepţia situaţiilor în care se fac precizări suplimentare.

1. Următoarele simboluri notează *neterminali*:
   - S, folosit pentru a nota *axioma* gramaticii;
   - A, B, C …, (literele majuscule de la începutul alfabetului);
   - numele italice scrise cu minuscule: *expresie*, *instrucţiune*,...
2. Următoarele simboluri notează terminali:
   - literele mici de la începutul alfabetului: a, b, c,...
   - operatori: +, -, *, /, etc.
   - simboluri de punctuaţie, paranteze.
   - cifrele: 0, 1, …, 9
   - numele scrise cu fontul `courier` (unităţile lexicale): `id, if, while, begin,` etc..
3. Literele mari de la sfârşitul alfabetului, X, Y, Z,... notează simboluri gramaticale (neterminali sau terminali, adică elementele din Σ);
4. Literele mici de la sfârşitul alfabetului, u, v, x, y, z, w notează cuvinte din T* (formate din terminali);

5. Literele greceşti α, β, γ,… reprezintă cuvinte peste Σ. Aşadar, o producţie a gramaticii se va nota prin $A \to \alpha$ (în loc de $(A, \alpha) \in P$, convenim să scriem $A \to \alpha$).
6. Dacă $A \to \alpha_1$, $A \to \alpha_2$,…, $A \to \alpha_n$ sunt toate producţiile care au A în partea stângă (numite A-producţii), le vom nota prin:
$$A \to \alpha_1 \mid \alpha_2 \mid,\ldots, \mid \alpha_n$$
7. O gramatică va fi dată prin producţiile sale. De aici se subînţeleg mulţimile V şi T, iar simbolul de start este partea stângă a primei producţii.

Exemplul 1.2.8 Gramatica:

$$E \to EOE \mid (E) \mid -E \mid \texttt{id}$$
$$O \to \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

este constituită din elementele $V = \{E, O\}$, $T = \{$ `id, +, -, *, /, (,)` $\}$, E este simbolul de start, iar producţiile sunt cele indicate mai sus.

Definiţia 1.2.10 Fie $G = (V, T, S, P)$ o gramatică independentă de context. Relaţia de *derivare* în gramatica G este o relaţie din $\Sigma^*V\Sigma^* \times \Sigma^*$, notată $\Rightarrow_G$, şi este definită astfel:

$\delta_1 \Rightarrow_G \delta_2$ dacă şi numai dacă $\delta_1 = \alpha A \gamma$, $\delta_2 = \alpha\beta\gamma$ şi $A \to \beta$.

Vom nota prin $\Rightarrow^*_G$ ($\Rightarrow^+_G$) închiderea reflexivă şi tranzitivă (închiderea tranzitivă) a relaţiei $\Rightarrow_G$. Atunci când nu există confuzii, vom renunţa la a indica G în aceste relaţii (scriem simplu $\Rightarrow$, $\Rightarrow^*$, $\Rightarrow^+$). Dacă $\alpha \Rightarrow \beta$ spunem că α derivează într-un pas β ; dacă $\alpha \Rightarrow^*_G \beta$ ($\alpha \Rightarrow^+_G \beta$) spunem că α *derivează* β(α derivează propriu β):

$\alpha \Rightarrow^*_G \beta$ dacă şi numai dacă există $\delta_0, \delta_1,\ldots, \delta_n$, $n \geq 0$ astfel ca:
$$\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow \ldots \Rightarrow \delta_n = \beta$$
$\alpha \Rightarrow^+_G \beta$ dacă şi numai dacă există $\delta_0, \delta_1,\ldots, \delta_n$, $n \geq 1$ astfel ca:
$$\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow \ldots \Rightarrow \delta_n = \beta$$

Dacă $S \Rightarrow^*_G \gamma$ spunem că γ este *formă propoziţională* în gramatica G, iar dacă γ este format din terminali, $\gamma = w$, spunem că w este o frază în gramatica G.

Definiţia 1.2.11 *Limbajul generat* de gramatica G (notat L(G)) este mulţimea frazelor gramaticii G, $L(G) = \{w \mid w \in T^*, S \Rightarrow^+ w\}$.

Pentru că în această lucrare ne vom ocupa numai de gramatici independente de context, le vom numi pe acestea simplu gramatici. Pentru studiul ierarhiei lui Chomsky, cititorul este invitat a consulta lucrările [Gri86], [Juc99]. De asemenea, vom considera că gramaticile cu care lucrăm în continuare sunt gramatici reduse, adică orice simbol $X \in \Sigma$ este *simbol util*:

- este *accesibil*: $S \Rightarrow^+ \alpha X \beta$, pentru anumiţi $\alpha, \beta \in \Sigma^*$;
- este *productiv*: $X \Rightarrow^* w$, pentru un anume $w \in T^*$.

Este cunoscut faptul că pentru orice gramatică G există o gramatică G', cu $L(G) = L(G')$ (G şi G' se zic echivalente în acest caz), astfel încât orice simbol al vocabularului gramaticii G' este util (vezi [Gri86], [Juc99], [JuA02]). O gramatică fără simboluri inutile (simboluri care nu sunt utile) se numeşte *gramatică redusă*. În continuare vom considera numai gramatici reduse.

1.2.4 Arbori sintactici. Ambiguitate

Un *arbore sintactic* ilustrează într-o gramatică modul în care axioma generează o (se rescrie într-o) frază a gramaticii. Ideea care stă la baza ataşării unui arbore unei derivări este aceea că, dacă într-o derivare se foloseşte o producţie de forma $A \to X_1X_2 \ldots X_n$, atunci în arbore există un nod interior etichetat A care are n descendenţi etichetaţi respectiv cu $X_1, X_2, \ldots, X_n$ de la stânga la dreapta.

Definiţia 1.2.12 Fie G = (V, T, S, P) o gramatică. Un *arbore sintactic* (*arbore de derivare*, *arbore de parsare*) în gramatica G este un arbore ordonat, etichetat, cu următoarele proprietăţi:

1. rădăcina arborelui este etichetată cu S ;
2. fiecare frunză este etichetată cu un simbol din T sau cu ε ;
3. fiecare nod interior este etichetat cu un neterminal;
4. dacă A etichetează un nod interior care are n succesori etichetaţi de la stânga la dreapta respectiv cu X_1, X_2, …, X_n, atunci $A \to X_1X_2 \dots X_n$ este o producţie. Cazul în care producţia este $A \to \varepsilon$ este un caz special: nodul etichetat A are un singur descendent etichetat ε.

Frontiera unui arbore de derivare este cuvântul $w = a_1a_2\dots a_n$ unde a_i, $1 \le i \le n$ sunt etichetele nodurilor frunză în ordinea de la stânga la dreapta.

Exemplul 1.2.10 Să considerăm gramatica care descrie expresiile aritmetice ce se construiesc cu `+`, `*`, `id` şi paranteze:

$$E \to E+E \mid E*E \mid (E) \mid \texttt{id}.$$

Un arbore de derivare în această gramatică este prezentat în figura 1.9.

Să observăm că acestui arbore de derivare îi putem ataşa derivările:

$$E \Rightarrow E+E \Rightarrow \texttt{id}+E \Rightarrow \texttt{id}+E*E \Rightarrow \texttt{id}+\texttt{id}*E \Rightarrow \texttt{id}+\texttt{id}*\texttt{id}$$

$$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*\texttt{id} \Rightarrow E+\texttt{id}*\texttt{id} \Rightarrow \texttt{id}+\texttt{id}*\texttt{id}$$

$$E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+\texttt{id}*E \Rightarrow \texttt{id}+\texttt{id}*E \Rightarrow \texttt{id}+\texttt{id}*\texttt{id}$$

Este clar că cele trei derivări (mai pot fi scrise şi altele) sunt distincte. Ele diferă prin ordinea de aplicare a regulilor. În prima derivare, la fiecare pas s-a rescris cea mai din stânga variabilă a formei propoziţionale (curente); astfel de derivări se numesc *derivări extrem stângi*. Vom nota o derivare extrem stânga prin $\underset{st}{\Rightarrow}$:

$$u A\alpha \underset{st}{\Rightarrow} u\beta\alpha \text{ dacă } u \in T^*, A \to \beta \in P, \alpha \in \Sigma^*$$

Aşadar, prima din derivările de mai sus se scrie: $E \overset{+}{\underset{st}{\Rightarrow}} \texttt{id}+\texttt{id}*\texttt{id}$.

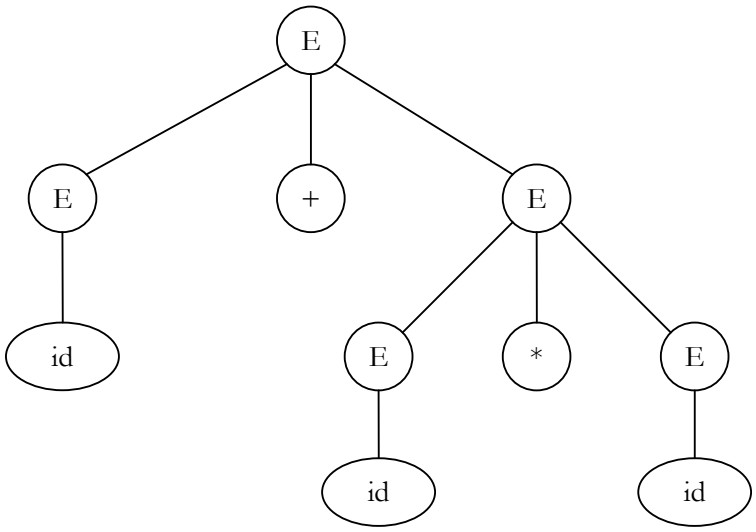


Figura 1.9

Cea de-a doua derivare are proprietatea că, la fiecare pas, s-a rescris cea mai din dreapta variabilă din forma propoziţională. Vom numi astfel de derivări *derivări extrem drepte* şi le vom nota prin $\underset{dr}{\Rightarrow}$:

$$\alpha Au \underset{dr}{\Rightarrow} \alpha\beta u \text{ dacă } u \in T^*, A \to \beta \in P, \alpha \in \Sigma^*$$

Prin urmare cea de-a doua derivare se scrie: $E \overset{+}{\underset{dr}{\Rightarrow}}$ `id+id*id`.

Aşadar, arborele sintactic este o reprezentare grafică a unei derivări în care a dispărut alegerea ordinii în care se aplică regulile de producţie. Unui arbore sintactic îi corespunde o singură derivare extrem stângă (respectiv o singură derivare extrem dreaptă). Se demonstrează relativ uşor următoarea lemă (vezi [Gri86] şi[Juc99]):

Lema 1.2.4. Fie G o gramatică şi $w \in T^*$. Atunci, există o derivare $S \Rightarrow^+ w$ (sau echivalent $w \in L(G)$) dacă şi numai dacă există un arbore de derivare cu frontiera w. Mai mult, dacă $A \in V$, $\gamma \in \Sigma^*$ are loc: $A \Rightarrow^+ \gamma$ dacă şi numai dacă există un arbore (construit după aceleaşi reguli ca arborele de derivare) în care rădăcina este etichetată cu A, iar frontiera este γ.

□

Să observăm acum că, pentru cuvântul w= id+id*id (gramatica din exemplul precedent), se mai poate construi un arbore de derivare distinct de cel de mai sus, anume cel din figura 1.10. Acestui arbore îi corespunde derivarea extrem stângă:

$S \Rightarrow E*E \Rightarrow E+E*E \Rightarrow$ `id+E*E` $\Rightarrow$ `id+id*E` $\Rightarrow$ `id+id*id`

care diferă de derivarea extrem stângă precedentă. Cele două derivări extrem stângi (respectiv cei doi arbori de derivare) corespund aceluiaşi cuvânt din L(G). Fenomenul este cunoscut sub numele de *ambiguitate*.

Definiţia 1.2.13 O gramatică G=(V, T, S, P) se zice că este *ambiguă* dacă există $w \in L(G)$ pentru care se pot construi măcar două derivări extrem stângi distincte (respectiv doi arbori de derivare distincţi).

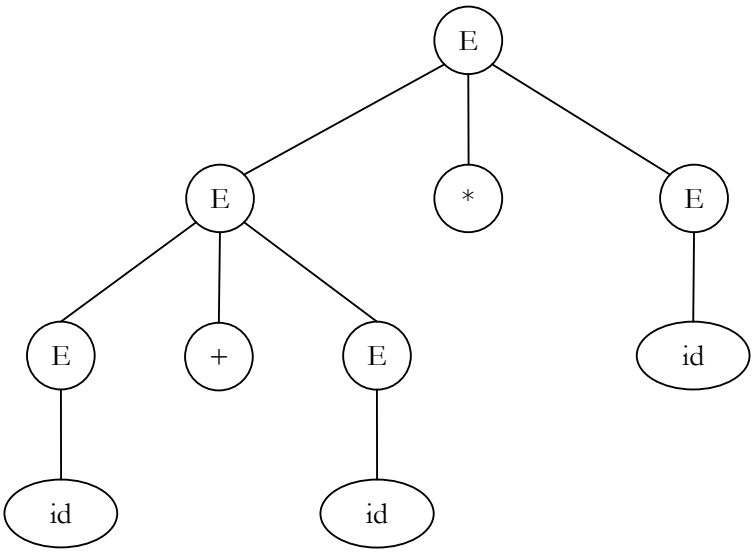


Figura 1.10

Exemplul 1.2.10 Gramatica următoare ce descrie construcţiile `if-then` şi `if-then-else` este ambiguă:

I → if e then I | if e then I else I | a

În adevăr, pentru fraza:

w = if e then if e then a else a

putem construi două derivări extrem stângi distincte:

I ⇒ if e then I ⇒ if e then if e then I else I ⇒
⇒ if e then if e then a else I ⇒
⇒ if e then if e then a else a

I ⇒ if e then I else I ⇒ if e then if e then I else I ⇒
⇒ if e then if e then a else I ⇒
⇒ if e then if e then a else a

care corespund, evident, la arbori de derivare distincţi.

Este de dorit ca, atunci când abordăm analiza sintactică, să o facem pentru o specificare neambiguă. Uneori, o gramatică ambiguă se poate transforma într-o gramatică echivalentă neambiguă. Această transformare este legată de anumite reguli care se impun. De pildă, în gramatica ambiguă:

E →E+E | E*E | (E) | id

nu este specificată nici o ordine de prioritate între operatorii + şi *. Dacă impunem ca * să aibă prioritate asupra lui +, atunci o nouă gramatică se poate scrie pentru descrierea expresiilor aritmetice:

E → E+T | T
T → T*F | F
F → (E) | id

În această nouă gramatică, noii neterminali au semnificaţia naturală: T notează ceea ce se cheamă termen, F este notaţia pentru factor, încât putem spune că de data aceasta gramatica exprimă mai fidel ceea ce se numeşte expresie. Dacă încercăm să impunem reguli similare pentru cazul if-then-else, putem ajunge la o exprimare de forma:

$I \rightarrow I_1 \mid I_2$
$I_1 \rightarrow$ if e then I_1 else $I_1 \mid$ a
$I_2 \rightarrow$ if e then $I \mid$ if e then I_1 else I_2

Prin aceasta s-a exprimat regula: se asociază else acelui then precedent, cel mai apropiat care nu are corespondent.

În capitolele următoare vom construi o serie de algoritmi. Pentru descrierea acestora vom folosi un limbaj(algoritmic) bazat pe limbajul C: vom utiliza operatori specifici limbajului C ca ++(incrementare), ==(egalitate logică), !=(neegalitate logică), !(negaţie) blocurile vor fi specificate prin acolade, instrucţiunile while, for, if care vor fi folosite funcţioneaza ca în C.

2 Analiza lexicală

Rolul pe care îl are un analizor lexical este de a citi textul sursă, caracter cu caracter, şi a-l transforma într-o secvenţă de unităţi primitive (elementare) numite unităţi lexicale (tokens în limba engleză). Fiecare unitate lexicală descrie o secvenţă de caractere cu o anumită semnificaţie şi este tratată ca o entitate logică. Pentru fiecare limbaj de programare se stabilesc (atunci când se proiectează limbajul) unităţile lexicale. În majoritatea limbajelor se disting următoarele unităţi lexicale:

Nr. Crt.	Unitatea Lexicală	Exemple
1	Constanta	`573 -5.89 2e+3`
2	Identificator	`alpha un_ident`
3	Operator	`+ * / < >`
4	Cuvânt rezervat	`begin while`
5	Semn special	`; . :`

În acest capitol vom discuta tehnicile de analiză lexicală, proiectarea şi implementarea unui analizor lexical. Problema analizei lexicale prezintă cel puţin două aspecte. Mai întâi, trebuie găsită o modalitate de descriere a unităţilor lexicale. Se constată că expresiile regulate – modelul algebric de descriere a limbajelor regulate - sunt mecanismele care pot descrie orice unitate lexicală. Un al doilea aspect este cel al recunoaşterii acestor unităţi lexicale (analiza lexicală propriu-zisă). Dacă descrierea se face cu expresii regulate atunci mecanismul de recunoaştere este automatul finit determinist. Se ştie din teoria limbajelor formale că mulţimea limbajelor descrise de expresii regulate coincide cu cea a limbajelor recunoscute de automate finite (clasa limbajelor regulate). Vom descrie în acest capitol algoritmii (eficienţi) de construcţie a unui automat finit (determinist) echivalent cu o expresie regulată şi, apoi, implementarea unui automat finit (analizorul lexical).
Unităţile lexicale sunt de două categorii:

- unităţi care descriu un şir de caractere anume (de exemplu `if, while, ++, :=, ;`);
- unităţi care descriu o clasă de şiruri: (identificatori, constante etc.).

În cel din urmă caz, vom trata o unitate lexicală ca fiind o pereche formată din *tipul* şi *valoarea unităţii lexicale*. Pentru unităţile lexicale care descriu un şir anume, convenim că tipul este acel şir, iar valoarea coincide cu tipul. De exemplu caracterul „(" este de tip paranteză stângă iar `alpha` este unitatea lexicală de tip *identificator* care are valoarea `alpha`. În literatura de specialitate `alpha` se mai numeşte instanţă a tokenului identificator sau *lexem*.
Vom descrie unităţile lexicale prin expresii regulate. De pildă, o constantă întreagă fără semn este descrisă de expresia:

(1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(0 | 1 | 2 |3 | 4 | 5 | 6 | 7 | 8 | 9)*

Pentru comoditatea notaţiilor, se pot da nume unor expresii regulate iar cu ajutorul acestora pot fi definite alte expresii regulate. Fie Σ un alfabet, d_1, d_2, …, d_n nume distincte date unor expresii regulate iar E_1, E_2,…, E_n expresii regulate astfel încât E_1 este expresie regulată peste Σ iar E_i ($1 < i \le n$) este expresie regulată peste alfabetul $\Sigma \cup \{d_1, d_2,\ldots, d_{i-1}\}$,. Atunci o definiţie regulată este un şir de definiţii de forma:

$d_1 \rightarrow E_1$
$d_2 \rightarrow E_2$
…
$d_n \rightarrow E_n$

De exemplu, mulţimea identificatorilor PASCAL poate fi dată prin definiţia regulată următoare (din comoditate s-a scris … dar trebuiesc înţelese caracterele corespunzătoare):

<literă> → A | B | … | Z | a | b | … | z
<cifră> → 0 | 1 |… | 9
<identificator> → <literă> (<literă> | <cifră>)*

Constantele fără semn în PASCAL se definesc astfel:

<cifră> → 0 | 1 |… | 9
<cifre> → <cifră><cifră>*
<fracţie> → .(<cifre> | ε)
<exponent> → (E | e)(+ | - | ε)<cifre>
<număr> → <cifre> | <cifre> <fracţie><exponent>

Operatorul unar * semnifică repetarea de zero sau mai multe ori a unei instanţe. Dacă folosim operatorul unar + pentru a desemna operaţia „cel puţin o instanţă din", adică $E^+ = EE^*$, atunci definiţia numelui <cifre> se poate scrie:

<cifre> → <cifră>$^+$

2.1 Un algoritm de trecere de la o expresie regulată la automatul echivalent

Să transpunem construcţiile din teorema 1.2.1 într-un algoritm pentru transformarea unei expresii regulate în automat finit. Mai întâi să observăm că fiecare din apariţiile operatorilor | şi * dintr-o expresie regulată E introduce două noi stări în automatul construit, pe când operatorul • nu introduce alte stări (figura 1.8). De asemenea, pentru orice apariţie a unui simbol din Σ, cât şi pentru ε, dacă acesta apare explicit în E, este nevoie de 2 stări în automatul construit. Aşadar, dacă n este numărul de simboluri din E iar m este numărul de paranteze împreună cu apariţiile simbolului •, atunci numărul stărilor automatului echivalent cu E este 2(n – m).

Să mai observăm că, din orice stare a automatului, se fac cel mult două tranziţii: fie o tranziţie cu un simbol din Σ, fie una sau două ε - tranziţii, fie zero tranziţii. Atunci, reprezentarea automatului echivalent cu o expresie regulată se poate face cu un tablou de dimensiune p×3 unde p este numărul stărilor (acestea sunt numerotate de la 1 la p), prima coloană conţine simbolurile cu care se fac tranziţiile iar următoarele două coloane conţin stările rezultate în urma tranziţiilor. Pentru descrierea algoritmului o să identificăm cele trei coloane prin vectorii simbol, next1, next2.

Algoritmul pe care-l descriem mai jos este datorat lui Rytter (Ryt91).

Algoritmul 2.1.1

Intrare: Expresia regulată E cu n simboluri dintre care m sunt paranteze şi apariţii ale operatorului produs;

Ieşire: Vectorii simbol, next1, next2 de dimensiune p = 2(n-m) ce descriu automatul cu ε - tranziţii echivalent cu E;

Metoda:

1. Se construieşte arborele ataşat expresie E (o metodă se dă mai jos);

2. Se parcurge arborele în preordine şi se ataşează nodurilor vizitate, exceptând pe cele etichetate cu •, respectiv numerele 1, 2, …, n-m;
3. Se parcurge arborele în postordine şi se ataşează fiecărui nod N o pereche de numere (i, f) care reprezintă starea iniţială respectiv finală a automatului corespunzător subarborelui cu rădăcina N, astfel:
   3.1. Dacă nodul are numărul k (de la pasul 2) atunci N.i = 2k-1, N.f = 2k;
   3.2. Dacă nodul este etichetat • atunci N.i = S.i iar N.f = D.f (S şi D sunt fii lui N, stâng respectiv drept);
4. for(1<= j <=2(n - m)) {simbol[j] = ‘ ’ , next1[j] = next2[j] = 0}
5. Se parcurge din nou arborele obţinut. Dacă N este nodul curent iar S şi D sunt fii săi, atunci, în funcţie de eticheta lui N, se execută următoarele:
   5.1. Dacă N este etichetat cu |:
   next1[N.i] = S.i, next2[N.i] = D.i, next1[S.f] = N.f, next1[D.f] = N.f
   5.2. Dacă N este etichetat cu • atunci next1[S.f] = D.i
   5.3. Dacă N este etichetat cu * (D nu există în acest caz):
   next1[N.i] = S.i, next2[N.i] = N.f, next1[S.f] = S.i, next2[S.f] = N.f
   5.4. Dacă N este etichetat cu a (deci este frunză):
   simbol[N.i] = ‘a’, next1[N.i] = N.f

Construcţia arborelui:

Intrare: Expresia regulată E = $e_0e_1…e_{n-1}$
Precedenţa operatorilor: prec(|) = 1, prec(•) = 2, prec(*) = 3.

Ieşire: Arborele asociat t.

Metoda: Se consideră două stive: STIVA1 – stiva operatorilor, STIVA2 – stiva operanzilor (care va conţine arborii parţiali construiţi).

```
1. i = 0;
2. while(i < n){
3.    c = e_i;
4.    switch(c){
5.       case '(' : {STIVA1.push(c); break;}
6.       case operand : {STIVA2.push(c); break;}
7.       case ')' : {
8.          do{build_tree();}while(STIVA1.top()!= ')');
9.          STIVA1.pop(); break;
         }//endcase
10.      case operator: {
11.         while(prec(STIVA1.top())>=prec(c))build_tree();
12.               STIVA1.push(c); break
         }//endcase
      }//endswitch
   }//endwhile
13.   while(STIVA1!= Φ ) build_tree();
14.   t = STIVA2.pop();
build_tree(){
      op = STIVA1.pop();
      D = STIVA2.pop();
      switch(op){
         case '*': {
            t = tree(op, D, NULL);
            STIVA2.push(t); break;
         }
         case '|': case'•':{
            S = STIVA2.pop();
            t = tree(op, S, D);
            STIVA2.push(t); break;
         }
}
```

Exemplul 2.1.1 Să considerăm expresia E = a*b | bb(a | c)*.

1. Arborele expresiei este prezentat în figura 2.1.
2. Nodurilor arborelui le sunt ataşate respectiv numerele 1, 2, …, 10 prin parcurgerea în preordine şi exceptând nodurile produs.
3. Numărul stărilor automatului va fi p = 20. După parcurgera în postordine a arborelui, fiecare nod are ataşat o pereche (i, j) (figura 2.2).
4. Se iniţializează vectorii simbol, next1 şi next2.
5. Tabela de tranziţie a automatului, în urma aplicării procedurilor descrise la 5 în fiecare nod al arborelui, este dată în continuare.

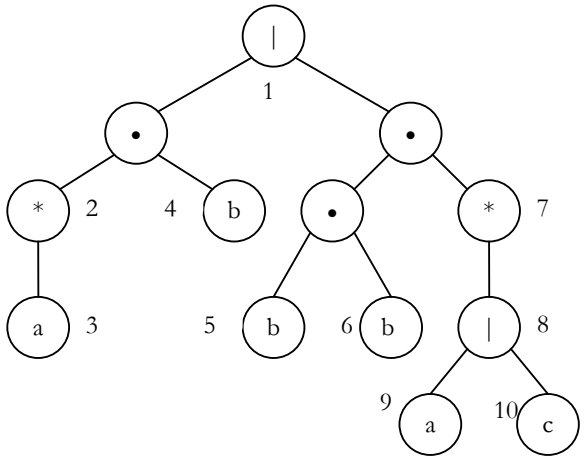


Figura 2.1

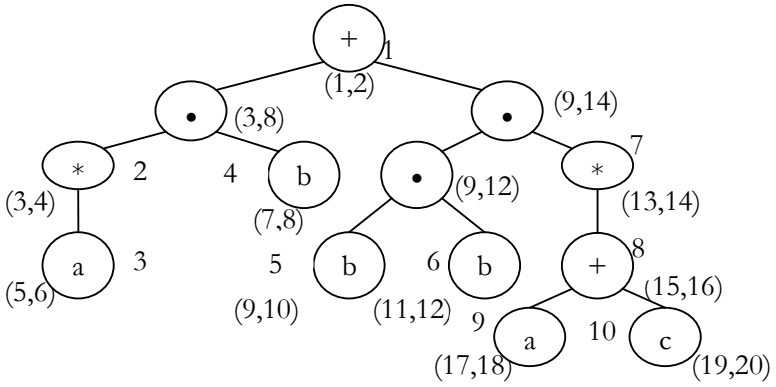


Figura 2.2

Tabela de tranziţie a automatului:

p	simbol[p]	next1[p]	next2[p]
1		3	9
2		0	0
3		5	4
4		7	0
5	a	6	0
6		5	4
7	b	8	0
8		2	0
9	b	10	0

10		11	0
11	b	12	0
12		13	0
13		15	14
14		2	0
15		17	19
16		15	14
17	a	18	0
18		16	0
19	c	20	0
20		16	0

Teorema 2.1.1 Algoritmul 2.1.1 este corect: automatul cu ε - tranziţii obţinut este echivalent cu expresia regulată E.

Demonstraţie Să observăm că modul în care au fost alese perechile (i, f) de stări pentru fiecare nod al arborelui construit corespunde construcţiilor din teorema 1.2.1. De asemenea tranziţiile care se definesc în pasul 5 al algoritmului urmăresc construcţia din teorema amintită. Aşadar, automatul obţinut este echivalent cu expresia dată la intrare.

□

2.2 De la un automat cu ε-tranziţii la automatul determinist echivalent

În paragraful precedent am descris procedeul de trecere de la o expresie regulată la automatul finit echivalent. Programul care descrie activitatea acestui automat finit pentru un cuvânt de intrare este de fapt analizorul lexical. Pentru ca acest analizor să fie eficient este de dorit ca automatul obţinut să fie determinist. Cum prin procedeul descris în teorema precedentă se obţine un automat cu ε - tranziţii (nedeterminist deci), să indicăm o modalitate de trecere de la un automat cu ε - tranziţii la automatul determinist echivalent. Mai întâi să descriem un algoritm pentru calculul mulţimii Cl(S) = $\hat{\delta}$ (S, ε).

Algoritmul 2.2.1

Intrare: Automatul (cu ε - tranziţii) A = (Q, Σ, δ, q_0, F), S ⊆ Q.

Ieşire: Cl(S) = $\hat{\delta}$ (S, ε).

Metoda: Stiva STIVA se iniţializează cu S şi pentru fiecare q din top, stările din δ(q, ε) ce nu au fost puse în Cl(S) se adaugă în Cl(S) şi în stivă.

```
1. R = S; STIVA = S;
2. while (STIVA ≠ Φ) {
3.     q = STIVA.pop(); // Se extrage q din stivă
4.     T = δ(q, ε) ;
5.     if(T ≠ Φ) {
6.           for ( p ∈ T ) {
7.                 if ( p ∉ R) {
8.                       R = R ∪ {p} ;
9.                       STIVA.push(p);//Se adaugă p in stiva
           }//endif
           }//endfor
     }//endif
   }//endwhile
10.   Cl(S) = R;
```

Algoritmul 2.2.2 Transformarea unui automat cu ε - tranziţii în automat finit determinist.

Intrare: Automatul (cu ε - tranziţii) A = (Q, Σ, δ, q_0, F).
Procedura de calcul pentru Cl(S) (Algoritmul 2.2.1).

Ieşire: Automatul determinist A' = (Q', Σ, δ', q_0', F'), echivalent cu A.

Metoda: Se construiesc stările lui A' începând cu q_0' şi continuând cu cele accesibile prin tranziţii cu simboluri din alfabet.

```
1. q0' = Cl(q0); Q' = {q0'} ;
2. marcat(q0') = false; F' = Φ ;
3. if ( q0 ∩ F ≠ Φ )  then F' = F' ∪ {q0'} ;
4. while (∃q' ∈ Q' && !marcat(q')){//q' este nemarcat
5.    for(a ∈ Σ ){
6.          p' = Cl(δ( q', a)); // = δ̂ (q', a)
7.          if ( p' ≠ Φ) {
8.                if ( p' ∉ Q') {
9.                      Q' = Q' ∪ {p'};
10.                     marcat(p') = false;
11.                     δ'(q',a) = p' ;
12.                     if(p' ∩ F != Φ)then F' = F' ∪ {p'};
                  }//endif
            }//endif
      }//endfor
13.   marcat(q') = true;
   }//endwhile
```

2.3 Proiectarea unui analizor lexical

Definiţia 2.4.1 Fie Σ un alfabet (al unui limbaj de programare). O *descriere lexicală* peste Σ este o expresie regulată E = $(E_1 \mid E_2 \mid \ldots \mid E_n)^+$, unde n este numărul unităţilor lexicale, iar E_i descrie o unitate lexicală, $1 \le i \le n$.

Exemplul 2.4.1 Să considerăm Σ = {a, b, …, y, z, 0, 1,…,9, :, =} şi următoarele expresii regulate:

Litera → a | b | c |… | z
Cifra → 0 | 1 | … | 9
Id → Litera (Litera | Cifra)*
Intreg → Cifră$^+$
Asignare → :=
Egal → =
Douapuncte → :

O descriere lexicală care cuprinde aceste unităţi lexicale este:

E = (Id | Intreg | Asignare | Egal | Douapuncte)$^+$

Definiţia 2.4.2 Fie E o descriere lexicală peste Σ ce conţine n unităţi lexicale şi w ∈ Σ^+. Cuvântul w este *corect relativ la descrierea* E dacă w ∈ L(E). O i*nterpretare* a cuvântului w ∈ L(E) este o secvenţă de perechi $(u_1, k_1), (u_2, k_2), \ldots, (u_m, k_m)$, unde $w = u_1u_2\ldots u_m$, $u_i \in L(E_{ki})$ $1 \le i \le m$, $1 \le k_i \le n$.

Uneori, în loc de k_i în (u_i, k_i), punem chiar E_{ki}, asta însemnând faptul că u_i este o unitate de tipul E_{ki}.

Exemplul 2.4.2 Fie E din Exemplul 2.4.1 şi w = `alpha:=beta=542`. O interpretare a cuvântului w este:

(`alpha`, Id), (`:=`, Asignare), (`beta`, Id), (`=`, Egal), (`542`, Intreg)

Sigur că aceasta nu este singura interpretare. Următoarea secvenţă este de asemenea o interpretare a aceluiaşi cuvânt: (`alpha`, Id), (`:`, Douapuncte), (`=`, Egal), (`b`, Id}), (`eta`, Id), (`=`, Egal), (`5`, Intreg), (`42`, Intreg).

Faptul că am obţinut interpretări diferite pentru acelaşi cuvânt este consecinţă a ambiguităţii expresiei regulate E: unitatea lexicală Asignare are două descrieri distincte în E : Asignare şi produsul DouapuncteEgal. La fel se întâmplă cu Id şi cu Intreg. Din teorie se ştie că, pentru orice expresie regulată E, există o expresie regulată E' neambiguă, echivalentă cu E. Soluţia de a construi E' pentru a obţine interpretări unice pentru fiecare cuvânt nu este convenabilă pentru că, pe de o parte, transformarea E în E' consumă timp iar, pe de altă parte, E' nu descrie în mod natural limbajul în cauză. În toate cazurile, se preferă să se dea nişte reguli în plus pentru a putea alege o singură interpretare a unui cuvânt atunci când descrierea lexicală este ambiguă.

Definiţia 2.4.3 Fie E o descriere lexicală peste Σ şi w ∈ L(E). O interpretare a cuvântului w, $(u_1, k_1)(u_2, k_2), \ldots(u_m, k_m)$, este *interpretare drept - orientată* dacă $(\forall i)\ 1 \le i \le m$, are loc:

$$|u_i| = \max\{|v| \ / v \in L(E_1 \mid E_2 \mid \ldots \mid E_n) \cap Pref(u_i u_{i+1} \ldots u_m)\}.$$

(unde prin Pref(w) am notat mulţimea prefixelor cuvântului w).

Observaţia 2.4.1 Există descrieri lexicale E în care nu orice cuvânt din L(E) admite o interpretare drept-orientată. Fie descrierea $E = (a \mid ab \mid bc)^+$ peste alfabetul Σ = {a, b} şi w = abc. Singura interpretare a cuvântului w este (a, 1), (bc, 3) dar aceasta nu este drept orientată pentru că a nu este cel mai lung cuvânt din L(a | ab | bc) ∩ Pref(abc), acesta din urmă fiind ab.

Definiţia 2.4.4 O descriere lexicală E este *bine - formată* dacă orice cuvânt w din limbajul L(E) are exact o interpretare drept-orientată.

Definiţia 2.4.5 Fie $E = (E_1 \mid E_2 \mid \ldots \mid E_n)^+$ o descriere lexicală bine formată peste Σ. Un *analizor lexical* (*scanner*) pentru E este un program ce recunoaşte limbajul L(E) şi produce, pentru fiecare w ∈ L(E), interpretarea sa drept-orientată.

Dată o descriere lexicală E peste alfabetul Σ, proiectarea unui analizor lexical cuprinde următoarele proceduri:

1. Se construieşte automatul finit (cu ε - tranziţii) A, astfel ca L(E) = L(A) (utilizând Algoritmul 2.1.1).
2. Se aplică Algoritmul 2.2.1 şi se obţine automatul determinist echivalent cu E, fie acesta A'.
3. (Opţional) Se aplică Algoritmul 2.3.1 pentru a obţine automatul minimal echivalent cu A'.
4. Se scrie un program care implementează evoluţia automatului obţinut.

Ne vom ocupa în continuare de modificările ce trebuiesc aduse construcţiilor 1-3 pentru a putea obţine o interpretare orientată dreapta pentru un cuvânt w din L(E). Distingerea între clasele de unităţi lexicale $E_1, E_2, \ldots, E_n$, poate fi făcută adăugând fiecărei expresii E_i o marcă de sfârşit $\#_i$, $1 \le i \le n$. Noua descriere lexicală, notată E# este:

$E\# = (E_1\#_1 \mid E_2\#_2 \mid \ldots \mid E_n\#_n)^+$, peste alfabetul $\Sigma \cup \{\#_1, \ldots, \#_n\}$.

Automatul determinist ce recunoaşte L(E#) va avea, desigur, tranziţii pentru simbolurile $\#_i$. Pentru că aceste simboluri nu apar în şirul de intrare (se analizează cuvintele $w \in \Sigma^+$), vom trata aceste tranziţii într-un mod special (în programul scris la etapa 4): când apare o tranziţie de tipul $\#_i$, programul raportează o unitate lexicală din clasa E_i.

Exemplul 2.4.3 Să considerăm descrierea lexicală:

litera → a | b |…|z cifra → 0 | 1 |…| 9

identificator → litera (litera | cifra)*

semn → + | - numar → (semn | ε) cifra$^+$

operator → + | - | * | / | < | > | <= | >= | < >

asignare → := doua_puncte → :

cuvinte_rezervate → if | then |else

paranteze →) | (

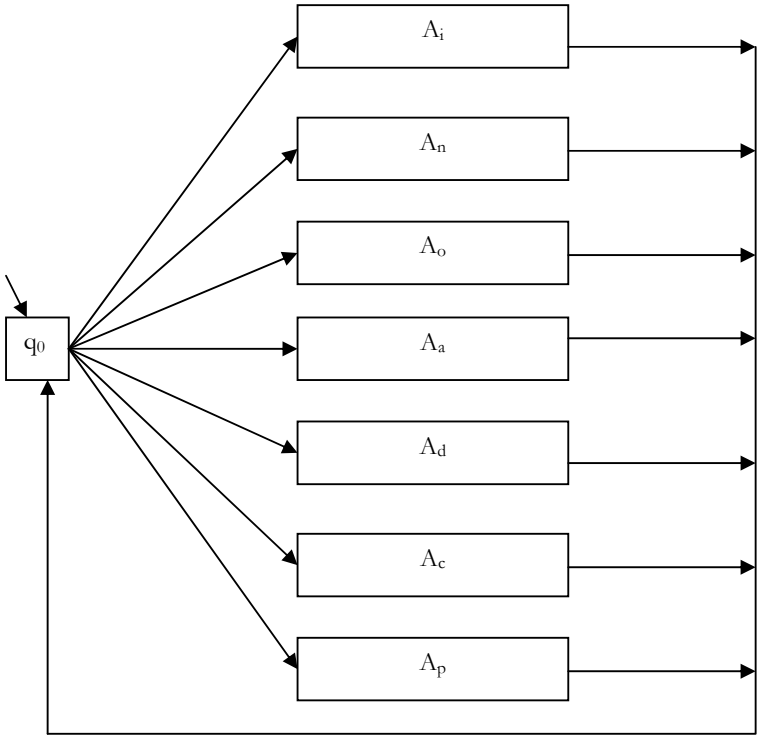


Figura 2.5

După cele discutate în paragraful precedent, există automatele (cu ε - tranziţii) A_i, A_n, A_o, A_a, A_d, A_c, A_p care recunosc respectiv limbajele descrise de expresiile regulate din E. Atunci, un automat cu ε - tranziţii ce recunoaşte E se construieşte după schema din figura 2.5.

Pentru acest automat cu ε - tranziţii, există un automat determinist echivalent care recunoaşte L(E). Acest automat determinist poate fi încă "simplificat" prin găsirea automatului cu cele mai puţine stări care recunoaşte acelaşi limbaj. În sfârşit, se scrie un program - analizorul lexical - care simulează acest automat. Automatul determinist (minimal) echivalent cu cel descris în fig. 2.5 şi cu tranziţiile corespunzătoare simbolurilor $\#_i$, $\#_n$, $\#_o$, $\#_a$, $\#_d$, $\#_c$, $\#_p$, care desemnează sfârşitul unei unităţi lexicale de tip *identificator*, *număr*, *operator*, *asignare*, *doua_puncte*, *cuvinte_rezervate*, *paranteze*, este dat în figura 2.6.

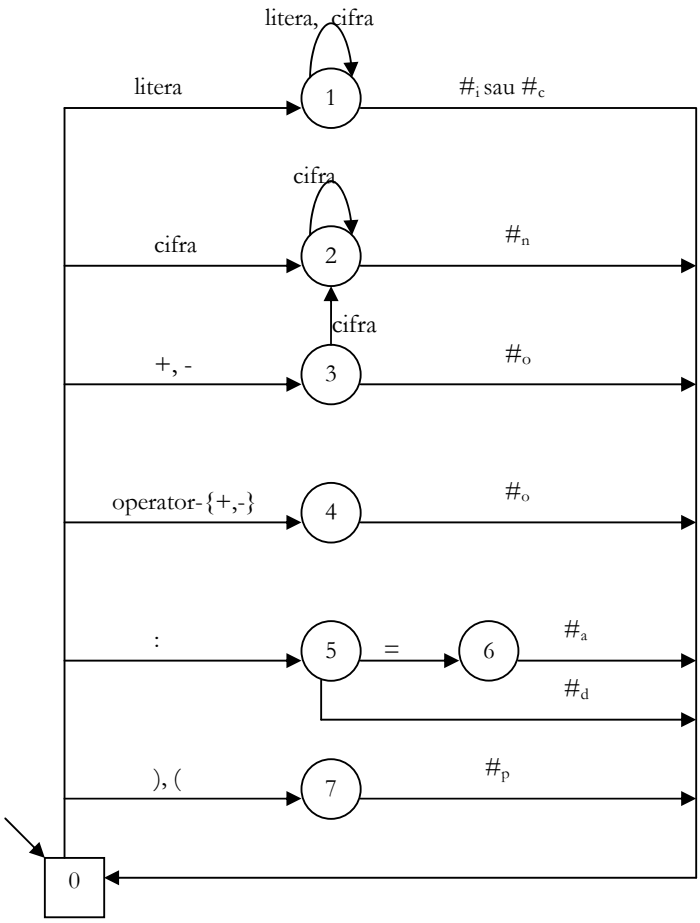


Figura 2.6

Aşadar, 0 este stare iniţială şi finală, tranziţiile notate cu *literă*, *cifră* înseamnă că sunt tranziţii pentru orice simbol din clasa *literă* respectiv *cifră*. Tranziţia cu un delimitator # semnifică faptul că s-a identificat o unitate lexicală; aceasta se raportează şi se reia activitatea automatului din starea iniţială.

Programul (analizorul lexical) constă din câte un segment de program pentru fiecare stare a automatului. Dacă vom nota prin `buffer` zona de memorie unde se încarcă o unitate lexicală, `getnext(ch), store(ch)` funcţiile ce citesc respectiv memorează `ch` în `buffer`, atunci pentru starea q care are tranziţii respectiv în stările q_i pentru simbolurile a_i, $1 \le i \le m$ şi tranziţia cu $\#_h$ în q_0, segmentul de program este dat mai jos.

```
q : switch(ch){
      case: a1
            store(ch);
            getnext(ch);
            goto q1;
      case: a2
            store(ch);
            getnext(ch);
            goto q2;
      .
      .
      case: am
            store(ch);
            getnext(ch);
            goto qm;
      default:
            write(buffer, i);
            empty(buffer);
            goto q0;
      }
```

Aşadar, în cazul în care caracterul citit este a_i, se memorează acesta, se citeşte alt caracter şi se continuă programul pentru starea q_i, $1 \le i \le m$, iar dacă nu este nici unul dintre simbolurile a_1, a_2, …, a_m, s-a depistat unitatea lexicală de tip i, se raportează aceasta (

`write(buffer,i)`), se goleşte zona `buffer` pentru a primi o nouă unitate lexicală după care se trece la programul stării iniţiale.

Dacă din starea q nu există # - tranziţie, grupul `default` se înlocuieşte prin:

```
default:
        write(buffer,"eroare");
        empty(buffer);
        goto q0;
```

Pentru starea iniţială q_0 (care este în acelaşi timp şi stare finală, grupul `default` se înlocuieşte prin:

```
default:
        if(EOF){
                write("Sfarsitul analizei");
                exit(0);
        }else{
                write(buffer,"eroare");
                empty(buffer);
                getnext(ch);
                goto qo;
        }
```

2.4 Generatorul Lex

LEX este un instrument software deosebit de util celor care doresc să implementeze limbaje de programare, dar poate fi folosit şi în multe alte aplicaţii. Acesta a fost dezvoltat la Bell Laboratories în anul 1975 de către M.E. Lesk şi E. Schmidt. LEX citeşte un fişier de intrare (în general cu extensia .l) care conţine expresii regulate ce definesc unităţile lexicale şi generează un program C care realizează analiza lexicală a unui text, în conformitate cu specificaţiile date.

LEX a devenit instrument standard UNIX începând cu versiunea a 7-a a acestui sistem de operare. Proiectul GNU al Fundaţiei "Free Software" distribuie FLEX (**F**ast **LEX**ical Analyzer Generator) care este o îmbunătăţire a generatorului LEX. De asemenea există versiuni pentru sistemele de operare DOS şi WINDOWS; una dintre acestea este PCLEX lansat de Abraxax Software Inc..

Pentru a utiliza LEX, se parcurg trei paşi:

1. Scrierea specificaţiilor LEX într-un fişier cu extensia .l care reprezintă descrierea lexicală pentru care dorim să construim un analizor lexical.
2. Executarea programului LEX (sau PCLEX) cu intrarea fişierul construit:

   lex [optiuni] <nume_fişier>
   flex [optiuni] <nume_fişier>
   pclex [optiuni] <nume_fişier>

Pentru a afla descrierea opţiunilor ce se pot folosi, executaţi programul lex –h.

Ca rezultat al execuţiei cu argumentul `<nume_fişier>` se obţine un fişier cu acelaşi nume dar cu extensia .c, care este de fapt un program în limbaj C.

3. Se compilează acest program împreună, eventual, cu alte fişiere sursă.

Programul C obţinut în urma executării LEX - ului este de fapt o funcţie cu numele `yylex()` care, pentru a fi executată, trebuie inclusă într-o funcţie `main()` (furnizată de utilizator) care conţine apelul `yylex()` sau, se integrează această rutină cu altele (de exemplu într-o aplicaţie `yacc`).

O specificare LEX (fişierul de intrare) are structura:

Declaraţii
%%
Reguli
%%
Rutine auxiliare

Secţiunea *Declaraţii* precum şi *Rutine auxiliare* pot lipsi încât o specificare minimă trebuie să conţină cele două linii %% între care sunt scrise liniile ce conţin reguli de specificare a unităţilor lexicale.

Secţiunea *Declaraţii* conţine două părţi. Prima parte constă din declaraţii C pentru variabilele sau constantele care se vor folosi ulterior iar a doua parte conţine definiţii ce se folosesc în specificaţiile LEX în secţiunea `Reguli`. O definiţie (macro pentru expresii regulate) are forma:

<nume> <expresie>

unde <expresie> este o expresie (regulată) ce poate folosi orice caracter împreună cu următorii operatori:

`" \ [ ] ^ - ? . * + | () $ / {} % <>`

Aceşti operatori au o anume semnificaţie iar dacă se doreşte a-i folosi drept caractere trebuiesc precedaţi de \ (backslash). Semnificaţia operatorilor este:

`"` Caracterele text sunt incluse între ghilimele "…". Construcţia `"c"` este echivalentă cu `\c`.

`[ ]` Defineşte o clasă de caractere sub forma unei liste alternative. În interior se poate folosi caracterul – pentru a indica un domeniu. De exemplu, o cifră întreagă se poate defini prin `[0123456789]` sau prin `[0-9]`. Expresiile următoare: `[-a+]`, `[+a-]`, `[a\-+]` sunt echivalente şi desemnează unul din caracterele `a`, `+` sau `-`. Dacă – se foloseşte în `[]` atunci el trebuie pus primul, ultimul sau sub forma `\-`.

`^` Acest simbol este considerat operator dacă este primul într-o anume descriere, altfel este caracter text. Se foloseşte pentru a exclude din definiţie anumite caractere. De exemplu expresia `[^ \t\n]` desemnează un caracter diferit de spaţiu, tab sau linie nouă.

`?` Elementul precedent acestui operator este opţional. Expresia `ab?c` desemnează `abc` sau `ac` iar `a[b-c]?d` notează `abd`, `acd` sau `ad`.

`*` Operatorul iteraţie de la expresii regulate.

`+` Operatorul iteraţie proprie (repetare de 1 sau mai multe ori) de la expresii regulate.

`|` Operatorul alternativă (reuniune) de la expresii regulate.

`()` Operatorul `()` (grupare) de la expresii regulate.

`{}` În macro-definiţii acest operator se poate folosi pentru a specifica numărul de repetiţii ale unei expresii. De pildă `AAA` şi `A{3}` sunt echivalente, iar dacă scriem [a-z]{1,5} aceasta desemnează una până la 5 litere mici ale alfabetului.

`.` Acest caracter (punct) desemnează orice caracter în afară de `\n`. De exemplu, descrierea `a.b` reprezintă `a0b`,`a1b`,`a\b`,`axb` etc. Într-o descriere lexicală ce urmează a fi inclusă într-un fişier intrare pentru lex, alternativa default se descrie printr-o linie ce are punctul în prima coloană, asta însemnând orice alt caracter ce nu a fost inclus până la această linie în vreo definiţie sau descriere.

Următoarele două linii în secţiunea declaraţii:

```
cifra [0-9]
litera [a-zA-Z]
```

sunt definiţii ale expresiilor regulate `cifra`, `litera` care pot fi folosite pentru definirea altor expresii în secţiunea Reguli. Numele definite aici trebuiesc folosite între acolade: `{litera}`, `{cifra}`.

Tot în această secţiune pot fi scrise segmente de cod C care urmează a fi incluse în procedura `yylex` (de pildă linii pentru preprocesor de tipul `#include` sau `#define`). Aceste elemente trebuiesc incorporate într-un bloc de forma `%{…%}`. De exemplu:

```
%{
#include <stdio.h>
%}
```

Secţiunea `Reguli` defineşte funcţionalitatea analizatorului lexical. Fiecare regulă este compusă dintr-o *expresie regulată* şi o *acţiune*. Aşadar, această secţiune are forma:

m_1 {Acţiune$_1$}
m_2 {Acţiune$_2$}
.
.
.
m_n {Acţiune$_n$}

unde mi sunt expresii regulate iar Acţiune$_i$ este un segment de program C care se va executa atunci când analizorul întâlneşte un exemplar din unitatea lexicală descrisă de m_i ($1 \le i \le n$). Pentru scrierea expresiilor regulate se folosesc operatorii pe care i-am descris mai sus, numele unor expresii deja definite, cuprinse între acolade precum şi orice alte caractere text. Spre exemplu, expresia `{litera}({litera}|{cifra})*` desemnează un identificator.

Secţiunea a treia, *Rutine auxiliare*, conţine toate procedurile auxiliare care ar putea fi utile în procesul de analiză lexicală. Dacă analizorul lexical creat este independent, această secţiune trebuie să conţină o funcţie `main()` care face apel la funcţia `yylex()`.

Exemplul 2.5.1 Iată conţinutul fişierului de intrare pentru descrierea lexicală din paragraful precedent:

```
%{
/* scaner1.l
// Exemplu de fisier de intrare pentru flex */
# include <stdio.h>
%}
litera [a-zA-Z]
cifra [0-9]
cifre ({cifra})+
semn [+-]
operator [+*/<>=-]
spatiu [' ''\t''\n']
%%
"if" | "then" | "else"        {
                  printf("%s  cuvant rezervat\n", yytext);}
({litera})({litera}|{cifra})* {
                  printf("%s  identificator\n", yytext);}
{cifre}|({semn})({cifre})           {
                  printf("%s  numar intreg\n", yytext);}
{operator}  {printf("%c  operator\n", yytext[0]);}
\:\=        {printf("%s  asignare\n", yytext);}
\:          {printf("%c  doua puncte\n", yytext[0]);}
(\()|(\))   {printf("%c  paranteza\n", yytext[0]);}
{spatiu}    {}
.           {printf("%c  caracter ilegal\n", yytext[0]);}
%%
int main( ){
      yylex( );
      return 0;
}
```

Exemplul 2.5.2 Următorul fişier constituie o descriere lexicală pentru un limbaj ce conţine cuvintele rezervate `set`, `sqrt`, `quit`, identificatori, numere şi operatori.

```
%{
/* scaner2.l
// Scaner expresii  */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
%}
%%
```

```
quit  {printf(" Cuvant cheie: %s\n", yytext);}
set   {printf(" Cuvant cheie: %s\n", yytext);}
sqrt  {printf(" Cuvant cheie: %s\n", yytext);}
\+    {printf(" Operator: %c\n", yytext[0]);}
\-    {printf(" Operator: %c\n", yytext[0]);}
\*    {printf(" Operator: %c\n", yytext[0]);}
\/    {printf(" Operator: %c\n", yytext[0]);}
\(    {printf(" Paranteza stanga: %c\n", yytext[0]);}
\)    {printf(" Paranteza dreapta: %c\n", yytext[0]);}
\:\=  {printf(" Asignare: %s\n", yytext);}
(([0-9]+(\.[0-9]*)?)|(\.[0-9]+))((e|E)(\-|\+)?[0-9]+)?  {
             printf(" Numar: %f\n", atof(yytext));}
[_A-Za-z]+    {printf(" Identificator: %s\n", yytext);}
[ \t]+            ; /* Eliminare spatii */
\n          {return 0;}
.           {printf(" Caracter ilegal: %c\n", yytext[0]);}
%%

int main(){
      printf("****************************************\n");
      printf("**            Exemplu de scaner        **\n");
      printf("**      Cuvinte cheie: set sqrt quit    **\n");
      printf("**      Operator: + - * / ( )           **\n");
      printf("**  Operanzi: numere, identificatori    **\n");
      printf("****************************************\n");
      yylex();
      return 0;
}
```

3 Analiza sintactică descendentă

La proiectarea limbajelor de programare se stabilesc reguli precise care descriu structura sintactică a programelor. În limbajul Pascal spre exemplu, un program este format din blocuri, un bloc conține instrucţiuni, o instrucţiune are în componenţa sa expresii, iar o expresie este o înşiruire de o anumită formă de unităţi lexicale.
Sintaxa unui limbaj de programare poate fi descrisă cu ajutorul gramaticilor independente de context sau a notaţiei BNF (Backus - Naur Form). Descrierea sintaxei prin gramatici independente de context are avantaje atât din punct de vedere al proiectării, cât şi din cel al implementării limbajelor. Aceste avantaje pot fi sintetizate astfel:

- o gramatică independentă de context oferă o specificare sintactică precisă şi în acelaşi timp uşor de înţeles;
- pentru anumite clase de gramatici este posibilă construirea automată a unui analizor sintactic eficace. Analizorul sintactic are rolul de a determina dacă un program sursă este corect din punct de vedere sintactic. Mai mult, în procesul de construcţie a analizorului sintactic pot fi depistate ambiguităţi sintactice sau alte construcţii dificile, care pot rămâne nedetectate în timpul proiectării unui limbaj;
- descrierea sintaxei printr-o gramatică independentă de context impune limbajului o structură care se dovedeşte a fi utilă pentru traducerea programului pusă în cod obiect corect, precum şi pentru detectarea erorilor.

3.1 Rolul analizorului sintactic

Un analizor sintactic primeşte la intrare un şir de unităţi lexicale şi produce arborele de derivare al acestui şir în gramatica ce descrie structura sintactică (mai degrabă o anume reprezentare a acestui arbore). În practică, odată cu verificarea corectitudinii sintactice, se produc şi alte acţiuni în timpul analizei sintactice: trecerea în tabela de simboluri a unor informaţii legate de unităţile sintactice, controlul tipurilor sau producerea codului intermediar şi alte chestiuni legate de analiza semantică (unele din acestea le vom discuta în capitolul ce tratează semantica). Să ne referim în câteva cuvinte la natura erorilor sintactice şi la strategiile generale de tratare a acestora.
Dacă un compilator ar trebui să trateze doar programele corecte, atunci proiectarea şi implementarea sa s-ar simplifica considerabil. Cum însă programatorii scriu adesea programe incorecte, un compilator bun trebuie să asiste programatorul în identificarea şi localizarea erorilor. Cea mai mare parte a specificărilor limbajelor de programare nu descriu modul în care implementarea (compilatorul) trebuie să reacţioneze la erori; acest lucru este lăsat pe seama celor care concep compilatoarele. Erorile într-un program pot fi:

- *lexicale*: scrierea eronată a unui identificator, a unui cuvânt cheie, a unui operator etc. ;
- *sintactice*: omiterea unor paranteze într-o expresie aritmetică, scrierea incorectă a unei instrucţiuni, etc. ;
- *semantice*: aplicarea unui operator aritmetic unor operanzi logici, nepotrivirea tipului la atribuiri etc. ;
- *logice*: un apel recursiv infinit.

3.2 Problema recunoaşterii. Problema parsării

Problema recunoaşterii în gramatici independente de context este următoarea: Dată o gramatică G = (V, T, S, P) şi un cuvânt w ∈ T*, care este răspunsul la întrebarea w ∈ L(G)? Se ştie că problema este decidabilă; mai mult, există algoritmi care în timp $O(|w|^3)$ dau răspunsul la întrebare (Cooke-Younger-Kasami, Earley, vezi [Gri86]). Problema *parsării* (*analizei sintactice*) este problema recunoaşterii la care se adaugă: dacă răspunsul la întrebarea w ∈ L(G) este afirmativ, se cere arborele sintactic (o reprezentare a sa) pentru w.

Fie G = (V, T, S, P) o gramatică şi w ∈ L(G). Să notăm prin $\underset{st}{\overset{p}{\Rightarrow}}$ ($\underset{dr}{\overset{q}{\Rightarrow}}$) relaţia de derivare extrem stângă (dreaptă) în care, pentru rescriere s-a aplicat producţia p ∈ P (q ∈ P). Atunci, pentru w ∈ L(G) există o derivare extrem stângă:

$$S = \alpha_0 \underset{st}{\overset{p1}{\Rightarrow}} \alpha_1 \underset{st}{\overset{p2}{\Rightarrow}} \alpha_2 \underset{st}{\overset{p3}{\Rightarrow}} \dots \underset{st}{\overset{pn}{\Rightarrow}} \alpha_n = w$$

şi o derivare extrem dreaptă:

$$S = \beta_0 \underset{dr}{\overset{q1}{\Rightarrow}} \beta_1 \underset{dr}{\overset{q2}{\Rightarrow}} \beta_2 \underset{dr}{\overset{q3}{\Rightarrow}} \dots \underset{dr}{\overset{qm}{\Rightarrow}} \beta_m = w$$

Atunci, arborele sintactic corespunzător se poate reprezenta prin secvenţa de producţii $p_1p_2\dots p_n \in P^+$ (respectiv $q_1q_2\dots q_m \in P^+$) care s-au aplicat, în această ordine, în derivarea extrem stângă (dreaptă) pentru obţinerea lui w.

Definiţia 3.2.1 Secvenţa de producţii $\pi = p_1p_2\dots p_n \in P^+$ pentru care $S \underset{st}{\overset{\pi}{\Rightarrow}} w$ se numeşte *analizare stângă* (*parsare stângă*) pentru cuvântul w ∈L(G).

Secvenţa de producţii $\pi = q_1q_2\ldots q_m \in P^+$ pentru care $S \underset{dr}{\overset{\tilde{\pi}}{\Rightarrow}} w$ se numeşte *analizare dreaptă* (*parsare dreaptă*) pentru cuvântul w (unde $\tilde{\pi} = q_mq_{m-1}\ldots q_1$).

Exemplul 3.2.4 Fie gramatica

1. $E \to E+T$ 2. $E \to T$ 3. $T \to T*F$,
4. $T \to F$ 5. $F \to (E)$ 6. $F \to id$

unde 1, 2, …, 6 identifică cele 6 producţii ale acesteia.

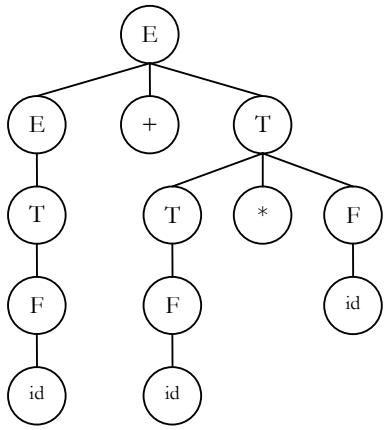


Figura 3.1

Pentru cuvântul w = `id+id*id`, arborele sintactic este dat în figura 3.1.
Analizarea stângă pentru acest cuvânt este: $\pi_1 = 12463466$ iar analizarea dreaptă este $\pi_2 = 64264631$. Aici am identificat fiecare producţie prin numărul său.

3.3 Analiza sintactică descendentă

Analiza sintactică descendentă (*parsarea descendentă*) poate fi considerată ca o tentativă de determinare a unei derivări extrem stângi pentru un cuvânt de intrare. În termenii arborilor sintactici, acest lucru înseamnă tentativa de construire a unui arbore sintactic pentru cuvântul de intrare, pornind de la rădăcină şi construind nodurile în manieră descendentă, în preordine (construirea rădacinii, a subarborelui stâng apoi a celui drept). Pentru realizarea acestui fapt avem nevoie de următoarea structură (figura 3.2):

- o *bandă de intrare* în care se introduce cuvântul de analizat, care se parcurge de la stânga la dreapta, simbol cu simbol;
- o *memorie de tip stivă* (pushdown) în care se obţin formele propoziţionale stângi (începând cu S). Prefixul formei propoziţionale format din terminali se compară cu simbolurile curente din banda de intrare obţinându-se astfel criteriul de înaintare în această bandă;
- o *bandă de ieşire* în care se înregistrează pe rând producţiile care s-au aplicat în derivarea extrem stângă care se construieşte.

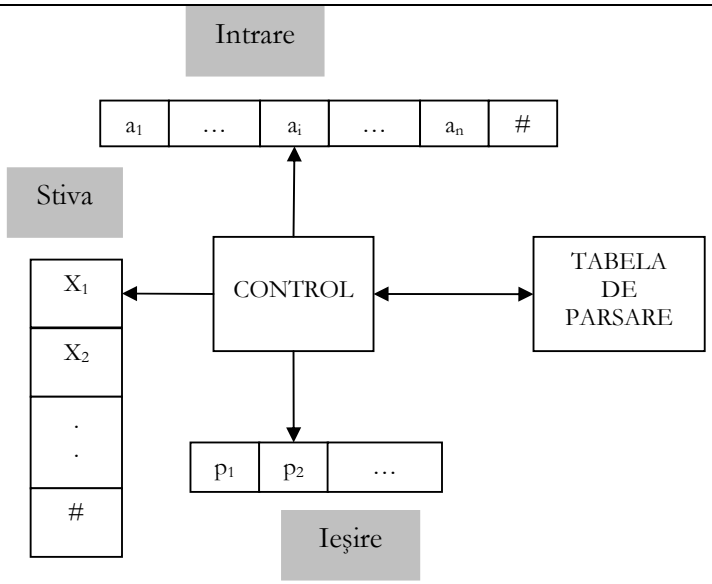


Figura 3.2

Criteriul de oprire cu succes este acela în care s-a parcurs întreaga bandă de intrare, iar memoria pushdown s-a golit. În acest caz în banda de ieşire s-a obţinut *parsarea stângă* a cuvântului respectiv. Iată un exemplu de cum funcţionează acest mecanism (considerăm gramatica din exemplul precedent şi cuvântul w = id+id*id). Vom considera caracterul # pentru marcarea sfârşitului benzii de intrare (marca de sfârşit a cuvântului) precum şi pentru a marca baza stivei.

Tabelul 1: Analiza sintactică descendentă pentru w = `id+id*id`

Pasul	Banda de intrare	Stiva de lucru	Banda de ieşire
1	id+id*id#	E#	
2	id+id*id#	E+T#	1
3	id+id*id#	T+T#	12
4	id+id*id#	F+T#	124
5	id+id*id#	id+T#	1246
6	id*id#	T#	1246
7	id*id#	T*F#	12463
8	id*id#	F*F#	124634
9	id*id#	id*F#	1246346
10	id#	F#	1246346
11	id#	id#	12463466
12	#	#	12463466

Se obţine la ieşire π = 12463466.

3.3.1 Parser descendent general

Fie G = (V, T, S, P) o gramatică care descrie sintaxa unui limbaj de programare. Să considerăm mulţimea $C = T^*\# \times \Sigma^*\# \times P^*$ numită *mulţimea configuraţiilor* (unui parser). Interpretarea pe care o dăm unei configuraţii este următoarea: dacă c = (u#, γ#, π), atunci u# este cuvântul rămas de analizat (conţinutul benzii de intrare la un moment dat), γ# este conţinutul stivei, cu primul simbol din γ în top-ul stivei (γ= $X_1X_2...X_n$ ca în figura 3.2), iar # la baza acesteia, iar π este conţinutul benzii de ieşire: dacă π = p_1p_2 … p_k, până în acest moment s-au aplicat producţiile din π în această ordine.

Definiţia 3.3.1 *Parserul* (*analizorul sintactic*) *descendent* ataşat gramaticii G = (V, T, S, P) este perechea ($\mathcal{C}_0$, $\vdash$) unde $\mathcal{C}_0$ = {(w#, S#, ε) | w ∈ T* } ⊆ C} este mulţimea configuraţiilor iniţiale, iar $\vdash \subseteq \mathcal{C} \times \mathcal{C}$ este relaţia de tranziţie între configuraţii dată de următoarea schemă:

1. (u#, Aγ#, π) $\vdash$ (u#, βγ#, πr), unde r = A → β ∈ P.
2. (uv#, uγ#, π) $\vdash$ (v#, γ#, π).
3. (#, #, π) este *configuraţie de acceptare* dacă π ≠ ε.
4. O configuraţie c pentru care nu există c' astfel ca c $\vdash$ c' (în sensul 1, 2) spunem că produce eroare.

Să considerăm relaţia $\vdash^+$ ($\vdash^*$) închiderea tranzitivă (şi reflexivă) a relaţiei $\vdash$ definită mai sus. Este clar că, dacă vom considera parserul ca un sistem de rescriere, acesta este nedeterminist aşa cum este definită tranziţia de tip 1: dacă în P există producţiile A → β_1 şi A → β_2 cu $\beta_1 \neq \beta_2$, atunci au loc, în acelaşi timp: (u#, Aγ#, π) $\vdash$ (u#, β_1γ#, π), (u#, Aγ#, πr_1) $\vdash$ (u#, β_2γ#, πr_2) unde r_1 = A → β_1, r_2 = A → β_2. Pentru a fi corect, parserul definit mai sus va trebui să transforme (în mod nedeterminist, în general) configuraţia (w#, S#, ε) în configuraţia (#, #, π) pentru cuvintele w ∈ L(G) şi numai pentru acestea! Vom demonstra în continuare corectitudinea în acest sens.

Lema 3.3.1 Dacă în parserul descendent ataşat gramaticii G = (V, T, S, P) are loc calculul (uv#, γ#, ε) $\vdash^*$ (v#, ψ#, π), atunci în gramatica G are loc derivarea γ $\underset{st}{\overset{\pi}{\Rightarrow}}$ uψ, oricare ar fi u, v ∈ T*, γ, ψ∈ Σ*, π ∈ P*.

Demonstraţie. Să demonstrăm afirmaţia din lemă prin inducţie asupra lungimii lui π (notată |π|):

- Dacă |π| = 0 înseamnă că în calculul considerat s-au folosit doar tranziţii de tip 2. Atunci γ = uψ şi are loc γ $\underset{st}{\overset{\pi}{\Rightarrow}}$ uψ, cu π = ε ;
- Dacă |π| > 0 fie π = π_1r, unde r = A → β este ultima producţie care se foloseşte în calculul considerat şi presupunem afirmaţia adevărată pentru calcule ce produc la ieşire π_1. Aşadar, putem scrie:
  (uv#, γ#, ε) = (u_1u_2v#, γ#, ε) $\vdash^*$ (u_2v#, Aγ_1#, π_1) $\vdash$ (u_2v#, βγ_1#, π_1r)=
  = (u_2v#, u_2ψ#, π_1r) $\vdash^*$ (v#, ψ#, π_1r)

Atunci din (u_1u_2v#, γ#, ε) $\vdash^*$ (u_2v#, Aγ_1#, π_1), conform ipotezei inductive, are loc γ $\underset{st}{\overset{\pi}{\Rightarrow}}$ u_1Aγ_1. Cum în u_1Aγ_1, u_1 ∈ T*, iar A ∈ V este cea mai din stânga variabilă, aplicând acesteia producţia A → β, obţinem:

γ $\underset{st}{\overset{\pi}{\Rightarrow}}$ u_1Aγ_1 $\underset{st}{\overset{r}{\Rightarrow}}$ u_1βγ_1 = u_1u_2ψ = uψ şi demonstraţia este încheiată.

□

Corolarul 3.3.1 Dacă în parserul descendent are loc (w#, S#, ε) $\vdash^+$ (#, #, π) atunci în gramatica G are loc S $\underset{st}{\overset{\pi}{\Rightarrow}}$ w.

Demonstraţie Se aplică lema precedentă pentru u = w, v = ε, γ = S, ψ = ε.

□

Lema 3.3.2 Dacă în gramatica G are loc derivarea γ $\underset{st}{\overset{\pi}{\Rightarrow}}$ uψ şi 1:ψ ∈V ∪{ε} (vezi definiţia 3.3.2), atunci în parserul descendent are loc calculul:

(uv#, γ#, ε) $\vdash^*$ (v#, ψ#, π), ∀v ∈ T*.

Demonstraţie Procedăm de asemenea prin inducţie după lungimea lui π :

- Dacă $|\pi| = 0$, atunci $\gamma = u\psi$ şi calculul are loc doar cu tranziţii de tipul al doilea: $(uv\#, \gamma\#, \varepsilon) = (uv\#, u\psi\#, \varepsilon) \vdash^* (v\#, \pi\#, \varepsilon)$.
- Dacă $|\pi| > 0$ fie $\pi = \pi_1 r$ unde $r = A \to \beta$ este ultima producţie aplicată în derivarea considerată: $\gamma \underset{st}{\overset{\pi 1}{\Rightarrow}} u_1\psi_1 = u_1 A\psi_2 \underset{st}{\overset{r}{\Rightarrow}} u_1\beta\psi_2 = u_1u_2\psi = u\psi$.

Am folosit aici efectiv ipoteza referitoare la primul simbol din ψ : primul simbol din ψ_1 este o variabilă (şi nu ε) pentru că se aplică efectiv un pas cu producţia r. Cum $|\pi_1| < |\pi|$, din $\gamma \underset{st}{\overset{\pi 1}{\Rightarrow}} u_1\psi_1$, conform ipotezei inductive are loc $(u_1v_1\#, \gamma\#, \varepsilon) \vdash^* (v_1\#, \psi_1\#, \pi_1)$ $\forall v_1 \in T^*$. Considerând $v_1 = u_2v$ cu $v \in T^*$ oarecare, putem scrie:

$$\begin{aligned}(u_1v_1\#, \gamma\#, \varepsilon) &= (u_1u_2v\#, \gamma\#, \varepsilon) &&\vdash^* (u_2v\#, \psi_1\#, \pi_1) &&= \\ &= (u_2v\#, A\psi_2\#, \pi_1) &&\vdash (u_2v\#, \beta\psi_2\#, \pi_1 r) &&= \\ &= (u_2v\#, u_2\psi\#, \pi) &&\vdash^* (v\#, \psi\#, \pi)\end{aligned}$$

□

Corolarul 3.3.2 Dacă în G are loc derivarea $S \underset{st}{\overset{\pi}{\Rightarrow}} w$ atunci în parserul descendent are loc calculul $(w\#, S\#, \varepsilon) \vdash^* (\#, \#, \pi)$.

Demonstraţie În lema 3.3.2 se consideră $u = w$, $v = \varepsilon$, $\gamma = S$ şi $\psi = \varepsilon$.

□

Teorema 3.3.1 (*Corectitudinea parserului descendent general*). Se consideră gramatica redusă $G = (V, T, S, P)$ şi $w \in T^*$. Atunci, în parserul descendent general ataşat acestei gramatici are loc $(w\#, S\#, \varepsilon) \vdash^* (\#, \#, \pi)$ (acceptare) dacă şi numai dacă $w \in L(G)$ şi π este o analizare sintactică stângă a frazei w.

Demonstraţie Teorema sintetizează rezultatele din lemele 3.3.1 şi 3.3.2.

□

3.3.2 Gramatici LL(k)

Parserul general introdus în paragraful precedent este un model nedeterminist iar implementarea sa pentru gramatici oarecare este ineficientă. Cea mai cunoscută clasă de gramatici pentru care acest model funcţionează determinist este aceea a gramaticilor LL(k): *Parsing from* **L***eft to right using* **L***eftmost derivation and* **k** *symbols lookahead*. Intuitiv, o gramatică este LL(k) dacă tranziţia de tip 1 din Definiţia 3.3.1 se face cu o unică regulă $A \to \beta$ determinată prin inspectarea a k simboluri care urmează a fi analizate în banda de intrare. Dar să definim în mod riguros această clasă de gramatici.

Definiţia 3.3.2 Fie $G = (V, T, S, P)$ o gramatică, $\alpha \in \Sigma^*$ şi $k \geq 1$ un număr natural. Atunci definim:

$k{:}\alpha$ = **if** $|\alpha| \leq k$ **then** α **else** α_1, unde $\alpha = \alpha_1\beta$, $|\alpha_1| = k$,

$\alpha{:}k$ = **if** $|\alpha| \leq k$ **then** α **else** α_2, unde $\alpha = \gamma\alpha_2$, $|\alpha_2| = k$.

Definiţia 3.3.3 O gramatică independentă de context redusă este gramatică LL(k), $k \geq 1$, dacă pentru orice două derivări de forma:

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\beta_1\gamma \underset{st}{\overset{*}{\Rightarrow}} ux$$

$$S \underset{st}{\overset{*}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\beta_2\gamma \underset{st}{\overset{*}{\Rightarrow}} uy$$

unde $u, x, y \in T^*$, pentru care $k{:}x = k{:}y$, are loc $\beta_1 = \beta_2$.

Definiţia prezentată exprimă faptul că, dacă în procesul analizei cuvântului w=ux a fost obţinut(analizat) deja prefixul u şi trebuie să aplicăm o producţie neterminalului A (având de ales pentru acesta măcar din două variante), această producţie este determinată în mod unic de următoarele k simboluri din cuvântul care a rămas de analizat (cuvântul x).

□

3.3.3 O caracterizare a gramaticilor LL(1)

Pentru ca un parser SLL(k) să poată fi implementat, trebuie să indicăm o procedură pentru calculul mulţimilor $FIRST_k$ şi $FOLLOW_k$. Pentru că în practică se foloseşte destul de rar (dacă nu chiar deloc) cazul $k \geq 2$, vom restrânge discuţia pentru cazul k = 1. Vom nota în acest caz $FIRST_1$ şi $FOLLOW_1$ prin FIRST respectiv FOLLOW. Aşadar, dacă $\alpha \in \Sigma^+$, $A \in V$:

$FIRST(\alpha) = \{a \mid a \in T, \alpha \overset{*}{\underset{st}{\Rightarrow}} au\} \cup$ if $(\alpha \overset{*}{\underset{st}{\Rightarrow}} \varepsilon)$ then $\{\varepsilon\}$ else Ø.

$FOLLOW(A) = \{a \mid a \in T \cup \{\varepsilon\}, S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma, a \in FIRST(\gamma)\}$

Mai întâi să arătăm că gramaticile SLL(1) coincid cu gramaticile LL(1).

Teorema 3.3.10 O gramatică G = (V, T, S, P) este gramatică LL(1) dacă şi numai dacă pentru orice $A \in V$ şi pentru orice producţii $A \to \beta_1 \mid \beta_2$ are loc:

$$FIRST(\beta_1 FOLLOW(A)) \cap FIRST(\beta_2 FOLLOW(A)) = \emptyset$$

Demonstraţie Să presupunem că G este LL(1). Acest lucru este echivalent cu(după Definiţia 3.3.3):

$$\left.\begin{array}{l} S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \overset{*}{\underset{st}{\Rightarrow}} u\beta_1\gamma \overset{*}{\underset{st}{\Rightarrow}} uv_1 \\ S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \overset{*}{\underset{st}{\Rightarrow}} u\beta_2\gamma \overset{*}{\underset{st}{\Rightarrow}} uv_2 \\ 1{:}v_1 = 1{:}v_2 \end{array}\right\} \Rightarrow \beta_1 = \beta_2$$

Să presupunem că G nu este SLL(1): există producţiile $A \to \alpha_1$, $A \to \alpha_2$ şi
$a \in FIRST(\alpha_1 FOLLOW(A)) \cap FIRST(\alpha_2 FOLLOW(A))$
Distingem următoarele cazuri:

- $a \in FIRST(\alpha_1) \cap FIRST(\alpha_2)$, adică $\alpha_1 \overset{*}{\underset{st}{\Rightarrow}} au_1$, $\alpha_2 \overset{*}{\underset{st}{\Rightarrow}} au_2$

Atunci, putem scrie derivările (G este redusă):

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_1v$$
$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_2v$$

ceea ce contrazice faptul că G este LL(1).

- $a \in FIRST(\alpha_1) \cap FOLLOW(A)$ şi $\alpha_2 \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$

Atunci: $\alpha_1 \overset{*}{\underset{st}{\Rightarrow}} au_1$ şi $S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma$, $a \in FIRST(\gamma)$ adică $\gamma \overset{*}{\underset{st}{\Rightarrow}} au_2$, şi putem scrie:

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_1v$$
$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \overset{*}{\underset{st}{\Rightarrow}} u\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_2$$

ceea ce contrazice de asemenea ipoteza.

- a ∈ FIRST (α_2) ∩ FOLLOW (A) şi $\alpha_1 \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$, caz ce se tratează analog cu precedentul.
- $\alpha_1 \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$, $\alpha_2 \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$ şi atunci a ∈ FOLLOW(A).

În acest caz avem:

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \overset{*}{\underset{st}{\Rightarrow}} u\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_2$$

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \overset{*}{\underset{st}{\Rightarrow}} u\gamma \overset{*}{\underset{st}{\Rightarrow}} uau_2$$

Deci din nou se contrazice proprietatea LL(1) pentru gramatica G.

Invers, să presupunem că pentru orice două producţii $A \to \beta_1$, $A \to \beta_2$ distincte are loc FIRST (β_1FOLLOW (A)) ∩ FIRST (β_2FOLLOW (A)) = Ø şi, prin reducere la absurd, G nu este LL(1). Asta înseamnă că există două derivări:

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_1\gamma \overset{*}{\underset{st}{\Rightarrow}} uav \text{ şi}$$

$$S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma \underset{st}{\Rightarrow} u\alpha_2\gamma \overset{*}{\underset{st}{\Rightarrow}} uav'$$

cu 1:v = 1:v' dar $\alpha_1 \neq \alpha_2$.

Analizând cele două derivări, se constată uşor că:

a ∈ FIRST (α_1FOLLOW (A)) ∩ FIRST (α_2FOLLOW (A))

ceea ce contrazice ipoteza.

□

3.3.4 Determinarea mulţimilor FIRST şi FOLLOW

Vom indica în acest paragraf modalitatea de determinare a mulţimilor FIRST şi FOLLOW pentru o gramatică G.

Un algoritm pentru determinarea mulţimilor FIRST(X) este descris mai jos. Cititorul este invitat să dovedească, pornind de la definiţia lui FIRST, că acest algoritm este corect.

Algoritmul 3.3.2

Intrare: Gramatica G = (V, T, S, P) redusă;

Ieşire: FIRST (X), X ∈ Σ.

Metoda: Mulţimile FIRST sunt completate prin inspectarea regulilor gramaticii

```
1. for (X ∈ Σ )
2.    if (X ∈ T) FIRST(X) = { X } else FIRST(X) = Φ;
3. for (A → aβ ∈ P)
4.    FIRST (A) = FIRST(A) ∪ {a };
5. FLAG = true;
6. while(FLAG) { // FLAG marcheaza schimbarile in FIRST
7.    FLAG = false;
8.    for(A → X1X2…Xn ∈ P) {
9.          i = 1;
10.         if((FIRST(X1)⊄ FIRST(A)){
11.            FIRST(A) = FIRST(A)∪(FIRST(X1);
12.            FLAG = true;
            }//endif
13.         while(i < n && Xi ⇒+st ε)
14.            if((FIRST(Xi+1)⊄ FIRST(A)){
```

```
15.              FIRST(A) = FIRST(A) ∪ FIRST(X_{i+1});
16.              FLAG = true; i++;
                }//endif
             }//endwhile
      }//endfor
   }//endwhile
17.   for (A ∈ V)
                 +
18.     if(A ⇒ ε) FIRST(A) = FIRST(A) ∪ { ε } ;
                 st
```

Pentru a verifica dacă o gramatică este LL(1), conform teoremei de caracterizare, este necesar să calculăm FIRST(α), unde $\alpha \in \Sigma^*$. Vom descrie în algoritmul următor acest lucru.

Algoritmul 3.3.3

Intrare: Gramatica G=(V, T, S, P).
Mulţimile FIRST(X), $X \in \Sigma$.
$\alpha = X_1X_2 \ldots X_n$, $X_i \in \Sigma$, $1 \le i \le n$.

Ieşire: FIRST(α).

Metoda:

```
1. FIRST(α) = FIRST(X_1) - { ε }; i =1;
                                +
2. while(i < n && X_i ⇒ ε) {
                                st
3.   FIRST(α) = FIRST(α) ∪ (FIRST(X_{i+1}) - { ε }) ;
4.   i = i+1 ;
   }//endwhile
                                +
5. if(i == n && X_n ⇒ ε)
                                st
6.   FIRST(α) = FIRST( α ) ∪ { ε } ;
```

Corectitudinea algoritmului descris rezultă din observaţia că, dacă pentru cuvântul $\alpha = X_1X_2 \ldots X_n$ are loc $X_1X_2 \ldots X_i \overset{+}{\underset{st}{\Rightarrow}} \varepsilon$, $i < n$ (adică $X_1, X_2, \ldots, X_i$ se pot şterge), atunci

$$(\bigcup_{k=1}^{i+1} \text{FIRST}(X_k) - \{ \varepsilon \}) \subseteq \text{FIRST}(\alpha).$$

Dacă în plus $\alpha \overset{+}{\underset{st}{\Rightarrow}} \varepsilon$, atunci FIRST(α) conţine şi ε.

Exemplul 3.3.1 Fie gramatica ce are următoarele reguli:

$E \to E+T \mid E-T \mid T$, $T \to T*F \mid T/F \mid F$, $F \to (E) \mid a$.

Aplicând algoritmul pentru calculul mulţimilor FIRST, obţinem:

FIRST(E) = FIRST(T) = FIRST(F) = {(, a },
FIRST(T*F) = FIRST(T) = {(, a }.

Exemplul 3.3.2 Fie gramatica:

$S \to E \mid B$ $\quad E \to \varepsilon$
$B \to a \mid$ `begin` SC `end` $\quad C \to \varepsilon \mid ;$ SC

FIRST(S) = {`a`, `begin`, ε } FIRST(E) = { ε }
FIRST(B) = {`a`, `begin` } FIRST(C) = {`;`, ε }.

Aplicând Algoritmul 3.3.3 se găseşte: FIRST(SEC) = {`a, begin, ;,` ε}, FIRST(SB)= {`a, begin` }, FIRST(;SC)= {`;`}.

Să trecem acum la determinarea mulţimilor FOLLOW(A), A ∈ V. Reamintind că FOLLOW(A) = {a ∈ T ∪{ ε } | $S \overset{*}{\underset{st}{\Rightarrow}} \alpha A\beta$, a ∈ FIRST(β)}, să facem câteva observaţii care vor fi utile în determinarea lui FOLLOW:

- ε ∈ FOLLOW(S) pentru că $S \overset{*}{\underset{st}{\Rightarrow}} S$.
- Dacă $A \rightarrow \alpha B\beta X\gamma \in P$ şi $\beta \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$, atunci FIRST(X)-{ε} ⊆ FOLLOW (B). În adevăr, putem scrie $S \overset{*}{\underset{st}{\Rightarrow}} \alpha_1 A\beta_1 \Rightarrow \alpha_1\alpha B\beta X\gamma\beta_1 \overset{*}{\underset{st}{\Rightarrow}} \alpha_1\alpha BX\gamma\beta_1$, şi atunci rezultă FIRST(X)-{ ε } ⊆ FOLLOW (B). Acest aspect, de fapt, se poate exprima mai simplu prin:
- Dacă $A \rightarrow \alpha B\beta \in P$ atunci FIRST(β)-{ ε } ⊆ FOLLOW (B).
- Dacă $A \rightarrow \alpha B\beta \in P$ şi $\beta \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$, atunci FOLLOW(A) ⊆ FOLLOW(B). În adevăr, dacă a ∈ FOLLOW(A) atunci există derivarea $S \overset{*}{\underset{st}{\Rightarrow}} \alpha_1 A\gamma$ şi a este din FIRST(γ). Din cele stipulate în ipoteză putem continua în această derivare:

$$S \overset{*}{\underset{st}{\Rightarrow}} \alpha_1 A\gamma \Rightarrow \alpha_1\alpha B\beta\gamma \overset{*}{\underset{st}{\Rightarrow}} \alpha_1\alpha B\gamma$$

şi deci a ∈ FOLLOW(B).

Ţinând cont de aceste observaţii rezultă următorul algoritm:

Algoritmul 3.3.4 (Determinarea mulţimilor FOLLOW)

Intrare: Gramatica G = (V, T, S, P) redusă ; Procedura FIRST(α), $\alpha \in \Sigma^+$.

Ieşire: Mulţimile FOLLOW (A), A ∈V.

Metoda:

```
1. for (A ∈ Σ) FOLLOW(A) = Φ;
2. FOLLOW(S)= { ε } ;
3. for (A → X1X2…Xn) {
4.   i = 1;
5.   while (i < n){
6.     while (Xi ∉ V) ++i;
7.     if (i < n) {// Xi este neterminal
8.       FOLLOW(Xi) = FOLLOW(Xi)∪(FIRST(Xi+1Xi+2…Xn)-{ε});
9.       ++i;
       }//endif
     }//endwhile
   }//endfor
10.FLAG = true;
11.while (FLAG) { // FLAG semnaleaza schimbarile in FOLLOW
12.  FLAG = false;
13.  for (A → X1X2…Xn) {
14.      i = n;
15.      while (i ≥ 1 && Xi ∈ V){
16.         if (FOLLOW (A) ⊄ FOLLOW(Xi) ){
17.            FOLLOW(Xi) = FOLLOW(Xi) ∪ FOLLOW (A);
18.            FLAG = true;
               }//endif
                    +
19.         if(Xi ⇒st ε)--i;// la Xi-1 daca Xi se sterge
20.         else continue; // la urmatoarea productie
          }//endwhile
     }//endfor
   }//endwhile
```

3.3.5 Tabela de analiză sintactică LL(1)

Pentru a implementa un analizor sintactic pentru gramatici LL(1), să considerăm o *tabelă de analiză*, sau *tabelă de parsare* LL(1):

$$M : V \times (T \cup \{ \# \}) \rightarrow \{(\beta, p) \mid p = A \rightarrow \beta \in P \} \cup \{eroare\}$$

construită după algoritmul următor:

Algoritmul 3.3.6 (Tabela de analiză sintactică)

Intrare: Gramatica G = (V, T, S, P);
Mulţimile FIRST(β), FOLLOW(A), A → β ∈ P.

Ieşire: Tabela de parsare M.

Metoda: Se parcurg regulile gramaticii şi se pun în tabelă

```
1.   for(A ∈ V )
2.      for( a ∈ T ∪ { # })
3.       M(A, a) = Φ ;
4.   for (p = A → β ∈ P) {
5.      for(a ∈ FIRST(β)-{ε})
6.       M(A, a) = M(A, a) ∪ {(β, p)} ;
7.      if(ε ∈ FIRST(B)){
8.       for(b ∈ FOLLOW(A)){
9.          if(b == ε)M(A, #) = M(A, #) ∪ {(β, p)};
10.         else  M(A, b) = M(A, b) ∪ {(β, p)};
           }//endfor
        }//endif
     }//endfor
11.  for(A ∈ V)
12.   for( a ∈ T ∪ { # })
13.  if(M(A, a) = Φ) M(A, a) = {eroare} ;
```

Lema 3.3.6 (caracterizarea gramaticilor LL(1)) Gramatica redusă G este LL(1) dacă şi numai dacă pentru orice A ∈ V şi orice a ∈ T ∪ { # } M(A, a), dacă nu este eroare, conţine cel mult o pereche (β, p).

Demonstraţie Dacă G este LL(1), atunci ∀A ∈ V, oricare ar fi două producţii cu partea stângă A, $A \rightarrow \beta_1$, $A \rightarrow \beta_2$, are loc, conform teoremei 3.3.10, $FIRST(\beta_1 FOLLOW(A)) \cap FIRST(\beta_2 FOLLOW(A)) = \Phi$. Dacă ar exista în tabela construită A ∈V şi a ∈V ∪ { # } astfel încât M(A, a) ⊇ {(β, p), (β', q)}, atunci, dacă a ∈ T rezultă

$$a \in FIRST(\beta FOLLOW(A)) \cap FIRST(\beta' FOLLOW(A))$$

ceea ce contrazice ipoteza. Dacă a = #, atunci ε ar fi comun în FIRST(βFOLLOW(A)) şi FIRST(β'FOLLOW(A)), din nou absurd. Deci M îndeplineşte condiţia enunţată. În mod analog se dovedeşte implicaţia inversă.

□

3.3.6 Analizorul sintactic LL(1)

Parserul (analizorul sintactic) LL(1) este un *parser top-down* în care tranziţiile sunt dictate de tabela de parsare. Aşadar, un astfel de parser are ca şi configuraţie iniţială tripleta (w#, S#, ε) unde w este cuvântul de analizat, iar tranziţiile se descriu astfel:

1. (u#, Aγ#, π) ⊢ (u#, βγ#, πr) dacă M(A, 1:u#) = (β, r). (operaţia *expandare*)
2. (uv#, uγ#, π) ⊢ (v#, γ#, π). (operaţia *potrivire*)
3. (#, #, π) ⊢ *acceptare* dacă π ≠ ε.
4. (au#, bγ#, π) ⊢ *eroare* dacă a ≠ b.
5. (u#, Aγ#, π) ⊢ *eroare* dacă M(A, 1:u#) = *eroare*.

Teorema de corectitudine a parserului descendent general împreună cu teorema de caracterizare LL(1), a modului în care s-a definit tabela M, dovedesc corectitudinea parserului LL(1). Suntem acum în măsură să indicăm modul de implementare al unui analizor sintactic LL(1). Presupunem că dispunem de o bandă de intrare (fişier de intrare) din care, cu o funcţie `getnext()`, se obţine caracterul (tokenul) următor. De asemenea dispunem de o stivă cu funcţiile specifice `pop()` şi `push()`. Producţiile care se aplică pe parcursul analizei le scriem într-o bandă (fişier) de ieşire cu funcţia write(). Atunci algoritmul de analiză sintactică se descrie astfel:

Algoritmul 3.3.6 (**Parser LL(1)**)

Intrare: Gramatica G = (V, T, S, P).
Tabela de analiză LL(1) notată M.
Cuvântul de intrare w#.

Ieşire: Analiza sintactică stângă π a lui w dacă w ∈ L(G),
eroare în caz contrar.

Metoda: Sunt implementate tranziţiile 1 – 5 folosind o stivă St.

```
1. St.push(#),St.push(S)// St = S#
2. a = getnext(), π = ε;
3. do {
4.    X = St.pop();
5.    if(X == a)
6.       if(X != '#') getnext();
7.       else{
8.          if (π != ε){write("acceptare"); exit(0);}
9.             else {write("eroare"); exit(1);}
               } //endelse
10.      else{
11.            if(X∈T){write("eroare"); exit(1);}
12.            else{
13.                  if(M(X,a) == "eroare")
                       {write("eroare"); exit(1);}
15.                  else {
                 // M(X,a)=(β,r), r=X → β, β=Y1Y2…Yn
                 //β inlocueste pe X in stiva
16.                  for(k = n; k>0; --k) push(Yk);
17.                  write(r); //se adauga r la π
                     } //endelse
               } //endelse
          } //endelse
   } while(1);
```

Exemplul 3.3.4 Să reluăm gramatica din exemplele precedente:

```
1. S → E          5. B → begin SC end
2. S → B          6. C → ε
3. E → ε          7. C → ;SC
4. B → a
```

Mulţimile FIRST şi FOLLOW sunt date în tabelul următor:

X	FIRST(X)	FOLLOW(X)
S	a begin ε	end ; ε
E	ε	end ; ε
B	a begin	end ; ε
C	; ε	end

Tabela de analiză LL(1) pentru această gramatică este dată mai jos:

M	a	begin	end	;	#
S	(B, 2)	(B, 2)	(E, 1)	(E, 1)	(E, 1)
E	eroare	eroare	(ε, 3)	(ε, 3)	(ε, 3)
B	(a, 4)	(begin SC end, 5)	eroare	eroare	eroare
C	eroare	eroare	(ε, 6)	(;SC, 7)	eroare

Se vede din tabelă că această gramatică este LL(1) (conform teoremei 3.3.13). Iată şi două exemple de analiză, unul pentru cuvântul `begin a;;a end` care este din limbajul generat de gramatica dată, iar altul pentru `begin aa end` care nu este corect.

PAS	INTRARE	STIVA	OPERAŢIE	IEŞIRE
1	`begin a;;a end#`	`S#`	*expandare*	
2	`begin a;;a end#`	`B#`	*expandare*	2
3	`begin a;;a end#`	`begin SC end#`	*potrivire*	5
4	`a;;a end#`	`SC end#`	*expandare*	
5	`a;;a end#`	`BC end#`	*expandare*	2
6	`a;;a end#`	`aC end#`	*potrivire*	4
7	`;;a end#`	`C end#`	*expandare*	
8	`;;a end#`	`;SC end#`	*potrivire*	7
9	`;a end#`	`SC end#`	*expandare*	
10	`;a end#`	`EC end#`	*expandare*	1
11	`;a end#`	`C end#`	*expandare*	3
12	`;a end#`	`;SC end#`	*potrivire*	7
13	`a end#`	`SC end#`	*expandare*	
14	`a end#`	`BC end#`	*expandare*	2
15	`a end#`	`aC end#`	*potrivire*	4
16	`end#`	`C end#`	*expandare*	
17	`end#`	`end#`	*potrivire*	6
18	`#`	`#`	*acceptare*	

PAS	INTRARE	STIVA	OPERAŢIE	IEŞIRE
1	`begin aa end#`	`S#`	*expandare*	
2	`begin aa end#`	`B#`	*expandare*	2
3	`begin aa end#`	`begin SC end#`	*potrivire*	5
4	`aa end#`	`SC end#`	*expandare*	
5	`aa end#`	`BC end#`	*expandare*	2
6	`aa end#`	`aC end#`	*potrivire*	4
7	`a end#`	`C end#`	*eroare*	

3.3.7 Eliminarea recursiei stângi

O gramatică stâng recursivă nu este LL(k) pentru nici un număr k. Este cazul, spre exemplu, al gramaticii care generează expresiile aritmetice:

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

Prin eliminarea recursiei stângi, există posibilitatea obţinerii unei gramatici din clasa LL. Acesta este un procedeu utilizat frecvent în proiectarea analizoarelor sintactice. Să reamintim că o gramatică G este stâng recursivă dacă există un neterminal A pentru care $A \overset{+}{\Rightarrow} A\alpha$. Spunem că un neterminal A este stâng recursiv imediat dacă există o producţie $A \to A\alpha$ în P. Să indicăm mai întâi un procedeu de eliminare a recursiei stângi imediate.

Lema 3.3.7 Fie G = (V, T, S, P) o gramatică pentru care A este stâng recursiv imediat. Să considerăm toate A – producţiile gramaticii, fie ele $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n$ cele cu recursie şi $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$ cele fără recursie imediată (părţile drepte nu încep cu A). Există o gramatică G' = (V', T, S, P') echivalentă cu G, în care A nu este stâng recursiv imediat.

Demonstraţie Să considerăm un nou neterminal $A' \notin V$ şi fie $V' = V \cup \{A\}$ iar P' se obţine din P astfel:

- Se elimină din P toate A -producţiile;
- Se adaugă la P următoarele producţii:
  $A \to \beta_i A' \qquad 1 \le i \le m.$
  $A' \to \alpha_j A' \mid \varepsilon \quad 1 \le j \le n.$
- Notăm cu P' mulţimea de producţii obţinută.

Să observăm că A nu mai este stâng recursiv în G'. Noul simbol neterminal introdus, notat cu A', este de data aceasta *drept recursiv*. Acest lucru însă nu este un inconvenient pentru analiza sintactică descendentă.

Să observăm că, pentru orice derivare extrem stânga în gramatica G, de forma:

$$A \Rightarrow A\alpha_{kn} \Rightarrow A\alpha_{kn-1}\alpha_{kn} \Rightarrow \ldots \Rightarrow A\alpha_{k1}\alpha_{k2}\ldots\alpha_{kn} \Rightarrow \beta_h\alpha_{k1}\alpha_{k2}\ldots\alpha_{kn}$$

există în gramatica G' o derivare extrem dreapta de forma:

$$A \Rightarrow \beta_h A' \Rightarrow \beta_h\alpha_{k1}A' \Rightarrow \ldots \Rightarrow \beta_h\alpha_{k1}\alpha_{k2}\ldots\alpha_{kn}A' \Rightarrow \beta_h\alpha_{k1}\alpha_{k2}\ldots\alpha_{kn}.$$

De asemenea, afirmaţia reciprocă este valabilă.

Aceste observaţii conduc la concluzia că cele două gramatici sunt echivalente.

□

Exemplul 3.3.5 În gramatica expresiilor aritmetice:

$E \to E+T \mid E-T \mid -T \mid T$
$T \to T*F \mid T/F \mid F$
$F \to (E) \mid a$

se fac transformările următoare:

- Producţiile $E \to E+T \mid E-T \mid -T \mid T$ se înlocuiesc cu:
  $E \to TE' \mid -TE',\ E' \to +T\,E' \mid -TE' \mid \varepsilon$
- Producţiile $T \to T*F \mid T/F \mid F$ se înlocuiesc cu:
  $T \to FT',\ T' \to *FT' \mid /FT' \mid \varepsilon$

Se obţine astfel gramatica echivalentă:

$E \to TE' \mid -TE'$
$E' \to +T\,E' \mid -TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid /FT' \mid \varepsilon$
$F \to (E) \mid a$

Aplicând algoritmii pentru determinarea mulţimilor FIRST şi FOLLOW se obţine:

X	FIRST(X)	FOLLOW(X)
E	(a -	ε)
E'	+ - ε	ε)
T	(a	+ - ε)
T'	* / ε	+ - ε)
F	(a	* / + - ε)

Tabela de parsare pentru această gramatică este dată mai jos.

M	a	+	-	*	/	(	)	#
E	(TE', 1)	*eroare*	(-TE', 2)	*eroare*	*eroare*	(TE', 1)	*eroare*	*eroare*
E'	*eroare*	(+TE',3)	(-TE', 4)	*eroare*	*eroare*	*eroare*	(ε, 5)	(ε, 5)
T	(FT', 6)	*eroare*	*eroare*	*eroare*	*eroare*	(FT', 6)	*eroare*	*eroare*
T'	*eroare*	(ε, 9)	(ε, 9)	(*FT',7)	(/FT',8)	*eroare*	(ε, 9)	(ε, 9)
F	(a, 11)	*eroare*	*eroare*	*eroare*	*eroare*	((E), 10)	*eroare*	*eroare*

Se observă de aici că gramatica este LL(1). Să urmărim analiza sintactică pentru expresia –(a+a)*a.

PASUL	INTRARE	STIVA	OPERAȚIE	IEȘIRE
1.	-(a+a)*a#	E#	*expandare*	
2.	-(a+a)*a#	-TE'#	*potrivire*	*4*
3.	(a+a)*a#	TE'#	*expandare*	
4.	(a+a)*a#	FT'E'#	*expandare*	*6*
5.	(a+a)*a#	(E)T'E'#	*potrivire*	*10*
6.	a+a)*a#	E)T'E'#	*expandare*	
7.	a+a)*a#	TE')T'E'#	*expandare*	*1*
8.	a+a)*a#	FT'E')T'E'#	*expandare*	*6*
9.	a+a)*a#	aT'E')T'E'#	*potrivire*	*11*
10.	+a)*a#	T'E')T'E'#	*expandare*	
11.	+a)*a#	E')T'E'#	*expandare*	*9*
12.	+a)*a#	+TE')T'E'#	*potrivire*	*3*
13.	a)*a#	TE')T'E'#	*expandare*	
14.	a)*a#	FT'E')T'E'#	*expandare*	*6*
15.	a)*a#	aT'E')T'E'#	*potrivire*	*11*
16.	)*a#	T'E')T'E'#	*expandare*	
17.	)*a#	E')T'E'#	*expandare*	*9*
18.	)*a#	)T'E'#	*potrivire*	*5*
19.	*a#	T'E'#	*expandare*	
20.	*a#	*FT'E'#	*potrivire*	*7*
21.	a#	FT'E'#	*expandare*	
22.	a#	aT'E'#	*potrivire*	*11*
23.	#	T'E'#	*expandare*	
24.	#	E'#	*expandare*	*9*
25.	#	#	*acceptare*	*5*

Să arătăm cum eliminăm în general recursia stângă într-o gramatică. Vom face ipoteza că gramatica G nu are ε - producţii şi nu are cicluri de forma $A \overset{+}{\Rightarrow} A$. Aceasta nu este de fapt o restricţie: este ştiut faptul că, pentru orice gramatică G există G' fără ε -producţii şi fără cicluri astfel încât L(G') = L(G) - { ε }. (vezi [Gri86], [Juc99]). Aşadar are loc următoarea teoremă:

Teorema 3.3.13 Pentru orice gramatică G fără ε - producţii şi fără cicluri, există o gramatică echivalentă G' care nu este stâng recursivă.

Demonstraţie Să considerăm următorul algoritm:

Intrare: Gramatica G = (V, T, S, P) fără ε-producţii, fără cicluri.

Ieşire: Gramatica G' fără recursie stângă astfel ca L(G)=L(G').

Metoda: După ordonarea neterminalilor se fac transformări ce nu schimbă limbajul gramaticii, inclusiv prin eliminarea recursiei imediate

```
1. Se consideră V = {A1, A2, …,An}
2. for (i=1; i ≤ n; ++i) {
3.       for (j=1; j ≤ i - 1; ++j) {
4.             for (Ai → Ajγ ∈ P) {
5.                   P = P – { Ai → Ajγ };
6.                   for (Aj→ β ∈ P ) P = P ∪ { Ai → βγ};
                   } //endfor
7.             Se elimina recursia imediata pentru Ai;          }
   //endfor
   } //endfor
8. V' = V; P' = P;// V s-a modificat la 7 iar P la 4 - 6
```

Să observăm că transformările din 4 - 7 nu modifică limbajul generat de gramatică încât L(G') = L(G). Să arătăm acum că G' nu are recursie stângă. Acest lucru rezultă din faptul că, după iteraţia i - 1 a buclei 2, fiecare producţie de forma $A_k \to A_l\alpha$ pentru k < i are proprietatea că l > k. Aceasta datorită transformărilor 4 - 6 şi eliminării recursiei stângi imediate în linia 7: după execuţia buclei 3 se obţin producţii $A_i \to A_m\alpha$ cu m ≥ i, iar dacă m = i, producţia $A_i \to A_i\alpha$ se elimină în linia 7. Aşadar, gramatica obţinută este fără recursie stângă.

□

Exemplul 3.3.6 Fie gramatica:

$$S \to Aa \mid b \qquad A \to Ac \mid Sd \mid a$$

Considerăm neterminalii în ordonaţi (S, A). Aplicând bucla 2 pentru i = 1 nu se face nici o modificare deoarece în producţia $S \to Aa$ ordinea neterminalilor este corectă. Pentru i = 2 trebuie modificată producţia $A \to Sd$ (pentru că S este *mai mic* decât A). A - producţiile se modifică astfel:

$$A \to Ac \mid Aad \mid bd \mid a$$

Gramatica, în acest moment, este

$$S \to Aa \mid b$$
$$A \to Ac \mid Aad \mid bd \mid a$$

şi conţine doar recursivitate imediată (A). Aplicând algoritmul de eliminare a recursiei stângi imediate (vezi Lema 3.3.7), obţinem:

$$S \to Aa \mid b \qquad A \to bdA' \mid aA' \qquad A' \to cA' \mid adA' \mid \varepsilon$$

Se poate observa că rezultatul nu este o gramatică LL(1) deoarece pentru primele două producţii FIRST(Aa) ∩ FIRST(a) = {b}. Aşadar eliminarea recursiei stângi nu conduce neapărat la proprietatea LL(1).

4 Analiza sintactică în gramatici LR

Vom prezenta în acest capitol o tehnică eficace de analiză sintactică ascendentă care este utilizată pentru o clasă largă de gramatici: gramaticile LR(k). Denumirea LR(k) vine de la: **L***eft to right scanning of the input, constructing a* **R***ightmost derivation in reverse, using* **k** *symbols lookahead*. Această metodă este cu siguranţă cea mai des utilizată metodă de analiză sintactică, din următoarele motive:

- se pot construi analizoare sintactice LR pentru recunoaşterea tuturor construcţiilor din limbajele de programare care se pot descrie printr-o gramatică independentă de context;
- clasa limbajelor ce pot fi analizate sintactic cu analizoare LR(1) coincide cu clasa limbajelor de tip 2 deterministe;
- metoda de analiză LR este o metodă de tip *deplasare-reducere* relativ uşor de implementat şi eficace în acelaşi timp;
- un analizor LR poate detecta o eroare de sintaxă cel mai rapid posibil parcurgând şirul de intrare de la stânga la dreapta.

Dezavantajul principal al metodei este acela că determinarea tabelei de analiză necesită un volum mare de muncă; există însă generatoare de analizoare de tip LR, precum *yacc* sau *bison*, care produc un astfel de analizor.

Vom considera în continuare o gramatică G = (V, T, S, P) redusă şi gramatica augmentată G' = (V', T', S', P') unde P' = P ∪ {S' → S }, V' = V ∪ { S' }, S' fiind un simbol nou. Gramatica G' este echivalentă cu G şi are proprietatea că simbolul de start nu apare în nici o parte dreaptă a producţiilor din P', condiţie esenţială pentru studiul gramaticilor LR(k). Să mai facem observaţia că, pentru gramaticile care au proprietatea amintită (simbolul de start nu apare în nici o parte dreaptă a producţiilor), nu este necesară augmentarea.

4.1 Gramatici LR(k)

Definiţia 5.1.1 O gramatică G se numeşte *gramatică LR(k)*, k ≥ 0, dacă pentru orice două derivări de forma:

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \alpha A u \underset{dr}{\Rightarrow} \alpha\beta u = \gamma u$$

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \alpha' A' u' \underset{dr}{\Rightarrow} \alpha'\beta' u' = \alpha\beta v = \gamma v$$

pentru care k:u = k:v, are loc: α = α', A = A', β = β'.

Să vedem care este semnificaţia acestor condiţii în contextul analizei sintactice. Să ne amintim că un pas în analiza sintactică ascendentă înseamnă:

- determinarea, în γ= αβu, a obiectului derivării β;
- reducerea lui β la A.

Cum în algoritmul general de *deplasare-reducere*, αβ se află în stivă iar u se află în banda de intrare, rezultă că o gramatică G este LR(k) dacă, în orice pas al analizei sintactice ascendente, obiectul derivării β şi producţia A → β cu care se face reducerea sunt determinate în mod unic prin inspectarea primelor k simboluri care au rămas de analizat (pe banda de intrare). Definiţia pe care am dat-o pentru gramatici LR(k) este dinamică: condiţia ca o gramatică să fie LR(k) trebuie să fie îndeplinită de orice două perechi de derivări. Cum, în general, numărul acestora este infinit, nu se poate verifica dacă o gramatică este LR(k) folosind definiţia. Vom încerca să găsim condiţii echivalente cu

definiţia pentru cazurile k = 0 şi k = 1. Să stabilim în continuare câteva proprietăţi ale gramaticilor LR(k).

Teorema 5.1.1 Dacă G este gramatică LR(k), k ≥ 0, atunci G este neambiguă.

Demonstraţie Fie G = (V, T, S, P) o gramatică LR(k) şi să presupunem, prin reducere la absurd, că ea este ambiguă. Există atunci un cuvânt w ∈ L(G) care are două derivări extrem drepte distincte:

$$S' \Rightarrow S = \gamma_0 \underset{dr}{\Rightarrow} \gamma_1 \underset{dr}{\Rightarrow} \dots \underset{dr}{\Rightarrow} \gamma_n = w$$

$$S' \Rightarrow S = \delta_0 \underset{dr}{\Rightarrow} \delta_1 \underset{dr}{\Rightarrow} \dots \underset{dr}{\Rightarrow} \delta_m = w$$

Să observăm că $\gamma_n = \delta_m = w$. Cele două derivări sunt presupuse distincte, deci există un număr i, $1 \le i \le \min(m, n)$ astfel încât $\gamma_{n-i-1} \ne \delta_{m-i-1}$, iar $\gamma_{n-i} = \delta_{m-i}$. Avem atunci:

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \gamma_{n-i-1} = \alpha A u \underset{dr}{\Rightarrow} \alpha\beta u = \gamma_{n-i}$$

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \delta_{m-i-1} = \alpha' A' u' \underset{dr}{\Rightarrow} \alpha'\beta' u' = \alpha\beta u = \delta_{m-i}$$

Dar G este LR(k) şi pentru că are loc k:u = k:u rezultă α = α', A = A', β = β' ceea ce este o contradicţie. Aşadar, presupunerea că există un număr i, $1 \le i \le \min(m, n)$ astfel încât $\gamma_{n-i-1} \ne \delta_{m-i-1}$ este falsă. Rezultă m = n şi $\gamma_i = \delta_i$, $1 \le i \le n$.

□

4.2 O caracterizare a gramaticilor LR(0)

Caracterizarea pe care o vom da pentru gramaticile LR(0) foloseşte un automat finit determinist care joacă un rol esenţial în construcţia analizoarelor de tip LR.

Definiţia 5.2.1 Fie G = (V, T, S, P) o gramatică independentă de context redusă. Să presupunem că simbolul • nu este în Σ. Un *articol* pentru gramatica G este o producţie A → γ în care s-a adăugat simbolul • într-o anume poziţie din γ. Notăm un articol prin A → α•β dacă γ = αβ. Un articol în care • este pe ultima poziţie se numeşte *articol complet*.

Definiţia 5.2.2 Un *prefix viabil* pentru gramatica G = (V, T, S, P) este orice prefix al unui cuvânt αβ dacă $S \overset{*}{\underset{dr}{\Rightarrow}} \alpha A u \underset{dr}{\Rightarrow} \alpha\beta u$. Dacă $\beta = \beta_1\beta_2$ şi $\varphi = \alpha\beta_1$ spunem că articolul A → $\beta_1 \bullet \beta_2$ este *valid* pentru *prefixul viabil* φ.

Exemplul 5.2.1 Fie gramatica S → A, A → aAa | bAb | c. Articolele acestei gramatici sunt:

S → •A, S → A•, A → •aAa, A → a•Aa, A → aA•a, A → aAa•,
A → •bAb, A → b•Ab, A → bA•b, A → bAb•, A → •c, A → c•.

Dacă, în plus, avem şi regula A → ε, atunci singurul articol corespunzător acesteia este articolul (complet) A → •. În următorul tabel sunt date şi câteva exemple de articole valide pentru prefixe viabile:

Prefixul viabil	Articole valide	Derivarea corespunzătoare
ab	A → b•Ab	S ⇒ A ⇒ aAa ⇒ abAba
	A → •aAa	S ⇒ A ⇒ aAa ⇒ abAba ⇒ abaAaba
	A → •bAb	S ⇒ A ⇒ aAa ⇒ abAba ⇒ abbAbba

ε	$S \to \bullet A$	$S \Rightarrow A$
	$A \to \bullet bAb$	$S \Rightarrow A \Rightarrow bAb$
	$A \to \bullet c$	$S \Rightarrow A \Rightarrow c$

Lema 5.2.1 Fie G = (V, T, S, P) o gramatică şi $A \to \beta_1 \bullet B\beta_2$ un articol valid pentru prefixul viabil γ. Atunci, oricare ar fi producţia $B \to \beta$, articolul $B \to \bullet\beta$ este valid pentru γ.

Demonstraţie Dacă $A \to \beta_1 \bullet B\beta_2$ este valid pentru γ există derivarea:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \alpha Au \underset{dr}{\Rightarrow} \alpha\beta_1 B\beta_2 u = \gamma B\beta_2 u$$

Dar G este redusă, deci există derivarea $\beta_2 \underset{dr}{\overset{*}{\Rightarrow}} v$ şi atunci putem scrie:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \alpha Au \underset{dr}{\Rightarrow} \alpha\beta_1 B\beta_2 u \underset{dr}{\overset{*}{\Rightarrow}} \alpha\beta_1 Bvu \underset{dr}{\Rightarrow} \alpha\beta_1\beta vu = \gamma\beta vu$$

ceea ce înseamnă că $B \to \bullet\beta$ este valid pentru γ.

□

Teorema 5.2.1 Gramatica G = (V, T, S, P) este gramatică LR(0) dacă şi numai dacă, oricare ar fi prefixul viabil γ, sunt îndeplinite condiţiile:

1. nu există două articole complete valide pentru γ.
2. dacă articolul $A \to \beta\bullet$ este valid pentru γ, nu există nici un articol $B \to \beta_1 \bullet a\beta_2$, $a \in T$, valid pentru γ.

Demonstraţie Să presupunem că G este gramatică LR(0). Atunci, conform definiţiei, oricare ar fi două derivări:

$$S' \Rightarrow S \underset{dr}{\overset{*}{\Rightarrow}} \alpha Au \underset{dr}{\Rightarrow} \alpha\beta u = \delta u \qquad (1)$$

$$S' \Rightarrow S \underset{dr}{\overset{*}{\Rightarrow}} \alpha'A'u' \underset{dr}{\Rightarrow} \alpha'\beta'u' = \delta v \qquad (2)$$

are loc: α = α', A = A', β = β'.

Să presupunem, prin reducere la absurd, că există un prefix viabil γ pentru care nu sunt îndeplinite simultan 1 şi 2. Dacă 1 nu are loc, există articolele $B \to \beta\bullet$ şi $B' \to \beta'\bullet$, diferite, valide pentru γ. Asta înseamnă că:

$$S' \Rightarrow S \underset{dr}{\overset{*}{\Rightarrow}} \varphi Bx \underset{dr}{\Rightarrow} \varphi\beta x = \gamma x \qquad (3)$$

$$S' \Rightarrow S \underset{dr}{\overset{*}{\Rightarrow}} \varphi'B'x' \underset{dr}{\Rightarrow} \varphi'\beta'x' = \gamma x \qquad (4)$$

Ori, (3) şi (4) contrazic faptul că G este LR(0).

Dacă 2 nu are loc, există articolele $B \to \beta\bullet$, $C \to \beta_1 \bullet a\beta_2$ valide pentru γ, adică are loc derivarea (3) şi derivarea:

$$S' \Rightarrow S \underset{dr}{\overset{*}{\Rightarrow}} \psi Cy \underset{dr}{\Rightarrow} \psi\beta_1 a\beta_2 y = \gamma a\beta_2 y \quad (5)$$

Dacă în (5) $\beta_2 \in T^*$ atunci din (3), (5) şi faptul că G este LR(0), rezultă că φ = = ψ, B = C, $\beta = \beta_1 a\beta_2$. Dar $\gamma = \varphi\beta = \psi\beta_1$ ceea ce înseamnă că $\varphi\beta_1 a\beta_2 = \varphi\beta_1$ ceea ce este imposibil deoarece $a \in T$.

Dacă în (5) β_2 conţine măcar o variabilă, cum G este redusă, există derivarea:

$$\beta_2 \underset{dr}{\overset{*}{\Rightarrow}} u_1 Du_3 \underset{dr}{\Rightarrow} u_1u_2u_3$$

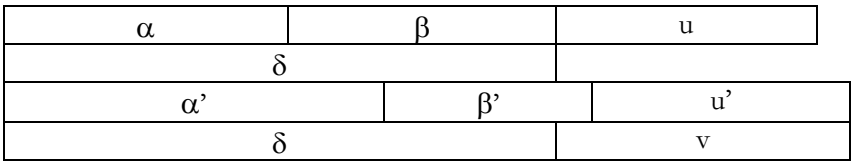
Aplicând acest lucru în (5) se obţine:

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \gamma a\beta_2 y \overset{*}{\underset{dr}{\Rightarrow}} \gamma a u_1 D u_3 y \underset{dr}{\Rightarrow} \gamma a u_1 u_2 u_3 y \quad (6)$$

Din (3) şi (6), pentru că G este LR(0), rezultă $\varphi = \gamma a u_1$, $B = D$, $\beta = u_2$. Dar din derivarea (3) are loc $\gamma = \varphi\beta$ şi egalitatea $\varphi = \varphi\beta a u_1$ este imposibilă pentru că a este terminal (nu poate fi nul). Aşadar, o implicaţie a teoremei este dovedită.

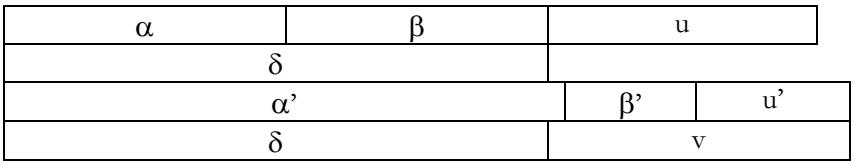
Să presupunem acum că oricare ar fi prefixul viabil γ au loc condiţiile 1 şi 2 din teoremă şi, prin reducere la absurd, G nu este LR(0). Asta înseamnă că există două derivări de forma (1) şi (2) dar nu este adevărat că $\alpha = \alpha'$, $A = A'$ şi $\beta = \beta'$. Fără a restrânge generalitatea, presupunem că în (1) şi (2) avem îndeplinită condiţia $|\delta| = |\alpha\beta| \leq |\alpha'\beta'|$. Distingem două cazuri:

Cazul 1: $|\alpha'| \leq |\delta|$. Schematic acest lucru arată astfel:



Rezultă de aici că $\beta' = \beta'_1\beta'_2$, $v = \beta'_2 u'$, $\delta = \alpha'\beta'_1$ şi $\beta'_2 \in T^*$. Atunci, din (1) rezultă că $A \rightarrow \beta\bullet$ este articol valid pentru δ iar din (2) rezultă că articolul $A' \rightarrow \beta'_1\bullet\beta'_2$ este valid de asemenea pentru δ. Asta contrazice 1 dacă $\beta'_2 = \varepsilon$ respectiv 2 dacă $\beta'_2 \neq \varepsilon$. Aşadar, în acest caz, presupunerea că G nu este LR(0) este falsă.

Cazul 2: $|\alpha'| > |\delta|$. Din nou schematic condiţia arată astfel:



Asta înseamnă că $\alpha' = \delta u_1$, $v = u_1\beta' u' \in T^*$ ($|u_1| \geq 1$). În derivarea (2), punem în evidenţă prima formă propoziţională care are prefixul δ; aceasta există deoarece S nu are prefixul δ, iar $\alpha'\beta' u' = \delta v$ are prefixul δ. Aşadar (2) devine:

$$S' \Rightarrow S \overset{*}{\underset{dr}{\Rightarrow}} \alpha_1 A_1 u_1 \underset{dr}{\Rightarrow} \alpha_1\beta_1 u_1 = \alpha_1\beta'_1\beta''_1 u_1 = \delta\beta''_1 u_1 \overset{*}{\underset{dr}{\Rightarrow}} \delta v \quad (7)$$

Din (7) rezultă că $A_1 \rightarrow \beta'_1\bullet\beta''_1$ este articol valid pentru δ (iar din (1) rezultă că $A \rightarrow \beta\bullet$ este valid de asemenea pentru δ).

Distingem şi aici trei subcazuri:

Cazul 2.1: $\beta''_1 = \varepsilon$, contradicţie: condiţia 1 din teoremă.

Cazul 2.2: $\beta''_1 = a\beta_2$, $a \in T$. Atunci $A_1 \rightarrow \beta'_1\bullet a\beta_2$, $A \rightarrow \beta\bullet$ sunt valide pentru δ, ceea ce contrazice 2 din teoremă.

Cazul 2.3: $\beta''_1 = B\beta_2$, $B \in V$. Să observăm atunci că există un articol $C \rightarrow \bullet a\varphi$ valid pentru δ (eventual $C = B$) pentru că gramatica este redusă şi există măcar o producţie a cărei parte dreaptă începe cu un terminal (se foloseşte şi Lema 5.2.1). Şi în acest caz se contrazice 2 din teoremă. În concluzie, G este LR(0).

□

Teorema precedentă ne îndreptăţeşte să cercetăm mai îndeaproape mulţimea prefixelor viabile pentru o gramatică G. Are loc:

Teorema 5.2.2 Fie G = (V, T, S, P) o gramatică independentă de context. Mulţimea prefixelor viabile pentru gramatica G este limbaj regulat.

Demonstraţie Să considerăm gramatica G' îmbogăţită cu S' → S. Vom construi un automat (nedeterminist cu ε-tranziţii) care recunoaşte mulţimea prefixelor viabile ale lui G. Considerăm automatul M = (Q, Σ, δ, q_0, Q), unde Q este mulţimea articolelor gramaticii G', Σ = V ∪ T, q_0 = S' → •S iar funcţia de tranziţie δ : Q × (Σ ∪ {ε}) → 2^Q este definită astfel:

$$\delta(A \to \alpha\bullet B\beta, \varepsilon) = \{B \to \bullet\gamma \mid B \to \gamma \in P\}.$$
$$\delta(A \to \alpha\bullet X\beta, X) = \{ A \to \alpha X\bullet\beta\}, X \in \Sigma.$$
$$\delta(A \to \alpha\bullet a\beta, \varepsilon) = \Phi, \forall a \in T.$$
$$\delta(A \to \alpha\bullet X\beta, Y) = \Phi, \forall X,Y \in \Sigma \text{ cu } X \neq Y.$$

Aşadar, automatul are ε-tranziţii numai din articolele (stările) care au un neterminal după punct iar X - tranziţii, X∈ Σ, din articolele care au X (acelaşi simbol) după punct. Să dovedim că are loc:

$A \to \alpha\bullet\beta \in \hat{\delta}(q_0, \gamma) \Leftrightarrow \gamma$ este prefix viabil şi $A \to \alpha\bullet\beta$ este valid pentru γ.

Să presupunem mai întâi că γ este prefix viabil şi $A \to \alpha\bullet\beta$ este valid pentru γ. Asta înseamnă că are loc o derivare de forma:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi Au \underset{dr}{\Rightarrow} \varphi\alpha\beta u = \gamma\beta u$$

Vom dovedi, prin inducţie după k, lungimea derivării, că $A \to \bullet\alpha\beta \in \hat{\delta}(q_0, \varphi)$. Se deduce apoi că $\hat{\delta}(q_0, \varphi\alpha)$ conţine articolul $A \to \alpha\bullet\beta$ (din definiţia automatului şi a extensiei lui δ).

k = 1: În acest caz derivarea este $S \underset{dr}{\Rightarrow} \alpha\beta$, deci $S \to \alpha\beta \in P$, φ = u = ε şi, după definiţia lui δ are loc $S \to \bullet\alpha\beta \in \hat{\delta}(q_0, \varepsilon)$.

k > 1: În derivarea de lungime k punem în evidenţă pasul la care s-a introdus neterminalul A:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi_1 Bu_2 \underset{dr}{\Rightarrow} \varphi_1\alpha_1 A\beta_1 u_2 \underset{dr}{\overset{*}{\Rightarrow}} \varphi_1\alpha_1 Au_1u_2 = \varphi Au \underset{dr}{\Rightarrow} \varphi\alpha\beta u = \gamma\beta u$$

unde $B \to \alpha_1 Au_2 \in P$, $\beta_1 \underset{dr}{\overset{*}{\Rightarrow}} u_1$, $u_1u_2 = u$, $\varphi_1\alpha_1 = \varphi$. E clar că derivarea

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi_1 Bu_2 \underset{dr}{\Rightarrow} \varphi_1\alpha_1 A\beta_1 u_2$$

are lungimea cel mult k – 1 şi, după ipoteza inductivă, are loc $B \to \bullet\alpha_1 Au_2 \in \hat{\delta}(q_0, \varphi_1)$ iar din definiţia extensiei lui δ avem şi $B \to \alpha_1\bullet Au_2 \in \hat{\delta}(q_0, \varphi_1\alpha_1)$. Atunci $A \to \bullet\alpha\beta \in \delta(B \to \alpha_1\bullet Au_2, \varepsilon)$ (definiţia lui δ) şi $A \to \bullet\alpha\beta \in \hat{\delta}(q_0, \varphi_1\alpha_1) = \hat{\delta}(q_0, \varphi)$ (definiţia lui $\hat{\delta}$).

Să presupunem acum că $A \to \alpha\bullet\beta \in \hat{\delta}(q_0, \gamma)$. Atunci, în graful automatului M există măcar un drum de la q_0 la starea $A \to \alpha\bullet\beta$ etichetat cu γ = φα (drum ce conţine eventual şi arce etichetate ε). Considerăm drumul de lungime minimă (unul din ele dacă sunt mai multe) cu această proprietate. Fie k lungimea acestuia. Demonstrăm prin inducţie după k faptul că γ este prefix viabil pentru G iar articolul $A \to \alpha\bullet\beta$ este valid pentru γ.

k = 1: Drumul este de la q_0 = S' → •S la o stare $S \to \bullet\beta$ sau la S' → S•. În primul caz γ = ε care este prefix viabil iar $A \to \alpha\bullet\beta = S \to \bullet\beta$ este valid pentru ε pentru că $S' \underset{dr}{\Rightarrow} S$

$\underset{dr}{\Rightarrow}\beta$. În al doilea caz $\gamma = S$ este prefix viabil iar $A \to \alpha\bullet\beta = S' \to S\bullet$ este valid pentru S pentru că $S' \underset{dr}{\Rightarrow} S$.

k > 1: În funcţie de ultimul arc din drum distingem două cazuri:

Cazul 1: Ultimul arc este etichetat cu $X \in \Sigma$. Atunci el are forma din figura 5.2.1. Aşadar $\gamma = \varphi\alpha_1 X = \gamma_1 X$.

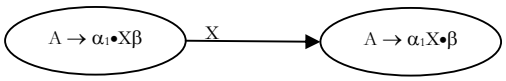


Figura 5.2.1

După ipoteza inductivă, γ_1 este prefix viabil iar articolul $A \to \alpha_1\bullet X\beta$ este valid pentru el. De aici rezultă existenţa unei derivări

$$S \overset{*}{\underset{dr}{\Rightarrow}} \varphi Au \underset{dr}{\Rightarrow} \varphi\alpha_1 X\beta u = \gamma_1 X\beta u = \gamma\beta u$$

de unde rezultă faptul că γ este prefix viabil iar $A \to \alpha_1 X\bullet\beta$ este valid pentru γ.

Cazul 2: Ultimul arc al drumului este etichetat cu ε. Atunci $\alpha = \varepsilon$ iar arcul este de forma celui din figura 5.2.2.

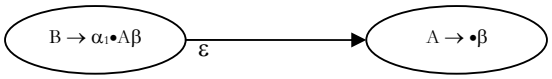


Figura 5.2.2

După ipoteza inductivă $B \to \alpha_1\bullet A\beta_1$ este valid pentru γ, adică există derivarea:

$$S \overset{*}{\underset{dr}{\Rightarrow}} \varphi Bu \underset{dr}{\Rightarrow} \varphi\alpha_1 A\beta_1 u = \gamma A\beta u$$

Dacă aplicăm $\beta \overset{*}{\underset{dr}{\Rightarrow}} v$ (gramatica este redusă) în derivarea de mai sus obţinem:

$$S \overset{*}{\underset{dr}{\Rightarrow}} \varphi Bu \underset{dr}{\Rightarrow} \varphi\alpha_1 A\beta_1 u = \gamma A\beta_1 u \overset{*}{\underset{dr}{\Rightarrow}} \gamma Avu \underset{dr}{\Rightarrow} \gamma\beta vu$$

ceea ce dovedeşte că şi $A \to \bullet\beta$ este valid pentru γ. Cu aceasta demonstraţia teoremei este încheiată.

□

Exemplul 5.2.2 Fie gramatica $S' \to S$, $S \to aSa \mid bSb \mid c$. Automatul care recunoaşte prefixele viabile este reprezentat în figura 5.2.3. Arcele neetichetate constitue ε-tranziţii. Condiţia necesară şi suficientă ca o gramatică G să fie LR(0), dată în Teorema 5.2.1, se "traduce" relativ la automatul construit în Teorema 5.2.2 astfel:

1. Nu există în graful automatului două noduri cu etichete articole complete, respectiv $A \to \alpha\bullet$ şi $B \to \beta\bullet$, şi câte un drum de la q_0 la fiecare din acestea etichetate cu acelaşi cuvânt γ.
2. Nu există în graful automatului două noduri ce au etichete $A \to \alpha\bullet$ respectiv $B \to \beta\bullet a\beta'$, şi câte un drum de la q_0 la fiecare din acestea etichetate cu acelaşi cuvânt γ;

Aceste două condiţii sunt greoi de verificat pe automatul (nedeterminist) cu ε-tranziţii construit. Pentru acest automat, există un automat determinist echivalent, fie acesta $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$ unde $\mathcal{T} \subseteq 2^Q$ (o stare pentru automatul determinist este o submulţime de articole), t_0 conţine toate articolele accesibile prin ε-tranziţii din $q_0 = S' \to \bullet S$, iar $g: \mathcal{T}\times\Sigma \to \mathcal{T}$ este parţial definită (vezi 2.2.2). Cum M recunoaşte mulţimea prefixelor viabile şi articolul $A \to \alpha\bullet\beta$ este în $\hat{\delta}(q_0, \gamma)$ dacă şi numai dacă $A \to \alpha\bullet\beta$ este valid pentru

prefixul viabil γ, înseamnă că, în automatul determinist, o stare $t \in \mathcal{T}$ conţine toate articolele valide pentru o anume mulţime de prefixe viabile şi numai pe acestea. Aşadar, dacă am construit $\mathcal{M}$, automatul determinist echivalent cu M, condiţiile necesare şi suficiente pentru ca G să fie gramatică LR(0) sunt:

1. Orice stare $t \in \mathcal{T}$ conţine cel mult un articol complet.
2. Pentru orice $t \in \mathcal{T}$, dacă t conţine un articol complet, atunci t nu conţine nici un articol cu terminal după punct.

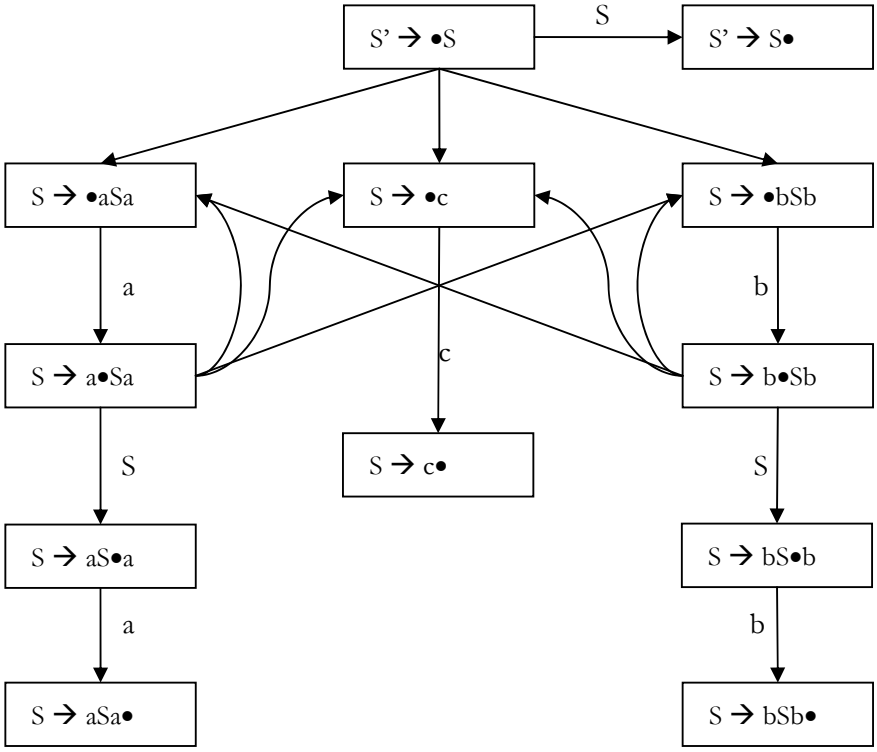


Figura 5.2.3

Aceste două condiţii sunt uşor de verificat dacă $\mathcal{M}$ este construit. $\mathcal{M}$ se construieşte aplicând algoritmul de trecere de la un automat (nedeterminist) cu ε-tranziţii la automatul determinist echivalent (vezi Teorema 2.2.2). Pentru că această problemă este deosebit de importantă pentru analiza sintactică de tip LR, vom indica în continuare un algoritm pentru obţinerea automatului $\mathcal{M}$, numit *automatul LR(0)* al gramaticii G. Mai întâi vom descrie o procedură, numită închidere(t) care are la intrare o mulţime t de articole (stări ale lui M) şi obţine la ieşire toate stările lui M accesibile din t prin ε.

Algoritmul 5.2.1 (procedura închidere(t))
Intrare: Gramatica G = (V, T, S, P);
Mulţimea t de articole din gramatica G;
Ieşire: t' = închidere(t) = $\{q \in Q \mid \exists p \in t, q \in \hat{\delta}(p, \varepsilon)\} = \hat{\delta}(t, \varepsilon)$
Metoda:

```
1. t' = t ; flag = true;
2. while( flag ) {
3.   flag = false;
4.   for ( A → α•Bβ ∈ t' ) {
```

```
5.        for ( B → γ ∈ P )
6.              if ( B → •γ ∉ t' ) {
7.                    t' = t' ∪ {B → •γ};
8.                    flag = true;
                }//endif
          }//endforB
       }//endforA
    }//endwhile
9.  return t';
```

Să observăm că algoritmul precedent nu foloseşte automatul M. Legătura cu acest automat este dată de:

Lema 5.2.2 Dacă M = (Q, Σ, δ, q_0, Q) este automatul cu ε-tranziţii al gramaticii G şi t ⊆ Q, atunci, în urma aplicării algoritmului precedent, se obţine t' = $\hat{\delta}$ (t, ε).

Demonstraţie Fie t' = închidere(t). Să arătăm mai întâi că t' ⊆ $\hat{\delta}$ (t, ε). Fie articolul A → α•β ∈ t'. Procedăm prin inducţie asupra momentului când a fost adăugat acest articol la t'. Dacă acest lucru a fost făcut la linia 1, rezultă că A → α•β ∈ t evident t ⊆ $\hat{\delta}$ (t, ε) încât, A → α•β ∈ $\hat{\delta}$ (t, ε). Dacă adăugarea s-a făcut la linia 7, înseamnă că în t', în acel moment, există B → γ•Aφ care a fost adăugat anterior iar α = ε. După ipoteza inductivă, B → γ•Aφ B ∈ $\hat{\delta}$ (t, ε) iar din definiţia lui δ (Teorema 5.2.2) rezultă că A →•β ∈ δ(B →γ•Aφ, ε) încât A →•β este în $\hat{\delta}$ (t, ε).

Invers, fie A → α•β ∈ $\hat{\delta}$ (t, ε). Prin inducţie asupra lungimii drumului de la o stare din t la A → α•β se arată uşor că A → α•β se adaugă la t' (în linia 1 sau în linia 7).

□

Să descriem acum algoritmul pentru construcţia automatului LR(0) pentru gramatica G.

Algoritmul 5.2.2 Automatul LR(0)

Intrare: Gramatica G = (V, T, S, P) la care s-a adăugat S' → S;

Ieşire: Automatul determinist $\mathcal{M}$ = ($\mathcal{T}$, Σ, *g*, t_0, $\mathcal{T}$) echivalent cu M.

Metoda:

```
1.  t0=închidere(S'→•S); T={t0}; marcat(t0)=false;
2.  while(∃t∈T && !marcat(t)) { // marcat(t) = false
3.        for( X ∈ Σ ) {
4.              t' = Φ;
5.              for( A → α•Xβ ∈ t )
6.                    t' = t' ∪ {B → αX•β | B → α•Xβ ∈ t};
7.                    if( t' ≠ Φ ){
8.                          t' = închidere( t' );
9.                          if( t' ∉ T ) {
10.                               T = T ∪ { t' };
11.                               marcat( t' ) = false;
                            }//endif
12.                         g(t, X) = t';
                      }//endif
                }//endfor
          }//endfor
13.       marcat( t ) = true;
    }// endwhile
```

Lema 5.2.3 Algoritmul 5.2.2 descris mai sus este corect: automatul $\mathcal{M}$ obţinut este determinist şi este echivalent cu M.

Demonstraţie Faptul că $\mathcal{M}$ este determinist rezultă din linia 12. Condiţia din linia 7 face ca g să fie parţial definită: dacă $\hat{\delta}(t, X) = \Phi$ în automatul M, atunci $g(t, X)$ nu este definită. Să mai observăm că, dacă $t' \neq \Phi$ în linia 7, atunci faptul că $g(t, X) = \hat{\delta}(t, X)$ ne îndreptăţeşte să afirmăm că $\mathcal{M}$ este echivalent cu M.

□

Exemplul 5.2.3 Pentru gramatica S' → S, S → aSa | bSb | c, automatul LR(0) este dat în figura 5.2.4. Se constată cu uşurinţă că automatul construit îndeplineşte condiţiile LR(0) deci gramatica este LR(0). Dacă înlocuim producţia A → c cu A → ε, în starea iniţială, în loc de A → •c o să avem articolul A → •, care este un articol complet. Cum în aceeaşi stare există articolul A → •aAa, aceasta contrazice condiţia 2 din caracterizarea LR(0); gramatica S' → S, S → aSa | bSb | ε , nu este LR(0).

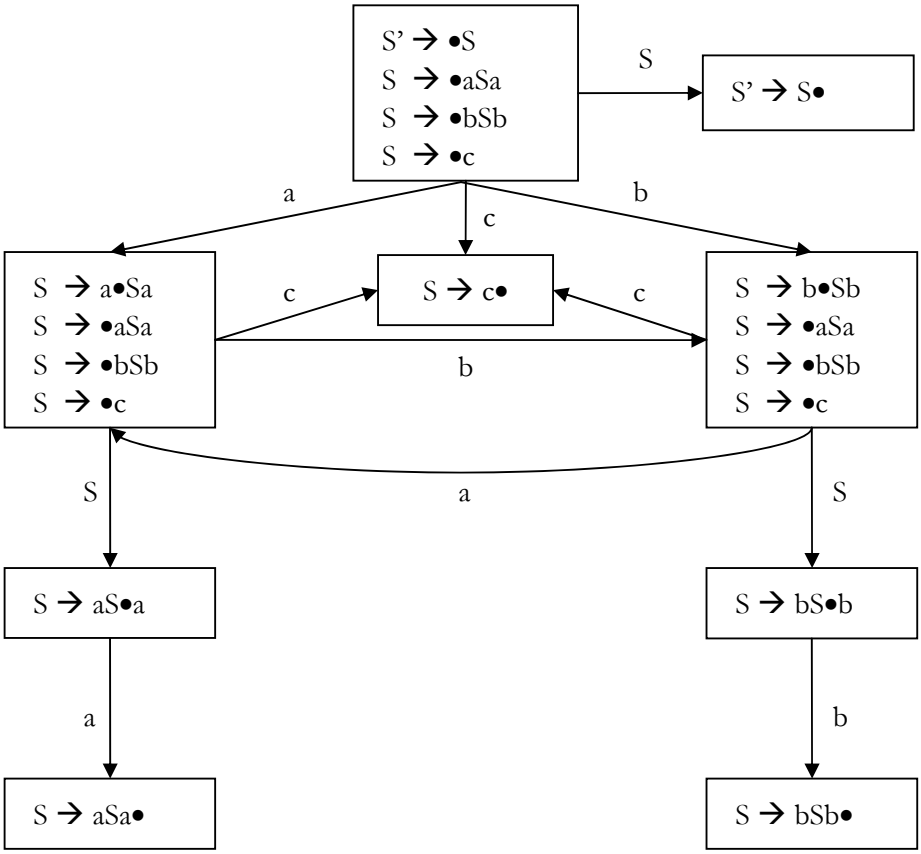
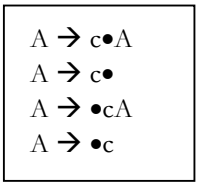


Figura 5.2.4

□

Exemplul 5.2.4 Fie gramatica: S → aAd | bAB A → cA | c B → d
Automatul LR(0) corespunzător este dat în figura 5.2.5. Se observă că starea:



conţine articolul complet A → c• şi articolele A → •cA, A → •c care au terminal după punct; gramatica nu este LR(0).

□

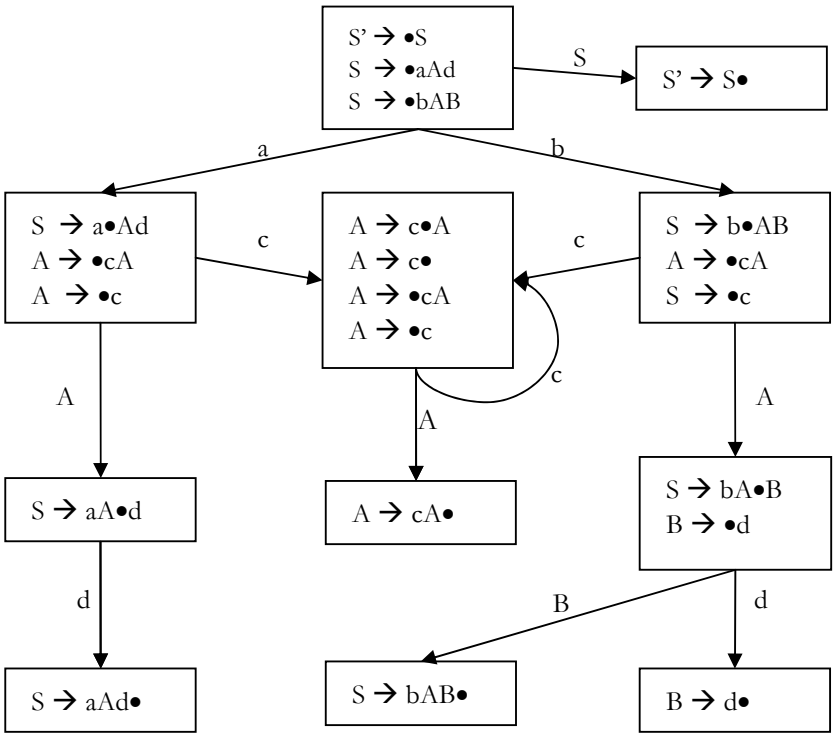


Figura 5.2.5

Clasa limbajelor LR(0) are mai multe caracterizări. Următoarea teoremă are o importanţă (mai ales teoretică) deosebită; demonstraţia ei se poate găsi în [Har78] pag 515.

Teorema 5.2.3 Fie $L \subseteq \Sigma^*$. Următoarele propoziţii sunt echivalente:

1. L este limbaj LR(0) (există G, o gramatică LR(0), cu L = L(G)).
2. L este un limbaj independent de context determinist cu proprietatea:
   $\forall x \in \Sigma^+$, $\forall w, y \in \Sigma^*$, dacă $w \in L$ şi $wx \in L$ iar $y \in L$, atunci $yx \in L$.
3. Există un automat pushdown determinist $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, unde $F = \{q_f\}$ şi există $Z_f \in \Gamma$ astfel ca:
   $L = T(A, Z_f) = T(A, \Gamma) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q_f, \varepsilon, Z_f)\}$.
4. Există limbajele strict deterministe L_0 şi L_1 astfel încât $L = L_0 L_1^*$.

4.3 Algoritm de analiză sintactică LR(0)

Modelul de analizor sintactic LR(k) este dat în figura 5.3.1.

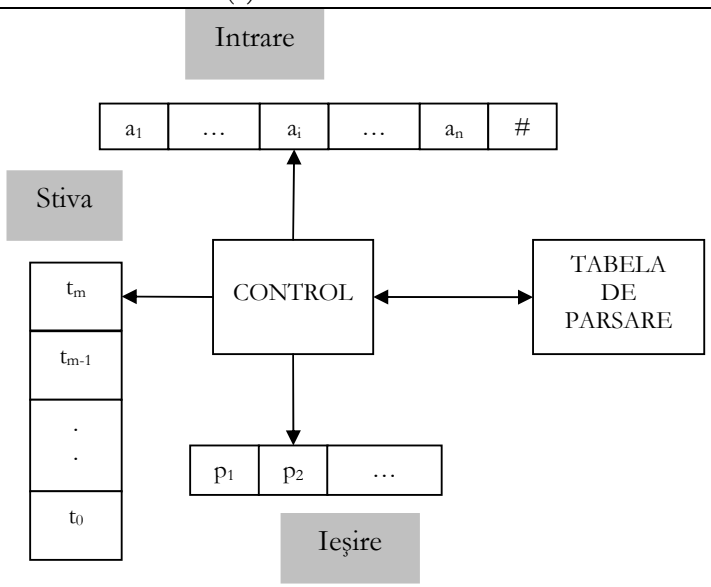


Figura 5.3.1

În cazul k = 0, tabela de parsare coincide cu automatul LR(0), $\mathcal{M}$. O configuraţie a analizorului LR(0) este o tripletă $(\sigma, u\#, \pi)$ unde $\sigma \in t_0\mathcal{T}^*$, (σ este o secvenţă de stări care începe cu t_0), $u \in T^*$, $\pi \in P^*$. Configuraţia iniţială este $(t_0, w\#, \varepsilon)$, unde w este cuvântul care trebuie analizat. Tranziţiile pe care le execută analizorul LR(0) sunt:

Deplasare: $(\sigma t, au\#, \pi) \vdash (\sigma tt', u\#, \pi)$ dacă $g(t, a) = t'$.

Reducere: $(\sigma t\sigma' t', u\#, \pi) \vdash (\sigma tt'', u\#, \pi r)$ dacă $A \to \beta\bullet \in t'$,

$r = A \to \beta$, $|\sigma' t'| = |\beta|$ şi $t'' = g(t, A)$.

Acceptare: $(t_0t_1, \#, \pi)$ este configuraţia de acceptare dacă $S' \to S\bullet \in t_1$. Analizorul se opreşte cu acceptarea cuvântului de analizat, iar π este parsarea acestuia (şirul de reguli care s-a aplicat, în ordine inversă, în derivarea extrem dreaptă a lui w).

Eroare: o configuraţie căreia nu i se poate aplica nici o tranziţie este configuraţie de eroare. Analizorul se opreşte cu respingerea cuvântului de analizat.

Algoritmul de analiză sintactică LR(0) funcţionează astfel:

Se porneşte cu starea iniţială a automatului LR(0) în topul stivei şi cuvântul de analizat în banda de intrare. Se execută o serie de operaţii de *deplasare - reducere*, până când, eventual, întreg cuvântul de intrare a fost parcurs şi ultima reducere a fost făcută cu regula $S' \to S$. Dacă nu se obţine acest lucru, atunci analiza sintactică eşuează: cuvântul nu este corect sintactic. Alegerea operaţiei de reducere sau de deplasare este dictată de starea din top-ul stivei:

- dacă această stare nu conţine un articol complet şi în automat există tranziţie din aceasta cu simbolul curent de intrare, se produce o deplasare: se înaintează cu o poziţie în banda de intrare (în alte implementări simbolul curent din intrare este deplasat în stivă) şi se adaugă în stivă starea obţinută în automatul LR(0) prin tranziţia cu simbolul în discuţie. În acest mod se parcurg în automatul LR(0), începând cu t_0, drumuri corespunzătoare prefixelor viabile din derivarea extrem dreaptă pentru cuvântul de analizat.
- dacă starea din top-ul stivei conţine un articol complet $A \to \beta\bullet$, atunci se produce o reducere: în forma propoziţională curentă, β se înlocuieşte cu A. În termenii configuraţiei de aici, asta înseamnă că vor fi eliminate din stivă un număr de stări egal cu lungimea lui β (stări care reprezintă în automatul LR(0)

drumul etichetat cu β) şi se adăugă starea obţinută prin tranziţie din starea rămasă în topul stivei cu A. În acest caz se raportează la ieşire aplicarea producţiei p = A →β. Să descriem algoritmul LR(0) şi în pseudocod.

Algoritmul 5.3.1 (Analiză sintactică LR(0))

Intrare: Gramatica G = (V, T, S, P) care este LR(0) augmentată cu S' → S.
Automatul LR(0) al gramaticii G, notat $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$.
Cuvântul de intrare w ∈ T*.

Ieşire: Analiza sintactică (parsarea) ascendentă a lui w dacă w ∈ L(G);
Eroare, în caz contrar.

Metoda: Fie STIVA o stivă, a simbolul curent,

```
char ps[] = "w#"; //ps este sirul de intrare w
i = 0; // pozitia in sirul de intrare
STIVA.push(t0); // se initializeaza stiva cu t0
while(true) { // se repeta pana la succes sau eroare
      t = STIVA.top();
      a = ps[i] // a este simbolul curent din intrare
      if( g(t, a) ≠ Φ { //deplasare
            STIVA.push(g(t, a));
            i++; //se inainteaza in intrare
      }
      else {
            if(A → X1X2…Xm• ∈ t){
                  if(A == 'S'')
                    if(a == '#')exit( "acceptare");
                    else exit("eroare");
                  else // reducere
                    for( i = 1; i ≤ m; i++) STIVA.pop();
                          STIVA.push(g(top(STIVA), A));
             } //endif
             else exit("eroare");
      }//endelse
}//endwhile
```

Exemplul 5.3.1 Fie gramatica:

S' → S	E → E+T	T → (E)
S → E$	E → T	T → a

Această gramatică generează expresiile aritmetice care au doar operatorul +, paranteze şi operandul a. Producţia S → E$ asigură că orice expresie se termină cu simbolul special $ ceea ce face ca limbajul gramaticii să fie liber de prefixe: nici un prefix propriu al unui cuvânt din limbaj nu este în limbaj.

Automatul LR(0) pentru această gramatică este dat în figura 5.3.2. Stările automatului sunt notate 0, 1, 2,..., 10 iar tabela de tranziţie a automatului, prezentată mai jos, este şi tabelă de parsare LR(0).

G	**a**	**+**	**(**	**)**	**$**	**S**	**E**	**T**
0	5		4			1	2	3
1								
2		7			6			
3								
4	5		4				8	3
5								
6								
7	5		4					9

8		7		10				
9								
10								

Marca $ care este simbol terminal al gramaticii asigură că gramatica este LR(0), lucru care se poate verifica uşor inspectând cele 10 stări ale automatului şi constatând că sunt îndeplinite condiţiile LR(0).
În tabela de mai jos, în care se vizualizează atât configuraţiile parserului, cât şi acţiunea ce se execută (deplasare sau reducere), este detaliată parsarea expresiei a+(a+a)$.

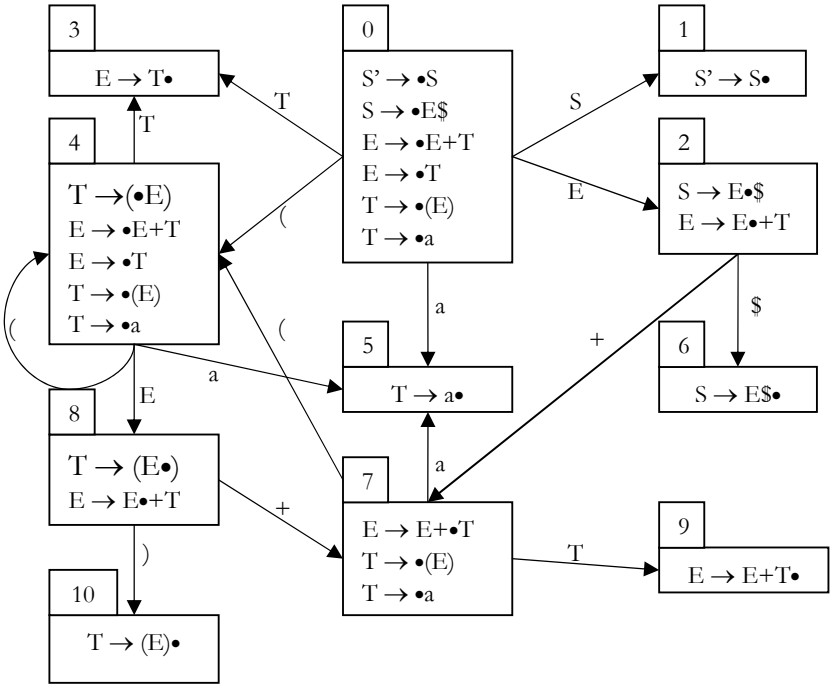


Figura 5.3.2

STIVA	INTRARE	ACŢIUNE	IEŞIRE
0	a+(a+a)$#	deplasare	
05	+(a+a)$ #	reducere	T → a
03	+(a+a)$ #	reducere	E → T
02	+(a+a)$ #	deplasare	
027	(a+a)$ #	deplasare	
0274	a+a)$ #	deplasare	
02745	+a)$ #	reducere	T → a
02743	+a)$ #	reducere	E → T
02748	+a)$ #	deplasare	
027487	a)$ #	deplasare	
0274875	)$ #	reducere	T → a
0274879	)$ #	reducere	E → E+T
02748	)$ #	deplasare	
02748(10)	$#	reducere	T → (E)
0279	$#	reducere	E → E+T
02	$#	deplasare	
026	#	reducere	S → E$
01	#	acceptare	

După acest exemplu putem să facem câteva consideraţii privind testul LR(0) pentru gramatici.

Definiţia 5.3.1 Fie G o gramatică şi $\mathcal{M}$ automatul LR(0) ataşat lui G. Spunem că o stare a lui $\mathcal{M}$ are un *conflict reducere/reducere* dacă ea conţine două articole complete distincte $A \to \alpha\bullet$, $B \to \beta\bullet$. Spunem că o stare a lui $\mathcal{M}$ are un *conflict deplasare/reducere* dacă ea conţine un articol complet $A \to \alpha\bullet$ şi un articol cu terminal după punct de forma $B \to \beta\bullet a\gamma$. Spunem că o stare este *consistentă* dacă ea nu conţine conflicte şi este *inconsistentă* în caz contrar. Cu aceste noţiuni, teorema de caracterizare a gramaticilor LR(0) se poate reformula astfel:

Teorema 5.3.1 Fie G o gramatică şi $\mathcal{M}$ automatul său LR(0). Gramatica G este LR(0) dacă şi numai dacă automatul $\mathcal{M}$ nu conţine stări inconsistente.

□

Dacă în gramatica din exemplul precedent în loc de producţia $S \to E\$$ considerăm $S \to E$, starea 2 din automat va fi $\{S \to E\bullet, E \to E\bullet+T\}$ care este *inconsistentă* deoarece conţine un conflict *deplasare/reducere*, deci gramatica nu este LR(0).

Pentru demonstrarea corectitudinii programului de analiză sintactică LR(0), să considerăm relaţia $\vdash^+$ care notează închiderea tranzitivă a relaţiei $\vdash$ definită pe mulţimea configuraţiilor. Vom dovedi două rezultate mai generale, care prin particularizare conduc la corectitudinea algoritmului.

Lema 5.3.1 Fie $G = (V, T, S, P)$ o gramatică LR(0), $t_0\sigma$, $t_0\tau$ drumuri în automatul LR(0) etichetate cu φ respectiv γ şi $u, v \in T^*$. Atunci, dacă în parserul LR(0) are loc $(t_0\sigma, uv\#, \varepsilon) \vdash^+ (t_0\tau, v\#, \pi)$, atunci în G are loc derivarea $\gamma \underset{dr}{\overset{\tilde{\pi}}{\Rightarrow}} \varphi u$.

Demonstraţie Procedăm prin inducţie după lungimea lui π.

$|\pi| = 0$: În acest caz parserul produce doar deplasări şi din definiţia acestor acţiuni rezultă că $\gamma = \varphi u$, şi derivarea din concluzie are loc în mod trivial.

$|\pi| > 0$: Fie $\pi = r\pi'$ şi $r = A \to \beta$. Calculul considerat se scrie (punând în evidenţă reducerea r) astfel:

$$(t_0\sigma, uv\#, \varepsilon) = (t_0\sigma, u_1u_2v\#, \varepsilon) \vdash^+ (t_0\sigma\sigma_1, u_2v\#, \varepsilon) =$$
$$= (t_0\sigma't'\sigma''t'', u_2v\#, \varepsilon) \vdash (t_0\sigma't't, u_2v\#, r) \vdash^+ (t_0\tau, v\#, r\pi')$$

unde eticheta subdrumului $\sigma''t''$ este β, $\varphi u_1 = \psi\beta$, ψ fiind eticheta drumului $t_0\sigma't'$, iar în automatul $\mathcal{M}$ are loc $g(t', A) = t$. După felul cum au fost definite tranziţiile rezultă că are loc şi:

$$(t_0\sigma't't, u_2v\#, \varepsilon) \vdash^+ (t_0\tau, v\#, \pi')$$

ceea ce înseamnă, conform ipotezei inductive:

$$\gamma \underset{dr}{\overset{\tilde{\pi}'}{\Rightarrow}} \psi A u_2 \text{ (} \psi A \text{ este eticheta drumului } t_0\sigma't't \text{)}$$

Aplicând în continuare regula $r = A \to \beta$ obţinem:

$$\gamma \underset{dr}{\overset{\tilde{\pi}'}{\Rightarrow}} \psi A u_2 \underset{dr}{\overset{r}{\Rightarrow}} \psi\beta u_2 = \varphi u_1 u_2 = \varphi u,$$

ceea ce trebuia demonstrat.

□

Corolarul 5.3.1 Dacă în parserul LR(0) are loc calculul $(t_0, w\#, \varepsilon) \vdash^+ (t_0t_1, \#, \pi)$ unde $S' \to S\bullet \in t_1$, atunci în gramatica G are loc derivarea $S \underset{dr}{\overset{\tilde{\pi}}{\Rightarrow}} w$.

Demonstraţie Lema 5.3.1 pentru cuvintele $\sigma = \varepsilon$, $u = w$, $v = \varepsilon$ şi $t_0\tau = t_0t_1$.

□

Lema 5.3.2 Fie G = (V, T, S, P) o gramatică LR(0) şi $\gamma \in \Sigma^+$ o formă propoziţională a lui G astfel încât $\gamma \underset{dr}{\overset{\tilde{\pi}}{\Rightarrow}} \varphi u$, cu $\varphi \in \Sigma^*$ şi dacă este nenul atunci φ:1 este neterminal. Atunci în parserul LR(0) are loc calculul

$$(t_0\sigma, uv\#, \varepsilon) \vdash^* (t_0\tau, v\#, \pi)$$

unde $t_0\sigma$ şi $t_0\tau$ sunt drumuri în automatul LR(0) etichetate cu φ respectiv γ, pentru orice cuvânt $v \in T^*$.

Demonstraţie Procedăm din nou prin inducţie după lungimea lui π.

$|\pi| = 0$: În acest caz $\gamma = \varphi u$ şi se verifică uşor că are loc calculul precizat cu tranziţii de tip deplasare.

$|\pi| > 0$: Fie $\pi = r\pi'$, $r = A \to \beta$. Atunci $\tilde{\pi} = \tilde{\pi}' r$ şi derivarea din G se scrie:

$$\gamma \underset{dr}{\overset{\tilde{\pi}'}{\Rightarrow}} \varphi_1 A u_2 \underset{dr}{\overset{r}{\Rightarrow}} \varphi_1 \beta u_2 = \varphi u_1 u_2 = \varphi u$$

Din $\varphi_1 A u_2 \underset{dr}{\overset{r}{\Rightarrow}} \varphi_1 \beta u_2 = \varphi u_1 u_2 = \varphi u$ şi faptul că γ este o formă propoziţională a lui G (deci şi $\varphi_1 A u_2$ este formă propoziţională), rezultă că în automatul $\mathcal{M}$ există un drum $t_0\sigma\sigma_1$ etichetat cu $\varphi_1\beta = \varphi u_1$ (pentru că acesta este un prefix viabil), şi:

$(t_0\sigma, u_1u_2v\#, \varepsilon) \vdash^* (t_0\sigma\sigma_1, u_2v\#, \varepsilon) = (t_0\sigma' t'\sigma'' t'', u_2v\#, \varepsilon) \vdash (t_0\sigma' t' t, u_2v\#, r)$

unde $A \to \beta\bullet \in t''$, $|\sigma'' t''| = |\beta|$, iar drumul $t_0\sigma' t' t$ este etichetat cu φA. După ipoteza inductivă, din $\gamma \underset{dr}{\overset{\tilde{\pi}'}{\Rightarrow}} \varphi_1 A u_2$ rezultă calculul:

$$(t_0\sigma' t' t, u_2v\#, \varepsilon) \vdash^* (t_0\tau, v\#, \pi').$$

Asta înseamnă că putem scrie:

$$(t_0\sigma, u_1u_2v\#, \varepsilon) \vdash^* (t_0\sigma\sigma_1, u_2v\#, \varepsilon) \vdash (t_0\sigma' t' t, v\#, r) \vdash^* (t_0\tau, v\#, r\pi')$$

ceea ce trebuia demonstrat.

□

Corolarul 5.3.2 Dacă în gramatica G, care este LR(0), avem $S \underset{dr}{\overset{\tilde{\pi}'}{\Rightarrow}} w$, atunci în parserul LR(0) are loc calculul $(t_0, w\#, \varepsilon) \vdash^+ (t_0t_1, \#, \pi)$ unde $S' \to S\bullet \in t_1$.

Demonstraţie Rezultă din lema precedentă luând $\gamma = S$, $\varphi = \varepsilon$, $u = w$, $v = \varepsilon$.

□

Combinând cele două leme obţinem teorema de corectitudine a algoritmului de analiză sintactică LR(0):

Teorema 5.3.2 Dacă G este gramatică LR(0) atunci, oricare ar fi cuvântul de intrare $w \in T^*$, parserul LR(0) ajunge la configuraţia de acceptare pentru w, adică $(t_0, w\#, \varepsilon) \vdash^+ (t_0t_1, \#, \pi)$, dacă şi numai dacă $S \underset{dr}{\overset{\tilde{\pi}}{\Rightarrow}} w$.

Demonstraţie Rezultă din lemele 5.3.1, 5.3.2 şi corolarele acestora.

□

4.4 Gramatici şi analizoare SLR(1)

Este mult prea optimist să credem că putem face analiza sintactică LR(0) pentru gramatici care descriu diverse construcţii din limbajele de programare. În paragraful precedent am văzut că în cazul expresiilor aritmetice, pentru a obţine o gramatică LR(0), trebuie să punem o marcă de sfârşit (\$) la aceste expresii. Dar expresiile aritmetice apar

în diverse contexte în limbajele de programare încât soluţia aceasta nu este viabilă. Dacă renunţăm la marca $ atunci, în automatul LR(0), se obţine cel puţin o stare inconsistentă (cu conflict deplasare/reducere).

Ne propunem în acest paragraf să găsim o modalitate - dacă este posibil - de eliminare a conflictelor într-un automat LR(0). Dacă o anumită stare *t* conţine un conflict *deplasare/reducere*, de pildă articolele $A \to \alpha\bullet$ şi $B \to \beta_1\bullet a\beta_2$, înseamnă că există în gramatică un prefix viabil γ pentru care cele două articole sunt valide:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi Au \underset{dr}{\Rightarrow} \varphi\alpha u = \gamma u$$

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi' Bv \underset{dr}{\Rightarrow} \varphi'\beta_1 a\beta_2 v = \gamma a\beta_2 v$$

Problema este să eliminăm conflictul, adică să găsim o modalitate de alegere în mod determinist a acţiunii care trebuie făcută, *deplasare* sau *reducere*. Aceasta ar putea fi făcută dacă simbolul 'a' nu este primul simbol din cuvântul u, ceea ce înseamnă că a ∉ FOLLLOW(A). În acest caz am putea alege *deplasare* dacă simbolul din banda de intrare este 'a' şi *reducere* dacă acesta ar fi unul din simbolurile din FOLLOW(A).

În cazul unui conflict *reducere/reducere* într-o stare *t*, rezultă existenţa unui prefix viabil γ pentru care articolele diferite $A \to \alpha\bullet$, $B \to \beta\bullet$ din *t* sunt valide:

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi Au \underset{dr}{\Rightarrow} \varphi\alpha u = \gamma u$$

$$S \underset{dr}{\overset{*}{\Rightarrow}} \varphi' Bv \underset{dr}{\Rightarrow} \varphi'\beta v = \gamma v$$

Eliminarea conflictului ar putea fi făcută dacă mulţimile FOLLOW(A) şi FOLLOW(B) sunt disjuncte. Astfel, dacă în banda de intrare urmează un simbol din FOLLOW(A), se face reducere cu regula $A \to \alpha$, dacă acesta este din FOLLOW(B) se face reducerea cu $B \to \beta$, iar în celelalte cazuri se obţine eroare. Din aceste considerente suntem îndreptăţiţi să introducem clasa de *gramatici SLR(1)* ("*simplu LR(1)*") prin:

Definiţia 5.4.1 Fie G o gramatică pentru care automatul LR(0) conţine stări inconsistente (deci G nu este LR(0)). Gramatica G este *gramatică SLR(1)* dacă oricare ar fi starea *t* a automatului LR(0) sunt îndeplinite condiţiile:

1. Dacă $A \to \alpha\bullet$, $B \to \beta\bullet \in t$ atunci FOLLOW(A) ∩ FOLLOW(B) = Φ;
2. Dacă $A \to \alpha\bullet$, $B \to \beta\bullet a\gamma \in t$ atunci a ∉ FOLLOW(A).

Modelul de algoritm de analiză sintactică SLR(1) este acelaşi ca în cel descris în paragraful precedent cu deosebirea că tabela de analiză sintactică se construieşte într-un anume mod. Ea va conţine două zone. Prima, numită *ACŢIUNE*, determină dacă parserul va face deplasare respectiv reducere, în funcţie de starea ce se află în topul stivei şi de simbolul următor din intrare, iar cea de a doua, numită *GOTO*, determină starea ce se va pune în stivă în urma unei reduceri. Următorul algoritm descrie obţinerea tabelei de parsare SLR(1).

Algoritmul 5.4.1 (construcţia tabelei de parsare SLR(1))

Intrare: Gramatica G = (V, T, S, P) augmentată cu S' → S;
Automatul $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$;
Mulţimile FOLLOW(A), A ∈ V (ε se înlocuieşte cu #).

Ieşire: Tabela de analiză SLR(1) compusă din două părţi:
ACŢIUNE(*t*, a), $t \in \mathcal{T}$, a ∈ T ∪ { # }.
GOTO(*t*, A), $t \in \mathcal{T}$, A ∈ V.

Metoda:

```
1. for(t ∈ T)//Se initializeaza tabela de parsare
2.       for (a ∈ T) ACTIUNE(t, a) = "eroare";
```

```
3.       for (A ∈ V) GOTO(t, A) = "eroare";
4. for(t ∈ T}{
5.       for(A → α•aβ ∈ t)
6.         ACTIUNE(t,a)="D g(t, a)";//deplasare in g(t, a)
7.       for(B → γ• ∈ t ) // acceptare sau reducere
8.         if(B == 'S'') ACTIUNE(t, a) = "acceptare";
           else
9.         for(a∈FOLLOW(B))ACTIUNE(t,a)="R B→γ";
         }// endfor
10.      for (A ∈ V) GOTO(t, A) = g(t, A);
   }
```

Toate intrările în tabela SLR(1) sunt iniţial completate cu "*eroare*" (linia 1-3) astfel încât intrările nedefinite în liniile 6, 8, 9, 10 rămân cu această valoare. Să observăm că, dacă gramatica G este SLR(1), atunci tabela dată de algoritmul de mai sus este bine definită în sensul că o tentativă de adăugare a unei valori în partea *ACTIUNE* sau partea *GOTO* se face doar pentru intrările cu valoarea "*eroare*". Algoritmul de analiză sintactică SLR(1) funcţionează pe aceleaşi principii ca şi cel LR(0). Deosebirea este că acţiunile sunt dictate de tabela de analiză SLR(1) încât tranziţiile pe care le execută acesta sunt:

Deplasare: (σ*t*, au#, π) ⊢ (σ*tt'*, u#, π) dacă *ACTIUNE*(*t*, a) = D*t'* ;

Reducere: (σ*t*σ*'t'*, u#, π) ⊢ (σ*tt''*, u#, πp) dacă *ACTIUNE*(*t*, a) = Rp unde p = A → β, |σ*'t'* | = |β| şi *t''*= *GOTO*(*t*, A);

Acceptare: (*t*0*t*, #, π) dacă *ACTIUNE*(*t*,a) = "*acceptare*"; Analizorul se opreşte cu acceptarea cuvântului de analizat iar π este parsarea acestuia (şirul de reguli care s-a aplicat, în ordine inversă, în derivarea extrem dreaptă a lui w).

Eroare: (σ*t*, au#, π) ⊢ *eroare* dacă *ACTIUNE*(*t*,a) = "*eroare*"; Analizorul se opreşte cu respingerea cuvântului de analizat.

Algoritmul 5.4.2 (Analiză sintactică SLR(1))

Intrare: Gramatica G = (V, T, S, P) care este SLR(1) ;
Tabela de parsare SLR(1) (*ACTIUNE, GOTO*);
Cuvântul de intrare w ∈ T*.

Ieşire: Analiza sintactică (parsarea) ascendentă a lui w dacă w ∈ L(G);
eroare, în caz contrar.

Metoda: Se foloseşte stiva St pentru a implementa tranziţiile de mai sus.

```
char ps[] = "w#"; //ps este cuvantul de intrare w
int i = 0; // pozitia curenta in cuvantul de intrare
St.push(t0); // se initializeaza stiva cu t0
while(true) { // se repeta pana la succes sau eroare
      t = St.top();
      a = ps[i] // a este simbolul curent din intrare
      if(ACTIUNE(t,a) == "acceptare") exit("acceptare");
      if(ACTIUNE(t,a) == "Dt'"){
            St.push(t');
            i++; // se inainteaza in w
            }//endif
      else {
            if(ACTIUNE(t,a) == "R A → X1X2…Xm"){
                  for( i = 1; i ≤ m; i++) St.pop();
                  St.push(GOTO(top(STIVA), A));
                  } //endif
            else exit("eroare");
```

```
            }//endelse
        }//endwhile
```

Corectitudinea algoritmului SLR(1) se dovedeşte în acelaşi mod ca şi cea a algoritmului LR(0). Aşa cum am precizat, pentru o gramatică în care automatul LR(0) are stări inconsistente, prin construcţia tabelei de analiză SLR(1) nu am făcut altceva decât să eliminăm conflictele de deplasare/reducere sau de reducere/reducere. Faptul că este corectă strategia propusă rezultă din analiza făcută înainte de a defini gramatica SLR(1). Lăsăm aşadar cititorului, ca un exerciţiu de acum simplu, demonstrarea corectitudinii algoritmului SLR(1).

Teorema 5.4.1 Dacă G este o gramatică SLR(1) atunci, oricare ar fi cuvântul de intrare w, algoritmul de analiză SLR(1) se opreşte cu "*acceptare*" şi obţine la ieşire π dacă şi numai dacă $S \overset{\tilde{\pi}}{\underset{dr}{\Rightarrow}} w$.

□

Exemplul 5.4.1 Să aplicăm algoritmul de analiză sintactică SLR(1) pentru gramatica ce generează expresiile aritmetice:

$S \to E$ $\quad T \to T*F$

$E \to E+T$ $\quad T \to F$

$E \to T$ $\quad F \to (E)$ $\quad F \to a$

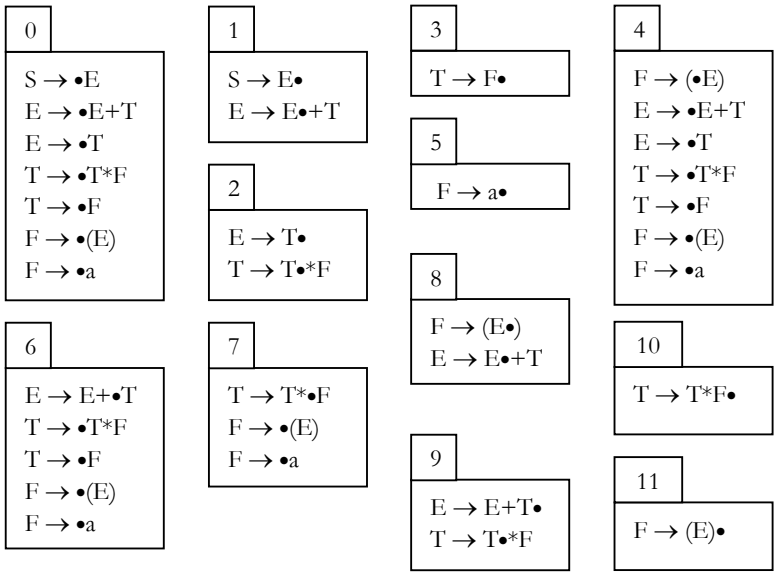


Figura 5.4.1

Tabela de tranziţie a automatului LR(0) este:

g	a	+	*	(	)	E	T	F
0	5			4		1	2	3
1		6						
2			7					
3								
4	5			4		8	2	3
5								
6	5			4			9	3
7	5			4				10

8					11			
9			7					
10								
11								

Stările automatului LR(0) sunt date în figura 5.4.1. Din acest automat se deduce că G nu este LR(0): stările 1, 2, 9 conţin conflict de deplasare/reducere. Pentru a verifica dacă gramatica este SLR(1) avem nevoie de mulţimile FOLLOW(S) şi FOLLOW(E). Se deduce uşor că: FOLLOW(S) = { ε } şi FOLLOW(E) = {+,), ε }. Atunci gramatica este SLR(1) pentru că:

în starea 1: + ∉ FOLLOW(S);

în starea 2: * ∉ FOLLOW(E);

în starea 9: * ∉ FOLLOW(E).

Pentru construcţia tabelei de parsare avem nevoie de mulţimile FOLLOW pentru toţi neterminalii (s-a înlocuit ε cu #):

NETERMINAL	FOLLOW
S	#
E	#, +,)
T	#, +, *,)
F	#, +, *,)

Tabela de analiză SLR(1) pentru acestă gramatică este dată mai jos.

STARE	ACŢIUNE						GOTO		
	a	+	*	(	)	#	E	T	F
0	D5			D4			1	2	3
1		D6				acceptare			
2		R2	D7		R2	R2			
3		R4	R4		R4	R4			
4	D5			D4			8	2	3
5		R6	R6		R6	R6			
6	D5			D4				9	3
7	D5			D4					10
8					D11				
9		R1	D7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Acţiunile analizorului SLR(1) pentru cuvântul de intrare w = a*(a+a) sunt date în tabela următoare:

STIVA	INTRARE	ACŢIUNE	IEŞIRE
0	a*(a+a)#	deplasare	
05	*(a+a)#	reducere	F → a
03	*(a+a)#	reducere	T → F
02	*(a+a)#	deplasare	
027	(a+a)#	deplasare	
0274	a+a)#	deplasare	
02745	+a)#	reducere	F → a
02743	+a)#	reducere	T → F
02742	+a)#	reducere	E → T
02748	+a)#	deplasare	
027486	a)#	deplasare	
0274865	)#	reducere	F → a

0274863	)#	reducere	T → F
0274869	)#	reducere	E → E+T
02748	)#	reducere	
02748(11)	#	reducere	F → (E)
027(10)	#	reducere	T → T*F
02	#	reducere	E → T
01	#	acceptare	

□

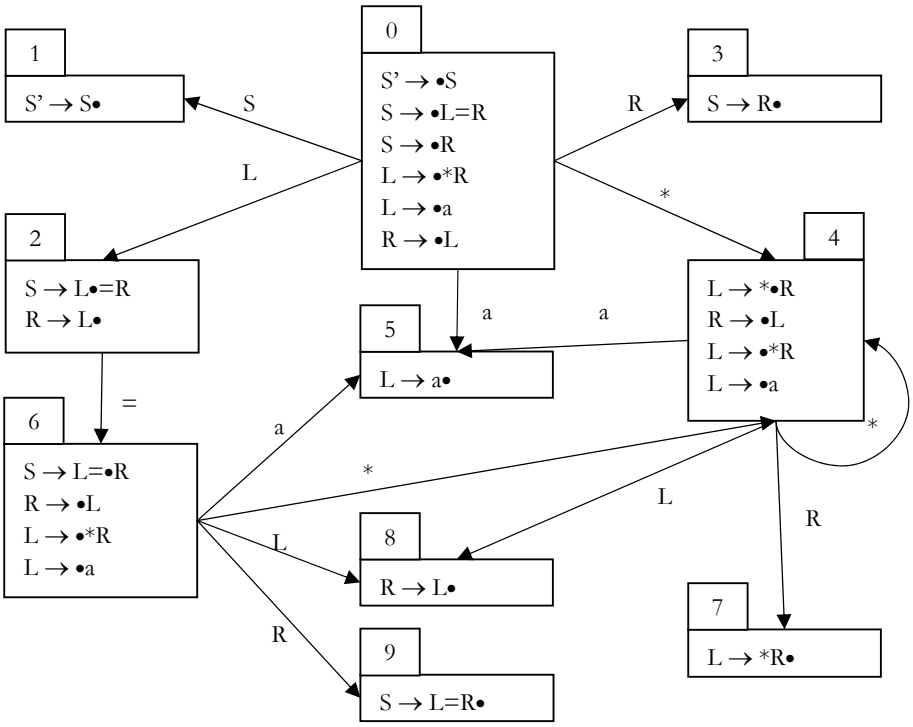


Figura 5.4.2

Vom vedea în paragraful următor că orice gramatică SLR(1) este gramatică LR(1), prin urmare este gramatică neambiguă. Există însă gramatici neambigue (chiar gramatici LR(1)) care nu sunt SLR(1). Vom da în continuare un exemplu de astfel de gramatică.

Exemplul 5.1.2 Fie gramatica dată de producţiile:

$S \to L=R \mid R \qquad L \to *R \mid a \qquad R \to L$

Se poate constata că această gramatică nu este ambiguă. Automatul LR(0) este dat în figura 5.4.2.

În starea 2 există un conflict deplasare/reducere: articolul $S \to L\bullet=R$ defineşte operaţia de deplasare (dacă următorul simbol este =) iar articolul $R \to L\bullet$ defineşte o reducere. Constatăm că FOLLOW(R) = { ε, = } ceea ce înseamnă că gramatica nu este SLR(1) pentru că simbolul '=' este în această mulţime. Vom arăta în paragraful următor că această gramatică este LR(1).

□

4.5 O caracterizare a gramaticilor LR(1)

Definiţia 5.5.1 Fie G = (V, T, S, P) o gramatică independentă de context redusă. Un articol LR(1) pentru gramatica G este o pereche ($A \to \alpha \bullet \beta$, a) unde $A \to \alpha\beta$ este un articol LR(0), iar $a \in FOLLOW(A)$ (se pune # în loc de ε).

Definiţia 5.5.2 Articolul ($A \to \beta_1 \bullet \beta_2$, a) este valid pentru prefixul viabil $\alpha\beta_1$ dacă are loc derivarea

$$S \overset{*}{\underset{dr}{\Rightarrow}} \alpha Au \underset{dr}{\Rightarrow} \alpha\beta_1\beta_2 u$$

iar a = 1:u (a = # dacă u = ε).

Teorema 5.5.1 O gramatică G = (V, T, S, P) este gramatică LR(1) dacă şi numai dacă oricare ar fi prefixul viabil φ, nu există două articole distincte, valide pentru φ, de forma ($A \to \alpha \bullet$, a), ($B \to \beta \bullet \gamma$, b) unde $a \in FIRST(\gamma b)$.

Demonstraţie Să interpretăm mai întâi condiţia (necesară şi suficientă) pentru ca G să fie LR(1). Este de aşteptat ca algoritmul de analiză LR(1) să funcţioneze după aceleaşi principii ca şi cel SLR(1). Asta înseamnă că un articol de forma ($A \to \alpha \bullet$, a) invocă reducere cu regula $A \to \alpha$ dacă a este simbolul curent din intrare, iar un articol de forma ($B \to \beta \bullet a\beta'$, b) invocă deplasare dacă a este la intrare. Aşadar, condiţia ca nici un prefix viabil să nu aibă două articole valide de forma indicată se traduce prin:

1. nu există conflict *deplasare/reducere*. Un astfel de conflict înseamnă două articole ($A \to \alpha \bullet$, a) şi ($B \to \beta \bullet a\beta'$, b) valide pentru acelaşi prefix.
2. nu există conflict *reducere/reducere*. Un astfel de conflict înseamnă două articole complete ($A \to \alpha \bullet$, a) şi ($B \to \beta \bullet$, a) valide pentru acelaşi prefix.

Demonstraţia se face analog cu cea de la caracterizarea LR(0).

□

Pentru a testa dacă o gramatică este LR(1) şi apoi pentru obţinerea tabelei de analiză sintactică LR(1) se construieşte automatul LR(1) care recunoaşte prefixele viabile ale gramaticii. Mai întâi, ca şi în cazul LR(0), să definim o procedură închidere(*I*) care găseşte toate articolele care sunt valide pentru aceleaşi prefixe pentru care sunt valide articolele din *I*.

Algoritmul 5.5.1 (procedura închidere(*I*))

Intrare: Gramatica G = (V, T, S, P);
Mulţimea *I* de articole LR(1) din gramatica G; Mulţimile FIRST.

Ieşire: *I* devine mulţimea tuturor articolelor valide pentru prefixele viabile date iniţial.

Metoda: asemănătoare cu cea de la LR(0).

```
flag = true;
while( flag ) {
      flag = false;
      for ( (A → α•Bβ, a) ∈ I) {
            for ( B → γ ∈ P )
                  for( b ∈ FIRST(βa)){
                        if ( (B → •γ, b)∉ I) {
                              I = I ∪ {(B → •γ, b)};
                              flag = true;
                              }//endif
                        }//endforb
                  }//endforB
            }//endforA
}//endwhile
return I;
```

În continuare, folosind această procedură, descriem algoritmul pentru construirea automatului LR(1).

Algoritmul 5.5.2 (Automatul LR(1))

Intrare: Gramatica G = (V, T, S, P) la care s-a adăugat S' → S;

Ieşire: Automatul determinist LR(1) $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$ ce recunoaşte prefixele viabile şi are ca stări mulţimi de articole LR(1).

Metoda:

```
1. t0 = închidere((S'→•S,#));T={t0};marcat(t0)=false;
2. while(∃t∈T && !marcat(t)){ // marcat(t) = false
3.    for( X ∈ Σ ) {
4.          t' = Φ;
5.          for( (A → α•Xβ,a) ∈ t )
6.             t' = t' ∪ {(B → αX•β,a) | (B → α•Xβ,a) ∈ t};
7.          if( t' ≠ Φ ){
8.             t' = închidere( t' );
9.             if( t' ∉ T ) {
10.               T = T ∪ { t' };
11.               marcat( t' ) = false;
                 }//endif
12.            g(t, X) = t';
            } //endif
      } //endfor
13.   marcat( t ) = true;
   } // endwhile
```

Să dovedim acum corectitudinea algoritmului descris mai sus.

Teorema 5.5.2 Automatul $\mathcal{M}$ construit în algoritmul 5.5.2 este determinist şi $L(\mathcal{M})$ coincide cu mulţimea prefixelor viabile ale lui G. Mai mult, pentru orice prefix viabil γ, $g(t_0,\gamma)$ reprezintă mulţimea articolelor LR(1) valide pentru γ.

□

Automatul LR(1) pentru o gramatică G, se foloseşte pentru a verifica dacă G este LR(1). Conform teoremei de caracterizare LR(1) acest lucru decurge astfel:

- Conflict *reducere/reducere*: Dacă în $\mathcal{T}$ există o stare ce conţine articole de forma
  $(A \to \alpha\bullet, a)$, $(B \to \beta\bullet, a)$ atunci gramatica nu este LR(1);
- Conflict *deplasare/reducere*: Dacă în $\mathcal{T}$ există o stare ce conţine articole de forma
  $(A \to \alpha\bullet, a)$ şi $(B \to \beta_1\bullet a\beta_2, b)$, atunci G nu este LR(1).

O gramatică este LR(1) dacă orice stare $t \in \mathcal{T}$ este liberă de conflicte.

Exemplul 5.5.1 Să considerăm din nou gramatica:

$$S \to L=R \mid R$$
$$L \to *R \mid a$$
$$R \to L$$

şi să construim automatul LR(1) pentru aceasta.

În continuare, articolele ce au aceeaşi primă componentă le notăm condensat prin $(A \to \alpha\bullet\beta, \{a_1, a_2, \ldots, a_n\})$. De exemplu, articolele $(A \to \alpha\bullet\beta, a)$, $(A \to \alpha\bullet\beta, b)$, se scriu $(A \to \alpha\bullet\beta, \{a, b\})$.

Am văzut în paragraful precedent că această gramatică nu este SLR(1). Dacă analizăm stările automatului LR(1), prezentate în figura 5.5.1, constatăm că gramatica este LR(1): toate stările sunt libere de conflict.

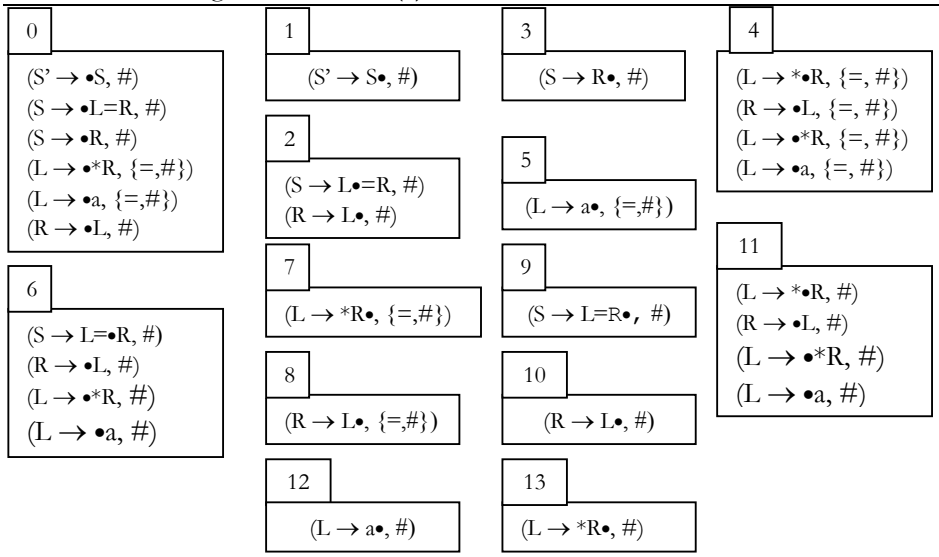


Figura 5.5.1

Tabela de tranziţie a automatului este prezentată în continuare.

g	a	=	*	S	L	R
0	5		4	1	2	3
1						
2		6				
3						
4	5		4		8	7
5						
6	12		11		10	9
7						
8						
9						
10						
11	12		11		10	13
12						
13						

4.6 Analiză sintactică LR(1) şi LALR(1)

Algoritmul de analiză sintactică LR(1) este, în esenţă, acelaşi cu algoritmul de analiză sintactică SLR(1). Ceea ce îl distinge este modul de construire a tabelei de analiză cu cele două părţi, *ACŢIUNE* şi *GOTO*. Vom indica în continuare construirea acestei tabele, pornind de la automatul LR(1).

Algoritmul 5.6.1 (Construcţia tabelei de analiză LR(1))

Intrare: Gramatica LR(1) G =(V, S, T, P) augmentată cu S' → S.
Automatul LR(1) $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$.

Ieşire: Tabela de analiză cu componentele:
ACŢIUNE(t, a), $t \in \mathcal{T}$, a ∈ T ∪ {#}
GOTO(t, A), $t \in \mathcal{T}$, A ∈ V.

Metoda:

```
for(t ∈ T){ //Se initializeaza tabela de parsare
```

```
      for (a ∈ T) ACTIUNE(t, a) = "eroare";
      for (A ∈ V) GOTO(t, A) = "eroare";
}
for(t ∈ T}{
      for((A → α•aβ, L) ∈ t)
        ACTIUNE(t,a)="D g(t, a)";//deplasare in g(t, a)
      for((B → γ•, L) ∈ t ) {
        for(c ∈ L){
            if(B == 'S'') ACTIUNE(t, c) = "acceptare";
                  else
              ACTIUNE(t,c)="R B→γ";//reducere cu B→γ
        }// endforc
      }// endforB
} // endfor
for (A ∈ V) GOTO(t, A) = g(t, A);
```

Aşadar, stările automatului LR(1) determină acţiunea analizorului sintactic :

- Dacă starea *t* din topul stivei conţine un articol cu terminalul 'a' după punct şi la intrare, se produce o deplasare, anume starea *g*(*t*,a)se adaugă în stivă;
- Dacă starea din topul stivei conţine un articol complet (A → β•, a) unde 'a' este terminalul curent din intrare, se produce reducere cu producţia A → β. În cazul în care A → β = S' → S, obţinem *acceptare*;
- În celelalte cazuri se obţine o *eroare*.

Exemplul 5.6.1 Să construim după algoritmul tocmai descris tabela de analiză pentru gramatica din exemplul 5.4.1:

0: S' → S 1: S → L=R 2: S → R
3: L → *R 4: L → a 5: R → L

STARE	ACŢIUNE				GOTO		
	a	=	*	#	S	L	R
0	D5		D4		1	2	3
1				acceptare			
2		D6		R5			
3				R2			
4	D5		D4			8	7
5		R4		R4			
6	D12		D11			10	9
7		R3		R3			
8		R5		R5			
9				R1			
10				R5			
11	D12		D11			10	13
12				R4			
13				R3			

Aplicarea algoritmului (de deplasare/reducere) folosind această tabelă de analiză, asupra cuvintelori **aa respectiv **a=a, este ilustrată în tabelele următoare:

STIVA	INTRARE	ACŢIUNE	IEŞIRE
0	**aa#	deplasare D4	
04	*aa#	deplasare D4	
044	aa#	deplasare D5	
0445	a#	eroare	
STIVA	**INTRARE**	**ACŢIUNE**	**IEŞIRE**

0	**a=a#	deplasare D4	
04	*a=a#	deplasare D4	
044	a=a#	deplasare D5	
0445	=a#	reducere R4	L → a
0448	=a#	reducere R5	R → L
0447	=a#	reducere R3	L → *R
048	=a#	reducere R5	R → L
047	=a#	reducere R3	L → *R
02	=a#	deplasare D6	
026	a#	deplasare D12	
026(12)	#	reducere R4	L → a
026(10)	#	reducere R5	R → L
0269	#	reducere R1	S → L=R
01	#	acceptare	

Să observăm că automatul LR(1) construit în paragraful precedent, pentru gramatica care tocmai am discutat-o, are un număr mai mare de stări decât numărul stărilor automatului LR(0) pentru această gramatică. Unele din stările automatului LR(1) au ceva în comun:

starea 5: (L → a•, {=,#}) cu starea 12: (L → a•, #)

starea 4: (L → *•R, {=,#}), (R → •L, {=,#}), (L → •*R, {=,#}), (L → •a, {=,#}) cu starea 11: (L → *•R, #), (R → •L, #), (L → •*R, #), (L → •a, #)

starea 7: (L → *R•, {=,#}) cu starea 13: (L → *R•, #)

starea 8: (R → L•, {=,#}) cu starea 10: (R → L•, #)

Aceste stări au respectiv prima componentă a articolelor aceeaşi.

Definiţia 5.6.1 Fie t o stare în automatul LR(1) al unei gramatici G. *Nucleul* acestei stări este mulţimea articolelor LR(0) care apar ca prime componente în articolele LR(1) din t.

Defininiţia 5.6.2 Două stări t_1 şi t_2 ale automatului LR(1) pentru gramatica G se numesc *echivalente* dacă ele au acelaşi nucleu.

Cum fiecare stare a automatului LR(1) este o mulţime de articole LR(1), având două stări t_1 şi t_2 putem să vorbim de $t_1 \cup t_2$. De pildă, dacă

$$t_1 = \{(L \to *R\bullet, \{=, \#\})\},\ t_2 = \{(L \to *R\bullet, \#)\}$$

atunci $t_1 \cup t_2 = t_1$ pentru că $t_2 \subset t_1$.

Definiţia 5.6.3 Fie G gramatică LR(1) şi $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$ automatul LR(1) corespunzător. Spunem că gramatica G este LALR(1) (**L**ook **A**head **LR(1)**) dacă oricare ar fi perechea de stări echivalente t_1, t_2 din automatul LR(1), starea $t_1 \cup t_2$ este liberă de conflicte.

Am introdus aceste noţiuni în scopul de a simplifica, atunci când este posibil, tabela de analiză LR(1), în sensul de a restrânge numărul liniilor tabelei. Acest lucru este posibil dacă prin reuniunea a două stări echivalente nu se introduc conflicte. În acest mod

putem obţine o tabelă de analiză de tip LR(1) dar cu numărul de linii (stări ale automatului) egal cu cel al tabelei SLR(1), adică, cu cel al numărului de stări LR(0). În exemplul considerat mai sus avem:

$4 \cup 11 = 4 \qquad 5 \cup 12 = 5 \qquad 7 \cup 13 = 7 \qquad 8 \cup 10 = 8$

Aşadar, gramatica în discuţie este LALR(1) pentru că nu s-au obţinut stări cu conflicte. În continuare se reconstruieşte automatul LR(1) prin reuniunea stărilor echivalente şi se construieşte tabela de analiză pornind de la acest nou automat LR(1). Mai facem observaţia că, dacă stările t_1 şi t_2 sunt echivalente, atunci şi stările $g(t_1, X)$, $g(t_2, X)$ sunt echivalente, pentru orice X din Σ. Să descriem în întregime acest algoritm.

Algoritmul 5.6.2 (Construcţia tabelei de analiză LALR(1))
Intrare: Gramatica G = (V, T, S, P) augmentată cu S' → S;
Ieşire: Tabela de analiză LALR(1) (*ACŢIUNE* şi *GOTO*).
Metoda:

1. Se construieşte automatul LR(1), $\mathcal{M} = (\mathcal{T}, \Sigma, g, t_0, \mathcal{T})$ folosind Algoritmul 5.5.2. Fie $\mathcal{T} = \{t_0, t_1, \ldots, t_n\}$. Dacă toate stările din $\mathcal{T}$ sunt libere de conflict, urmează 2, altfel algoritmul se opreşte: gramatica nu este LR(1).
2. Se determină stările echivalente din $\mathcal{T}$ şi, prin reuniunea acestora, se obţine o nouă mulţime de stări $\mathcal{T}' = \{t'_0, t'_1, \ldots, t'_m\}$
3. Dacă în $\mathcal{T}'$ există stări ce conţin conflicte, algoritmul se opreşte: gramatica G nu este LALR(1). Altfel se continuă cu 4.
4. Se construieşte automatul $\mathcal{M}' = (\mathcal{T}', \Sigma, g', t'_0, \mathcal{T}')$, unde $\forall t' \in \mathcal{T}'$:
   Dacă $t' \in \mathcal{T}$ atunci $g'(t', X) = g(t, X)$, $\forall X \in \Sigma$;
   Dacă $t' = t_1 \cup t_2 \cup \ldots$, $t_1, t_2, \ldots \in \mathcal{T}$, atunci
   $g'(t', X) = g(t_1, X) \cup g(t_2, X) \cup \ldots$.
5. Se aplică algoritmul 5.6.1 pentru construirea tabelei LR(1) pornind de la automatul $\mathcal{M}'$. Tabela obţinută se numeşte tabela LALR(1) pentru gramatica G.

Exemplul 5.6.2. Aplicând algoritmul de mai sus automatului $\mathcal{M}$ din exemplul 5.5.1, obţinem automatul $\mathcal{M}'$ dat de tabela de tranziţie de mai jos (stările au fost redenumite prin 0, 1, …, 9).

g	**a**	**=**	*****	**S**	**L**	**R**
0	5		4	1	2	3
1						
2		6				
3						
4	5		4		8	7
5						
6	5		4		8	9
7						
8						
9						

Pornind de la acest automat, aplicâd algoritmul de construcţie al abelei LR(1), obţinem tabela de parsare LALR(1) pentru gramatica dată:

STARE	**ACŢIUNE**				**GOTO**		
	a	=	*	#	S	L	R
0	D5		D4		1	2	3

1				acceptare			
2		D6		R5			
3				R2			
4	D5		D4			8	7
5		R4		R4			
6	D5		D4			8	9
7		R3		R3			
8		R5		R5			
9				R1			

Analiza cuvântului w = **a=a este ilustrată în tabela următoare:

STIVA	**INTRARE**	**ACŢIUNE**	**IEŞIRE**
0	**a=a#	deplasare D4	
04	*a=a#	deplasare D4	
044	a=a#	deplasare D5	
0445	=a#	reducere R4	$L \rightarrow a$
0448	=a#	reducere R5	$R \rightarrow L$
0447	=a#	reducere R3	$L \rightarrow *R$
048	=a#	reducere R5	$R \rightarrow L$
047	=a#	reducere R3	$L \rightarrow *R$
02	=a#	deplasare D6	
026	a#	deplasare D5	
0265	#	reducere R4	$L \rightarrow a$
0268	#	reducere R5	$R \rightarrow L$
0269	#	reducere R1	$S \rightarrow L=R$
01	#	acceptare	

Dacă vom compara tabela LR(1) cu tabela LALR(1) pentru gramatica din exemplul dat, vom observa că tabela LALR(1) coincide cu liniile 0-9 din tabela LR(1) cu deosebirea că în locul stările 10, 11, 12 şi 13 apar stările 8, 4, 5, 7 respectiv. Acest fapt face ca erorile, în unele cuvinte, să fie semnalate mai târziu în analiza LALR(1) decât s-ar face în analiza LR(1). În exemplul precedent, pentru tabela LR(1) avem *ACŢIUNE*(12, =) = *eroare* pe când *ACŢIUNE*(5, =) = R4 încât, pentru că stările 5 şi 12 sunt echivalente, în tabela LALR(1) rămâne doar *ACŢIUNE*(5, =) = R4. Cititorul este invitat să verifice că în analiza LR(1), pentru cuvântul de intrare *a=a=#, se obţine o eroare la configuraţia (026(12), =#, π), pe când algoritmul LALR(1) va ajunge, pentru acelaşi cuvânt, în configuraţia (0265, =#, π) care are tranziţie în configuraţia (0268, =#, π4) apoi în (0269, =#, π45) şi abia aceasta din urmă produce eroarea.

□

Avantajul metodei LALR(1) constă în aceea că dimensiunea tabelei LALR(1) este aceeaşi cu cea a tabelei SLR(1); tabela LR(1) pentru o gramatică care descrie părţi ale sintaxei unui limbaj de programare poate fi foarte mare!
Să arătăm în continuare că există gramatici LR(1) care nu sunt LALR(1).

Exemplul 5.6.3 Fie gramatica:

$S \rightarrow aAb \mid bAd \mid aBd \mid bBc$

$A \rightarrow e \quad B \rightarrow e$

Stările automatul LR(1) pentru această gramatică sunt cele din figura 5.6.1.

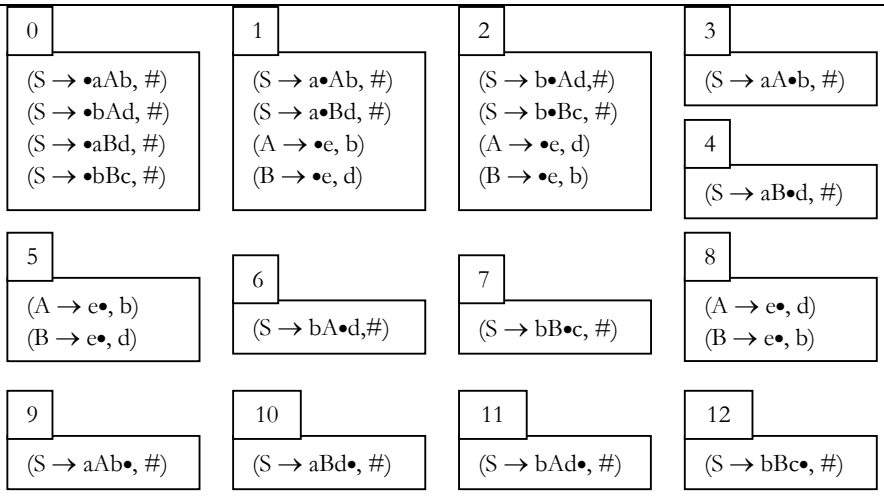


Figura 5.6.1

Să observăm că toate stările automatului LR(1) sunt libere de conflicte; gramatica este LR(1). Stările 5 şi 8 sunt stări echivalente. Constatăm că $5 \cup 8 = = \{(A \rightarrow e\bullet, \{b, d\}), (B \rightarrow e\bullet, \{b, d\})\}$, iar această nouă mulţime conţine conflicte de reducere/reducere, ceea ce înseamnă că gramatica considerată nu este gramatică LALR(1).

5 Generatoare de analizoare sintactice

Pentru a uşura munca implementatorilor de limbaje au fost concepute instrumente care generează automat analizoare (lexicale sau sintactice). Printre cele mai populare instrumente de acest fel se numără sistemul YACC ("**Y**et **A**nother **C**ompiler **C**ompiler") conceput şi realizat în 1975 de S. C. Johnson la Bell Laboratory în USA. Produsul a fost inclus în sistemul de operare UNIX şi a fost folosit la scară industrială pentru implementarea multor limbaje. Un alt produs, distribuit în sistemele de operare Linux este Bison, compatibil în întregime cu YACC, scris de Robert Corbett şi Richard Stallmann. În 1990 a apărut versiunea pentru MS-DOS a produsului YACC, numit PCYACC, dezvoltat de Abraxax Software Inc.

Există astăzi numeroase astfel de generatoare, construite de grupuri de cercetare din universităţi sau din societăţi specializate pe software. Enumerăm câteva din acestea împreună cu resursele web corespunzătoare.

La adresa http://dinosaur.compilertools.net/ se găsesc informaţii (inclusiv documentaţie) despre Lex, Yacc, Flex, şi Bison.

La adresa http://epaperpress.com/lexandyacc/index.html se găseşte un ghid compact care este util celor care vor să utilizeze lex şi yacc.

Pagina oficială destinată proiectului GNU Flex (Fast lexical analyser generator) este http://www.gnu.org/software/flex/. Aşa cum se preciza şi în capitolul 3.5, Flex este un instrument pentru generare de programe care realizează pattern-matching pentru texte iar împreună cu Bison constituie perechea de instrumente (www.gnu.org/software/bison/) cel mai des utilizată pentru scrierea compilatoarelor. Bison, ca şi Yacc, este un generator de parsere care converteşte descrierea unei gramatici context – free LALR(1) într-un program C care realizează analiza sintactică în gramatica descrisă.

Iată şi alte instrumente de acest fel:

- *Spirit C++ Parser Framework* - http://spirit.sourceforge.net/

Spirit este un instrument ce implementează parsarea recursiv descendentă pornind de la descrierea EBNF(Extended Backus Normal Form) a unei gramatici. Implementarea este orientată obiect în C++.

- *BtYacc: BackTracking Yacc* - http://www.siber.com/btyacc/

BtYacc este o versiune modificată a lui Yacc care suportă backtraking. Pentru corectarea bug-urilor vizitaţi http://www.vendian.org/mncharity/dir3/btyacc/

- *LEMON* - http://www.hwaci.com/sw/lemon/

LEMON este un generator de parsere LALR(1), este open-source, produce cod C şi pretinde că produce un parser mai rapid decât yacc/bison.

- *SYNTAX*: http://www-rocq.inria.fr/oscar/www/syntax/

SYNTAX cuprinde un set de instrumente pentru proiectarea şi implementarea părţii front-end a unui compilator. Are toate capabilităţile lex şi yacc iar partea de procesare a erorilor este mult dezvoltată.

- *PCCTS* : http://www.polhode.com/pccts.html

PCCTS este un generator pentru parsare recursiv descendentă LL(k). Proiectul s-a transformat în ANTLR.

- *ANTLR*: http://www.antlr.org

ANTLR (ANother Tool for Language Recognition) este un instrument pentru construcţia compilatoarelor care suportă descrierea acţiunilor în Java, C# sau C++. ANTLR conţine module pentru construcţia arborilor, traversarea arborilor, construcţia translatoarelor. ANTLR implementează parsarea LL(k) şi este în domeniul public.

- *Visual Parse++ 5.0* - http://www.sand-stone.com/

Visual Parse permite proiectarea vizuală a analizoarelor lexicale şi sintactice şi utilizarea lor în aplicaţii C, C++, C#, Delphi, Java, sau Visual Basic sub sistemele de operare UNIX, Linux şi Windows. Este un produs comercial.

- *AnaGram*: - http://www.parsifalsoft.com/

AnaGram este un genarator de parsere LALR(1) care produce C/C++ ce poate fi compilat pe orice platformă şi poate fi executat sub Win9x/NT. Este un produs comercial dar firma oferă şi o variantă gratuită. De asemenea, este disponibil XIDEK (Extensible Interpreter Development Kit) ce poate fi folosit pentru proiectarea şi implementarea interpreterelor.

- *Yacc++(R)* - http://world.std.com/~compres/

Yacc++(R) este o rescriere orientată obiect a instrumentelor Lex şi Yacc pentru C++.

- *LLgen* - http://www.cs.vu.nl/~ceriel/LLgen.html

LLgen implementează eficient parsere recursiv descendente ale gramaticilor ELL(1). Dispune de facilităţi statice şi dinamice pentru tratarea ambiguităţilor.

- *Elkhound*: - http://www.cs.berkeley.edu/~smcpeak/elkhound/

Este un generator de parsere bazat pe tehnica GLR; fiind dezvoltat la Universitatea Berkeley este open source.

- *Grammatica* - http://www.nongnu.org/grammatica/

Grammatica este un generator de parsere LL(k) pentru C# şi Java. Este distribuit sub licenţa GNU şi suportă gramatici cu un număr nelimitat de token-uri look-ahead. De asemenea, are un modul puternic pentru recuperarea automată a erorilor.

- *jay* este o variantă Java pentru Yacc.
  http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/
- *RDP* este un generator de parsere pentru gramatici LL(1) cu atribute.
  http://www.dcs.rhbnc.ac.uk/research/languages/projects/rdp.shtml
- *TPG* - http://christophe.delord.free.fr/en/tpg/

TPG (Toy Parser Generator) este un generator care, pentru o gramatică cu atribute produce un parser recursiv descendent în limbajul Python.

- *ProGrammar* - http://www.programmar.com/main.shtml

ProGrammar este un mediu vizual pentru proiectarea de parsere ce sunt independente de platforma şi de limbajul de programare.

- *GOLD Parser* - http://www.devincook.com/goldparser/

GOLD (Grammar Oriented Language Developer) este un generator care implementează parsarea LALR(1) într-o nouă manieră (faţă de yacc de exemplu): descrirea gramaticii este analizată de un modul – builder – care creează tabela de parsare ce este salvată într-un fişier separat, independent de parser. Acesta este încărcat de motorul de parsare care este disponibil pentru limbajele Java, C# (.NET), ActiveX.

- *Win32 Lex/YACC* - http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html Flex şi Bison pentru platforma Win32.
- *Depot4* - http://www.math.tu-dresden.de/wir/depot4/

Depot4 este un generator de translator dezvoltat la Universitatea din Dresda.

- *IParse* - http://home.planet.nl/~faase009/MM.html

Program ce implementează un parser greedy backtraking. Are la intrare un fişier ce conţine descrierea unei gramatici şi un altul ce conţine fraza de parsat în conformitate cu gramatica dată. Produce arborele abstract al frazei.

- *Styx* - http://www.speculate.de/styx/

Styx este un generator de scaner şi parser LALR(1). Este disponibil atât sub Lynux, cât şi sub Windows.

- *YaYacc(Yet another Yacc)* acceptă gramatici yacc şi produce cod C++. http://www.gradsoft.com.ua/eng/Products/YaYacc/yayacc.html
- *Rie* - http://www.is.titech.ac.jp/~sassa/lab/rie-e.html

Rie este un generator de compilator (front end) bazat pe o clasă de gramatici LR cu atribute numită ECLR-attributed grammar. Generatorul este scris în C, este open source şi este cu siguranţă o extensie a sistemelor Yacc/Bison.

- *Coco/R,* http://www.ssw.uni-linz.ac.at/Research/Projects/

Compiler.html. Coco/R este un generator de scaner şi parser LL(1). Sunt disponibile versiuni Java, C#, Oberon, C, Pascal, Modula2.

- *PRECC* - http://www.afm.sbu.ac.uk/precc/

PREC(PREttier Compiler-Compiler) este un generator de parser pentru gramatici dependente de context care generează cod C. Este disponibil sub Unix sau sub MS-Dos.

- *SGLR*: - http://www.cwi.nl/projects/MetaEnv/sglr/

SGLR (Scannerless Generalized LR) este un generator de parser LR care nu utilizează scanarea: tabela de parsare generată de la o definiţie a sintaxei conţine informaţii suficiente pentru a realiza analiza sintactică.

- *SLK* - http://home.earthlink.net/~slkpg/

Este un generator de parser LL(k) ce produce cod C, C++, C#, şi Java. Disponibil gratuit pentru Unix şi Windows.

- *YooLex (Yet another Object-Oriented Lex)*

http://yoolex.sourceforge.net/ . Produce cod C++ compatibil cu STL.

- *Happy* - http://haskell.cs.yale.edu/happy/ Generator pentru Haskell.
- *VLCC (Visual Languages Compiler-Compiler)*

www.dmi.unisa.it/people/costagliola/www/home/ricerca/vlcc/vlcc.htm

- *Hapy* - http://hapy.sourceforge.net/ generează un parser scris în C++ folosind descrierea sintaxei cu o interfaţă similară cu EBNF.
- *CppCC (C++ Compiler Compiler)* - http://cppcc.sourceforge.net/

Este un generator de scaner şi parser LL(k).

- *ClearParse* - http://www.clearjump.com/products/ClearParse.html

Parser comercial.

- *TPLex/Yacc for Delphi 3* - http://www.17slon.com/gp/gp/tply.htm
- *JB2CSharp* - http://sourceforge.net/projects/jb2csharp/

Este un proiect al Universităţii Boulder – Colorado de a porta instrumentele Java-Bison/Flex pentru platforma .NET: acţiunile pot fi scrise în C#.

- *oolex* http://www.inf.uos.de/alumni/bernd/oolex/

oolex (object-oriented lexer) este un generator pentru analiza lexicală construit în manieră orientată obiect. Există şi un succesor al lui oolex – lolo care se găseşte la http://www.inf.uos.de/alumni/bernd/lolo/index.html . De altfel, împreună cu oops (http://www.inf.uos.de/alumni/bernd/oops/) aceste sisteme fac parte dintr-un proiect dezvoltat la Universitatea din Osnabruck privind construcţia compilatoarelor folosind programarea orientată obiect în Java.

- *EAG* - http://www.cs.ru.nl/~kees/eag/

Acest sistem foloseşte formalismul Extended Affix Grammar (EAG) care descrie sintaxa şi semantica limbajelor de programare.

- *Bison++, Flex++*: http://www.kohsuke.org/flex++bison++/

Sunt variantele Bison, Flex care produc cod C++.

5.1 Utilizarea generatorului de parsere YACC

YACC poate genera un "traducător" în modul următor: Specificaţiile sintactice precum şi unele elemente semantice sunt incluse într-un fişier de intrare pentru YACC. Acest fişier are în general extensia ".y". Linia de comandă (în UNIX):

yacc nume_fisier.y

produce, plecând de la specificaţiile cuprinse în fişierul de intrare nume_fisier.y, un program scris în limbaj C care implementează metoda de analiză sintactică LALR(1) pentru gramatica respectivă – în speţă funcţia yyparse(). Fişierul de intrare (programul sursă) pentru YACC are structura următoare:

Declaraţii
%%
Reguli
%%
Rutine C

Partea *Declaraţii* a fişierului sursă este opţională şi poate cuprinde două secţiuni. O primă secţiune cuprinde între delimitatorii %{ şi %} declaraţii în limbajul C pentru variabilele care se utilizează în regulile de traducere sau în procedurile din cea de-a treia parte a fişierului. Aşadar, textul cuprins între %{ şi %} se copie nealterat în fişierul C produs de YACC. A doua secţiune a acestei prime părţi conţine declaraţii ale unităţilor lexicale ale gramaticii, declaraţii de asociativitate şi precedenţă a operatorilor, declaraţii ale tipurilor de date pentru valorile semantice ale simbolurilor gramaticii.
Simbolurile terminale ale gramaticii – unităţile lexicale – cu excepţia celor formate dintr-un singur caracter (precum +, * etc.), trebuiesc declarate. Aceste simboluri sunt reprezentate în YACC prin nişte coduri numerice; funcţia yylex transmite codul unităţii respective funcţiei yyparse. Programatorul nu trebuie să ştie aceste coduri numerice, ele sunt generate automat folosind opţiunea –d la lansarea lui YACC şi sunt trecute într-un fişier nume.tab.h (sau yytab.h sub DOS).Unităţile lexicale se definesc prin linii de forma:

`% token` <nume _unitate _lexicală>

şi pot fi utilizate în celelalte două părţi ale fişierului de intrare. Tot în această secţiune se poate introduce simbolul de start al gramaticii, în cazul în care acesta nu este partea stângă a primei reguli sintactice:

`start` <nume _simbol _de _start>

Alte linii care pot fi incluse în această secţiune sunt:

`type` - pentru definirea tipului;
`left` - pentru asociativitate stânga a operatorilor;
`right` - pentru asociativitate dreapta a operatorilor;
`prec` - pentru precizarea precedenţei operatorilor;
`nonassoc` - pentru declaraţiile de neasociativitate.

Simbolurile gramaticii pot avea valori semantice. În mod implicit, aceste valori sunt valori întregi si sunt specificate în YYSTYPE. Pentru a specifica alt tip, în partea de declaraţii C se adaugă o directivă define pentru YYSTYPE:

`#define YYSTYPE` <nume tip>

declară tipul valorilor semantice pentru simbolurile gramaticii ca fiind <nume tip>. Utilizarea de tipuri diferite pentru simboluri diferite se realizează prin specificarea acestor tipuri într-o declaraţie union (specifică YACC) şi declararea tipului simbolurilor cu type. Iată un exemplu:

```
%union {
       int intval;
```

```
        double doubleval;
        symrec *tptr;
}
type <intval> INT
type <doubleval> REAL
type <tptr> ID
```

Prin aceste declaraţii se specifică tipul valorilor pentru token-urile INT, REAL, ID ca fiind respectiv int, double, pointer la symrec (pointer în tabela de simboluri).

Partea a doua a fişierului de intrare, *Reguli*, este partea obligatorie. Aici se specifică regulile gramaticii care descriu sintaxa pe care dorim să o verificăm cu acest analizor. Fiecare regulă conţine:

- partea stângă;
- partea dreaptă;
- partea de acţiune.

Partea stângă trebuie să conţină un neterminal al gramaticii, iar partea dreaptă şirul format din terminali şi neterminali corespunzător unei reguli. Cele două părţi sunt despărţite prin ":" (două puncte). Partea de acţiune, cuprinsă între{ şi }, conţine un text scris în limbajul C care va fi inclus în analizor şi reprezintă operaţiile care se execută atunci când analizorul realizează o reducere cu regula specificată. O regulă se termină prin ";" (punct virgulă). Dacă sunt mai multe reguli care au aceeaşi parte stângă, acestea se scriu o singură dată şi părţile drepte se despart prin "|". Iată, spre exemplu, cum se scriu regulile gramaticii care descrie expresiile aritmetice:

```
expr  : NUMAR
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr')'
;
```

Aici, `expr` este numele neterminalului care defineşte expresia aritmetică, terminalii +, -, *, /, (,) se pun între apostrof, iar `NUMAR` reprezintă token-ul (unitatea lexicală) număr (ce trebuie declarat în secţiunea declaraţii). Dacă dorim ca analizorul pe care-l construim să realizeze şi acţiuni semantice, de pildă să evalueze expresiile (pentru şirul 25+15, pe lângă faptul că verifică sintaxa, raportează şi rezultatul adunării, valoarea 40), se adaugă la reguli partea de acţiune:

```
expr  : NUMAR     {$$ = $1;}
| expr '+' expr   {$$ = $1+$3;}
| expr '-' expr   {$$ = $1-$3;}
| expr '*' expr   {$$ = $1*$3;}
| expr '/' expr   {$$ = $1/$3;}
| '(' expr')'     {$$ = $2;}
;
```

În partea de acţiune se scriu instrucţiuni în limbajul C. Aici, `$$` reprezintă valoarea unui atribut al neterminalului `expr` din stânga regulii sintactice iar `$i` reprezintă valoarea atributului celui de-al i-lea simbol din partea stângă a regulii.

Ultima parte a fişierului de intrare conţine rutine scrise în limbaj C care se includ nealterate în analizorul obţinut de YACC. În această parte trebuie furnizat un analizor lexical, adică o funcţie `yylex()` precum şi apelul (în programul principal) la funcţia `yyparse()` pe care o creează YACC-ul.

Exemplul 6.1.1

```
/* Exemplul 1 de fisier intrare YACC */
%{
int yylex(void);
void yyerror(char* s);
```

```
%}
%token ID
%token WHILE
%token BEGIN
%token END
%token DO
%token IF
%token THEN
%token ELSE
%token SEMI
%token ASSIGN
%start prog
%%
prog: stmlist
    ;
stm: ID ASSIGN ID
   | WHILE ID DO stm
   | BEGIN stmlist END
   | IF ID THEN stm
   | IF ID THEN stm ELSE stm
   ;
stmlist: stm
       | stmlist SEMI stm
       ;
%%
```

Dacă se compilează acest fişier cu YACC, acesta construieşte automatul LALR(1) pentru gramatica descrisă în secţiunea Reguli. Această gramatică are 3 neterminali: prog (simbolul de start al gramaticii), stm şi stmlist. Terminalii sunt declaraţi prin directiva `%token`. YACC raportează conflictele de deplasare – reducere şi cele de reducere-reducere dacă există. Aceste conflicte sunt rezolvate de către YACC astfel:

- conflictul de deplasare-reducere este rezolvat în favoarea deplasării;
- conflictul de reducere-reducere este rezolvat în favoarea reducerii cu regula care apare prima în lista de reguli

Pentru descrierea precedentă este raportat un singur conflict de deplasare reducere (este obţinut datorită celor două forme ale instrucţiunii IF). Automatul LALR(1) obţinut este vizibil dacă în linia de comandă pentru lansarea YACC – ului se scrie opţiunea –v (pentru a afla ce opţiuni pot fi folosite lansaţi YACC –h). Iată cum se descrie automatul în exemplul dat:

```
-*-=-*-=-*-=-*-=-*-  LALR PARSING TABLE   -*-=-*-=-*-=-*-=-*-
+----------------------     STATE 0     --------------------+
+ CONFLICTS:
+ RULES:
        $accept : ^prog $end
+ ACTIONS AND GOTOS:
             ID : shift & new state 4
          WHILE : shift & new state 5
          BEGIN : shift & new state 6
             IF : shift & new state 7
                : error
           prog : goto state 1
        stmlist : goto state 2
            stm : goto state 3
+----------------------     STATE 1     --------------------+
+ CONFLICTS:
+ RULES:
        $accept :  prog^$end
+ ACTIONS AND GOTOS:
```

```
           $end : accept
                : error
+----------------------     STATE 2     --------------------+
+ CONFLICTS:
+ RULES:
           prog :  stmlist^    (rule 1)
        stmlist :  stmlist^SEMI stm
+ ACTIONS AND GOTOS:
           SEMI : shift & new state 8
                : reduce by rule 1
.............................................................
+----------------------     STATE 21     -------------------+
+ CONFLICTS:
+ RULES:
            stm :  IF ID THEN stm ELSE stm^    (rule 6)
+ ACTIONS AND GOTOS:
                : reduce by rule 6
====================      SUMMARY      ====================
.............................................................
-*-=-*-=-*-=-*-=-*-      END OF TABLE       -*-=-*-=-*-=-*-
```

□

Exemplul 6.1.2

```
/* Intrare YACC pentru expresii aritmetice de forma a*(a+a) */
%{
#define QUIT      ((double) 101010)
%}
%start S

%%
S:                { prompt(); }
   | S '\n'       { prompt(); }
   | S E '\n'     { if ($2 == QUIT) {
                        return(0);
                        } else {
                          printf("Expresia este corecta.\n");
                          prompt();
                        }
                  }
  ;
E : E '+' T       {printf(" E->E+T\n");}
  | E '-' T       {printf(" E->E-T\n");}
  | T             {printf(" E->T\n");}
  ;
T : T '*' F       {printf(" T->T*F\n");}
  | T '/' F       {printf(" T->T/F\n");}
  | F             {printf(" T->F\n");}
  ;
F : '(' E ')'     {printf(" F->(E)\n");}
  | 'a'           {printf(" F->a\n");}
  ;
%%

#include <stdio.h>
#include <ctype.h>

int main(){
printf("\n*********************************************\n");
printf("**                                           **\n");
printf("**    Parser pentru expresii aritmetice      **\n");
printf("**   cu operanzi a si operatori +,-,*,/,( )   **\n");
printf("**     La promterul READY>                   **\n");
```

```
printf("**   Introduceti o expresie. Ex. a*(a+a)-a    **\n");
printf("**     Pentru iesire tastati quit             **\n");
printf("**                                            **\n");
printf("\n*********************************************\n");
yyparse();
}
yyerror(s){
  printf("Expresia este incorecta\n");
}
yylex(){
  int c;
  while ((c=getchar()) == ' ' || c == '\t' )
          ;
  if(c==EOF)
    return 0;
  if (c == 'Q' || c == 'q')
         if ((c=getchar()) == 'U' || c == 'u')
             if ((c=getchar()) == 'I' || c == 'i')
                 if ((c=getchar()) == 'T' || c == 't') {
                     yylval = QUIT;
                     return( EOF );
                 }
                 else return '?';
 return c;
}
void prompt( void ){
  printf("READY> ");
}
```

YACC poate fi folosit pentru proiectarea de aplicaţii diverse. Dezvoltarea unui proiect cu ajutorul YACC - ului presupune parcurgerea următoarelor etape:

1. **Identificarea problemei**. Acest lucru presupune o viziune de ansamblu asupra proiectului stabilindu-se arhitectura sistemului ce urmează a fi proiectat, descompunerea sistemului în funcţiile componente, etc;
2. **Definirea limbajului sursă**. Obiectul acestui pas este specificarea limbajului care urmează a fi tradus. Cel mai adecvat mecanism pentru această specificare este gramatica independentă de context pentru că se poate descrie această gramatică direct în limbajul YACC;
3. **Scrierea programului** de descriere a gramaticii în fişierul de intrare pentru YACC;
4. **Scrierea codului auxiliar**. În acest pas se stabilesc şi se scriu acţiunile ataşate regulilor sintactice, funcţiile necesare lansării aplicaţiei (main(), yylex(), yyerror()) precum şi alte instrucţiuni C care urmează a fi încorporate în programul generat de YACC;
5. **Obţinerea funcţiei yyparse().** Odată creat fişierul de intrare, se lansează YACC pentru acesta şi se obţine programul sursă C. Dacă sunt necesare şi alte funcţii C, acestea se pot incorpora în programul obţinut de YACC;
6. **Compilarea textului** obţinut şi obţinerea programului executabil.

5.2 Aplicaţii cu LEX şi YACC

Instrumentele Lex şi Yacc pot fi folosite împreună pentru a realiza analiza lexicală şi sintactică a unui text şi chiar pentru a obţine un interpreter sau un compilator prin mecanismul acţiunilor semantice. Schema de utilizare împreună a celor două generatoare este dată în figura 6.2.1. Dacă fişierul calc.l descrie unităţile lexicale ale limbajului pe care

dorim să-l implementăm iar calc.y descrie sintaxa limbajului, atunci comenzile (Lynux) pentru a obţine fişierul calc.exe din calc.l şi calc.y sunt următoarele:

```
yacc –d calc.y
lex calc.l
cc lex.yy.c y.tab.c –ocalc.exe
```

Comanda yacc produce fişierul y.tab.c care conţine funcţia de parsare yyparse(). Opţiunea –d produce fişierul header y.tab.h care conţine directivele ce atribuie unităţilor lexicale coduri numerice. Acestea vor fi folosite de analizorul lexical produs de lex. Fişierul lang.l va trebui să conţină directiva `#include y.tab.h`.
Comanda lex va produce fişierul lex.yy.c care conţine definiţia funcţiei yylex() care este invocată de yyparse() pentru obţinerea, pe rând, a unităţilor lexicale din textul sursă. Cele două fişiere .c vor fi compilate împreună, şi se obţine în urma editării legăturilor executabilul lang.exe ce poate fi lansat pentru a analiza un text sursă din limbajul dat.

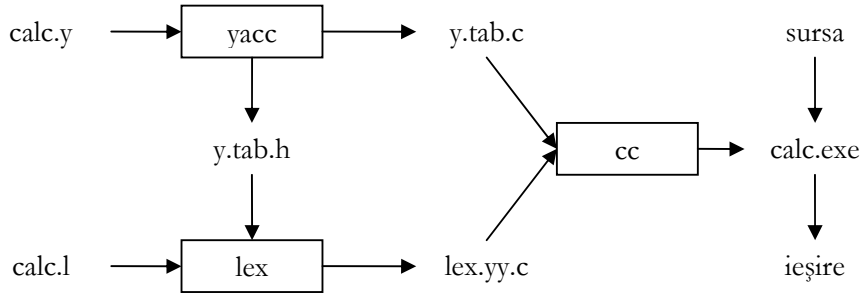


Figura 6.2.1

Exempul 6.2.1 Implementarea unui calculator de buzunar
Vom construi fişierele pentru lex şi yacc care descriu sintaxa expresiilor aritmetice ce se pot forma cu constante întregi (CINT), variabile (VAR), operaţiile aritmetice uzuale şi paranteze. Desigur, orice variabilă ce poate apărea într-o expresii trebuie să fie iniţializată cu valoarea unei expresii . Astfel, un program în accepţiunea de aici este o succesiune de expresii aritmetice şi/sau asignări. Gramatica ce descrie un astfel de limbaj este următoarea:

```
program     → program statement | ε
statement   → expression | VAR '=' expression
expression  → CINT | VAR
              | expression '+' expression
              | expression '-' expression
              | expression '*' expression
              | expression '/' expression
              | '(' expression ')'
```

Fişierul pentru yacc calc.y de mai jos este scris în aşa fel încât acţiunile semantice au ca efect obţinerea valorii numerice a unei expresii atunci când ea este transmisă la intrare programului calc.exe ce se obţine. Tabloul sym[26] este folosit pentru a păstra valorile variabilelor ce apar în programe; o variabilă este aici o literă (va fi descrisă în fişierul pentru lex).

```
// Fisierul calc.y
%{
    #include <stdio.h>
    void yyerror(char *);
    int yylex(void);
    int sym[26];
%}
```

```
%token CINT VAR
%left '+' '-'
%left '*' '/'

%%

program:
        program statement '\n'
        | /* NULL */
        ;

statement:
        expression                      {printf("%d\n", $1);}
        | VAR '=' expression            { sym[$1] = $3; }
        ;

expression:
        CINT
        | VAR                           { $$ = sym[$1]; }
        | expression '+' expression     { $$ = $1 + $3; }
        | expression '-' expression     { $$ = $1 - $3; }
        | expression '*' expression     { $$ = $1 * $3; }
        | expression '/' expression     { $$ = $1 / $3; }
        | '(' expression ')'            { $$ = $2; }
        ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
}
```

Declararea token-urilor `CINT` şi `VAR` prin `%token INT VAR` în fişierul calc.y produce obţinerea fişierului y.tab.h (dacă la invocarea lui yacc se foloseşte parametrul –d) în care sunt definite valorile numerice pentru aceste token-uri. Acest fişier, care este listat mai jos, trebuie inclus în fişierul calc.l.

```
// Fisierul y.tab.h
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define     CINT  258
#define     VAR   259
extern YYSTYPE yylval;
```

Fişierul calc.l conţine descrierea token-urilor `VAR` şi `CINT`. Valorile asociate token-urilor sunt returnate de lex în variabila (standard) yylval. Pentru variabile yylval va însemna indexul în tabloul sym iar pentru constante yylval este numărul întreg corespunzător (scanat de analizorul lexical). Variabila yytext este pointer la unitatea lexicală tocmai depistată, de aceea pentru operatori (token-uri semn) definiţia în fişierul lex este:

```
            [-+()=/*\n]     { return *yytext; }
```

```
// Fisierul calc.l

%{
    #include "y.tab.h"
```

```
    #include <stdlib.h>
    void yyerror(char *);
%}
%%

[a-z]       {
                yylval = *yytext - 'a';
                return VAR;
                }

[0-9]+      {
                yylval = atoi(yytext);
                return CINT;
            }

[-+()=/*\n]     { return *yytext; }

[ \t]   ;       /* ignora spatiile */

.               yyerror("Caracter necunoscut");

%%

int yywrap(void) {
    return 1;
}
```

După obţinerea fişierului calc.exe după schema din figura 6.2.1, o sesiune de lucru cu acest program ar putea fi:

```
utilizator:     x=4
calc:           y=12
utilizator:     (x-y)*(x-y)
calc:           64
utilizator:     a=3
utilizator:     b=7
utilizator:     u=y*y-x*x
utilizator:     v=b*b-a*a
utilizator:     u+v
calc:           168
utilizator:     (u+v)/2+(u-v)/2
calc:           128
utilizator:     u
calc:           128
```

6 Analiza semantică

6.1 Gramatici cu atribute

În 1968 Donald Knuth [Knu68] introduce un formalism pentru specificarea semanticii limbajelor de programare: gramaticii independente de context care exprimă *sintaxa* i se adaugă un sistem de atribute care exprimă *semantica*. Acest formalism s-a dovedit deosebit de util în *analiza semantică* în cadrul compilatoarelor. O *gramatică cu atribute* este o extindere a unei gramatici independente de context: fiecare simbol al acesteia are ataşată o mulţime de *atribute* care pot lua valori într-o mulţime precizată. Valorile atributelor se calculează după nişte reguli ataşate producţiilor gramaticii, numite *reguli semantice*. O *regulă semantică* defineşte modul de calcul al unui atribut al părţii stângi a producţiei – şi atunci atributul se zice că este *sintetizat* – sau al unui atribut al unui simbol din partea dreaptă a producţiei – şi atunci atributul se zice că e *moştenit*. Valoarea unui atribut este definită în funcţie de valorile altor atribute ale simbolurilor din aceeaşi producţie. Se justifică denumirea de *atribut sintetizat*: valorile sale depind de valorile atributelor fiilor într-un arbore de derivare (eventual şi de unele atribute proprii) şi se propagă în arbore de la frunze către rădăcină, precum şi de *atribut moştenit*: valorile sale depind de valorile atributelor părinţilor şi fraţilor, propagându-se de la rădăcină către frunze. Într-un arbore de derivare al gramaticii independente de context un anume atribut poate avea mai multe apariţii, după cum simbolul la care aparţine are mai multe apariţii. Valorile acestora, desigur, pot fi diferite. Este posibil ca simboluri diferite ale gramaticii să aibă atribute cu acelaşi nume; acestea se consideră totuşi atribute diferite. De pildă, dacă simbolurile X şi Y au câte un atribut numite `val`, atunci acestea se exprimă prin X.`val` respectiv Y.`val`. Simbolul de start al gramaticii are un atribut sintetizat care este desemnat a defini *semantica cuvântului* w din limbajul gramaticii. Este natural să considerăm că o gramatică cu atribute care are ca suport (sintactic) gramatica G este bine definită (lucru ce se referă la regulile semantice), dacă pentru orice cuvânt $w \in L(G)$ se poate determina semantica acestuia, respectiv, valoarea atributului desemnat pentru simbolul de start în arborele de derivare al cuvântului w. De asemenea, este natural să spunem că un anume atribut este *util* dacă el contribuie la calculul semanticii lui w, adică semantica lui w depinde, direct sau indirect, de valoarea acestui atribut. Astfel, o gramatică cu atribute se zice că este *bine definită* (*necirculară*) dacă, pentru orice arbore de derivare construit în gramatica suport, se pot calcula valorile tuturor apariţiilor atributelor utile în acest arbore (se mai spune că arborele poate fi decorat). Aşadar, există două probleme importante legate de gramaticile cu atribute:

- **Problema circularităţii**: este gramatica cu atribute bine definită?
- **Problema evaluării**: care este cea mai bună strategie de evaluare a atributelor în gramatica dată (presupusă necirculară).

Abordarea acestor probleme conduce la identificarea unor clase de gramatici cu atribute pe care le vom trece în revistă în acest capitol. Înainte de a defini riguros noţiunea de gramatică cu atribute şi a studia clasele importante, să prezentăm nişte exemple.

6.1.1 Numere raționale în baza 2 (Knuth)

Se consideră gramatica independentă de context cu următoarele reguli:

$N \to L$ $\quad L \to LB$ $\quad B \to 0$

$N \to L.L$ $\quad L \to B$ $\quad B \to 1$

Se constată uşor că limbajul generat de această gramatică este:

$L(G) = \{b_1b_2\ldots b_n \mid b_i \in \{0, 1\}, 1 \le i \le n\} \cup$

$\cup \{ b_1b_2\ldots b_n . b_{n+1}b_{n+2}\ldots b_m \mid b_i \in \{0, 1\}, 1 \le i \le m, m > n \ge 1\}$

Dorim să înzestrăm această gramatică cu atribute şi reguli semantice astfel ca pentru fiecare cuvânt w generat de aceasta să putem determina numărul raţional (în baza 10) a cărui reprezentare în baza 2 este w. Pentru aceasta considerăm atributele:

- Pentru simbolul B (bit):
  - Un atribut sintetizat, numit **val,** care are ca domeniu al valorilor mulţimea numerelor raţionale şi care trebuie interpretat astfel: dacă B generează 0, atunci **B.val** are valoarea 0, iar dacă B generează 1 atunci **B.val** are valoarea 2^s unde s este locul lui 1 în reprezentarea binară în cauză (s se numeşte scală);
  - Un atribut moştenit numit **scala** care va defini exponentul lui 2 pentru calculul valorii lui B. Domeniul valorilor acestui atribut este mulţimea numerelor întregi;
- Pentru variabila L (listă de biţi):
  - Două atribute sintetizate: **val** şi **lung** care au ca domenii de valori mulţimea numerelor raţionale, respectiv naturale iar ca interpretare, valoarea numerică zecimală, respectiv lungimea lui w dacă $L \overset{+}{\Rightarrow} w$;
  - Un atribut moştenit **scala** care are ca domeniu de valori mulţimea numerelor întregi şi care va fi folosit pentru calculul scalei lui B.
- Pentru variabila N (număr):
  - Un atribut sintetizat **val** care reprezintă semantica cuvintelor generate în gramatică. Domeniul său de valori este mulţimea numerelor raţionale.

Regulile semantice sunt ataşate regulilor sintactice (producţiilor gramaticii) şi sunt date în tabela de mai jos. Facem precizarea că, în producţiile în care L are apariţii multiple ($L \to LB$, $N \to L.L$), aceste apariţii au fost indexate pentru a evita confuzia.

Reguli sintactice	Reguli semantice
$B \to 0$	B.**val** = 0
$B \to 1$	B.**val** = $2^{B.\mathbf{scala}}$
$L \to B$	L.**val** = B.**val**; L.**lung** = 1; B.**scala** = L.**scala**
$L_1 \to L_2B$	L_1.**val** = L_2.**val** + B.**val**; L_1.**lung** = L_2.**lung** + 1 L_2.**scala** = L_1.**scala** + 1; B.**scala** = L_1.**scala**
$N \to L$	N.**val** = L.**val**; L.**scala** = 0
$N \to L_1.L_2$	N.**val** = L_1.**val** + L_2.**val**; L_1.**scala** = 0; L_2.**scala** = - L_2.**lung**

Să determinăm, folosind regulile semantice date mai sus, semantica cuvântului w = 101.11. Arborele de derivare pentru acest cuvânt în gramatica suport este dat în figura 7.1.1

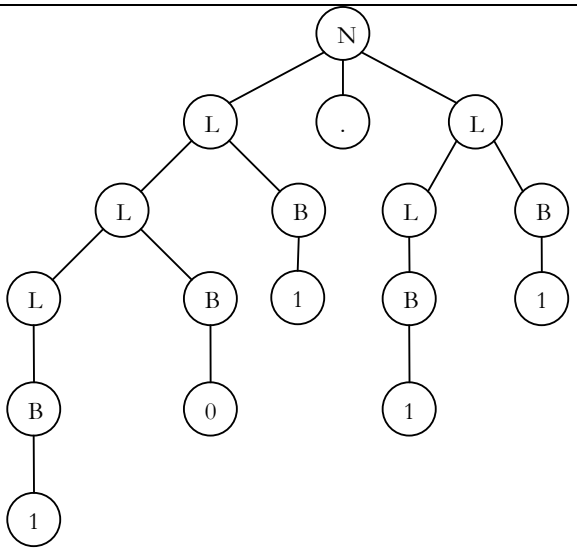


Figura 7.1.1

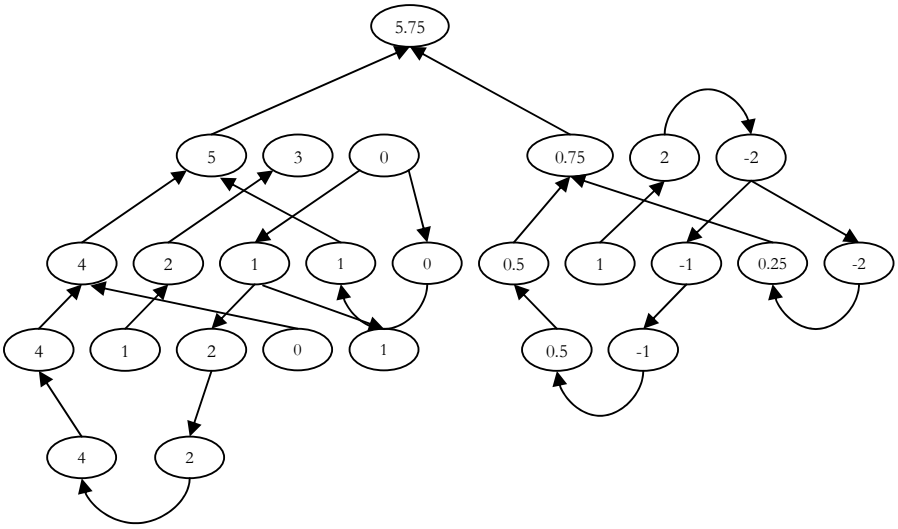


Figura 7.1.2

Arborele de derivare decorat este reprezentat în figura 7.1.2. Să precizăm faptul că pentru nodurile etichetate cu L s-au reprezentat valorile pentru atributele **val**, **lung** respectiv **scala**, în această ordine iar pentru cele etichetate cu B sunt reprezentate valorile atributelor **val** şi **scala**. Arcele definesc un graf de dependenţă care dă ordinea de determinare a valorilor atributelor.

6.1.2 Declararea variabilelor într-un program

Să considerăm gramatica următoare:

$Z \rightarrow$ `begin` D `;` I `end`

$D \rightarrow D$ `type` V | `type` V

$I \rightarrow I$ `;` `use` V | `use` V

$V \rightarrow V$, `v` | `v` unde `v` $\in$ Var = {`a, b, c, …`}.

Această gramatică generează programe care încep cu `begin`, se sfârşesc cu `end` iar între aceşti doi delimitatori pot fi liste de declaraţii urmate de liste de instrucţiuni. O declaraţie

este de forma : `type` v_1, v_2, …v_n iar o instrucţiune are forma `use` v_1, v_2, …v_m. Iată exemple de programe corecte în această descriere sintactică:

```
1) begin type a, b; type c; use a, c; use b, c end
2) begin type a, b; type a, c; use a, c; use c end
3) begin type x, a; use x ; use a, c end
4) begin type x, c; type x, z; use x; use z, x end
```

În descrierea (semantică) a limbajelor de programare există în general regulile:

- Orice variabilă din program trebuie declarată.
- Declararea multiplă a unei variabile nu este permisă.

După aceste reguli, programele 2, 3, 4 de mai sus nu sunt corecte. Ne propunem să descriem o gramatică cu atribute care să descrie prin regulile semantice cele precizate mai sus.

Variabilelor gramaticii de mai sus le ataşăm atributele:

Variabila Z:

- atributul sintetizat **Răspuns** cu valori în mulţimea {true, false} care dă răspuns la întrebarea: este programul corect?

Variabila D:

- Atributul sintetizat **Dec**, care are ca valori submulţimi de variabile şi eventual constanta literală *eroare*. D.**Dec** conţine toate variabilele care sunt declarate în subarborele lui D, şi numai pe acelea, iar dacă o variabilă are declaraţii multiple, D.**Dec** va conţine şi *eroare*.

Variabila I:

- Atributul sintetizat **Răspuns** cu valori în {true, false}; I.**Răspuns** este true dacă şi numai dacă variabilele folosite în instrucţiunile generate de I au fost declarate în acelaşi program.
- Atributul moştenit **Tabel** cu valori în 2^{Var}. I.**Tabel** conţine toate variabilele declarate în programul curent.

Variabila V:

- Atributul sintetizat **Lista** cu valori în $2^{Var \cup \{eroare\}}$. V.**Lista** reprezintă lista variabilelor generate de V şi eventual constanta *eroare*.
- Atributul moştenit **Loc** cu valori în mulţimea {*dec*, *inst*} .V.**Loc** este *dec* dacă V face parte din subarborele cu rădăcina D şi este *inst* dacă V face parte din subarborele cu rădăcina I.

Regulile semantice pentru calculul valorilor atributelor sunt date în tabelul de mai jos.

SINTAXA	SEMANTICA
Z → `begin` D `;` I `end`	Z.**Răspuns** = `if (eroare ∈` D.**Dec**`)` `then false else` I.**Răspuns** I.**Tabel** = D.**Dec** − {eroare}
$D_1 \rightarrow D_2$ `;` `type` V	D_1.**Dec** = D_2.**Dec** ∪ V.**Lista** ∪ ∪ `if(`V.**Lista** ∩D_2.**Dec**`)` `then {eroare} else` Φ V.**Loc** = *dec*
D → `type` V	D.**Dec** = V.**Lista**; V.**Loc** = dec.
$I_1 \rightarrow I_2$ `;` `use` V	I_1.**Răspuns** = `if(`V.**Lista** ⊄ I_1.**Tabel**`)` `then false else` I_2.**Răspuns** I_2.**Tabel** = I_1.**Tabel**; V.**Loc** = *inst*.
I → `use` V	I.**Răspuns** = `if(`V.**Lista** ⊆ I.**Tabel**`)` `then true else false` V.**Loc** = *inst*
$V_1 \rightarrow V_2$, v	V_1.**Lista** = V_2.**Lista** ∪{v} ∪ ∪ `if(`V_1.**Loc** = dec and v ∈ V_2.**Lista**`)` `then eroare else` Φ;

	V_2.**Loc** = V_1.**Loc**
V → v	V.**Lista** = {v}

Fie acum programul:

```
begin type x, a; use x ; use a, c end
```

Arborele de derivare pentru acesta este dat în figura 7.1.3. Din motive lesne de înţeles, etichetele nodurilor corespunzătoare terminalilor begin, end, type şi use sunt reprezentate în arbore respectiv prin b, e, t şi u.

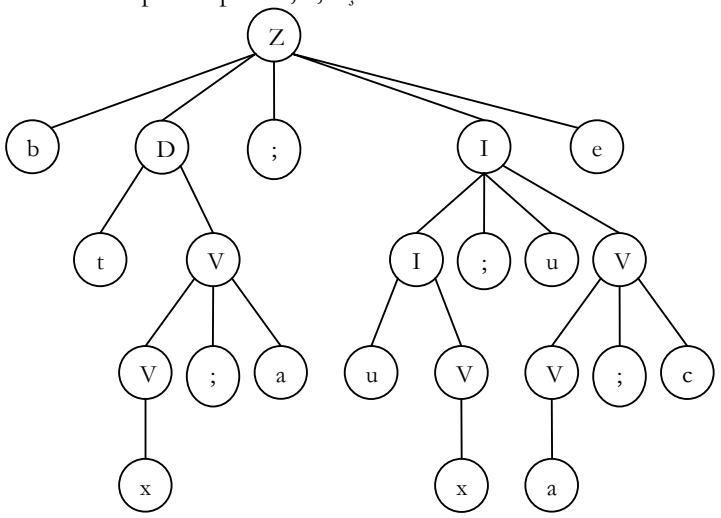


Figura 7.1.3

Dacă pentru I reprezentăm atributele în ordinea I.**Tabel**, I.**Răspuns** iar pentru V în ordinea V.**Lista**, V.**Loc,** atunci valorile calculate ale atributelor pentru acest arbore sunt date în graful din figura 7.1.4. Nodurile acestui graf corespund aparițiilor atributelor neterminalilor Z, D, I şi V din arborele de mai sus, iar arcele reprezintă dependenţele între valorile acestor atribute prin regulile semantice.

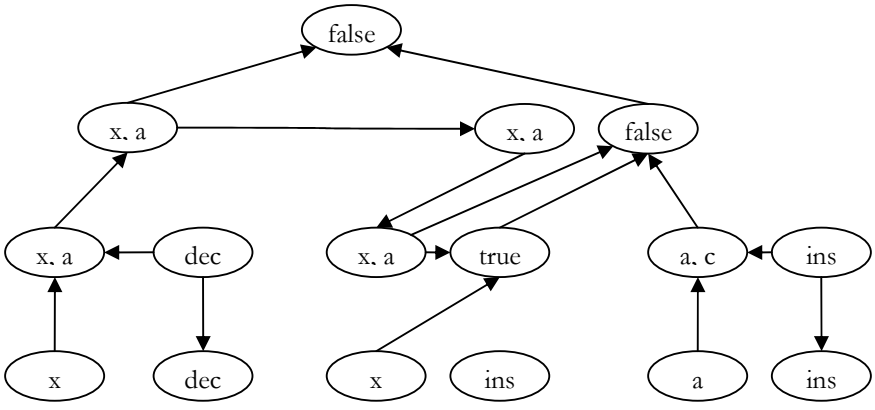


Figura 7.1.4

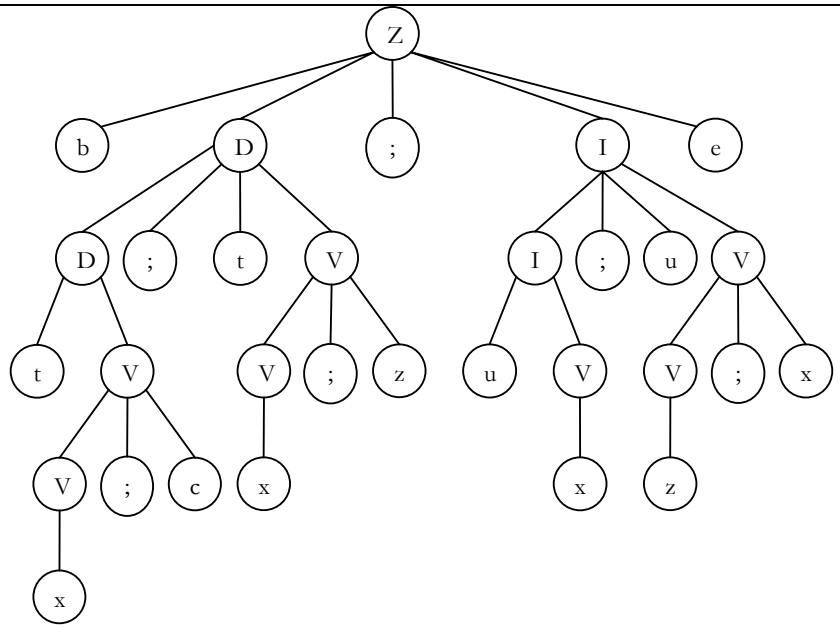


Figura 7.1.5

Pentru programul:

```
begin type x, c; type x, z; use x; use z, x end
```

arborele de derivare este dat în figura 7.1.5 iar valorile atributelor în figura 7.1.6.

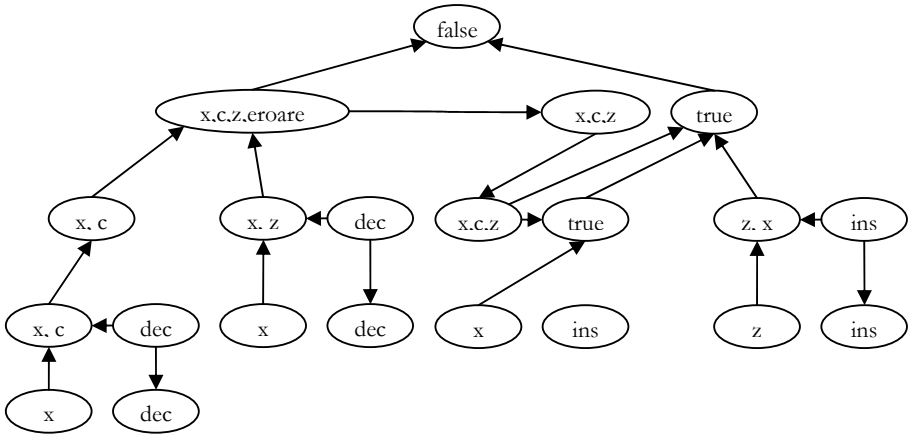


Figura 7.1.6

6.1.3 Definiția gramaticilor cu atribute

Definiția 7.1.1 O gramatică cu atribute este un 4-uplu $\mathcal{GA} = (G, \mathcal{D}, \mathcal{A}, \mathcal{R})$ unde:

1. G = (V, T, Z, P) este o gramatică independentă de context redusă, numită *gramatica suport* pentru $\mathcal{GA}$. Notăm prin $\Sigma = V \cup T$ vocabularul gramaticii. Aici Z reprezintă simbolul de start al gramaticii.
2. $\mathcal{D} = (\mathcal{M}, \mathcal{F})$ este *domeniul semantic* format din:
   2.1. $\mathcal{M}$ *– o familie finită de mulțimi*;
   2.2. $\mathcal{F}$ *– o familie finită de funcții total definite și calculabile* de forma
   $$f : M_1 \times M_2 \times \ldots \times M_n \rightarrow M_0 \text{ , unde } M_i \in \mathcal{M},\ 0 \leq i \leq n.$$
3. $\mathcal{A} = (I, S, \text{type})$ este o *descriere a atributelor gramaticii*, unde;

3.1. $\mathcal{I} = \cup_{x\in\Sigma} I(X)$ este *mulţimea atributelor moştenite*;

3.2. $\mathcal{S} = \cup_{x\in\Sigma} I(X)$ este *mulţimea atributelor sintetizate*;

3.3. Pentru orice atribut $\alpha \in \mathcal{I} \cup \mathcal{S}$, type($\alpha$) $\in \mathcal{M}$ este mulţimea din domeniul semantic în care α poate lua valori;

3.4. Atributele *moştenite* $I(X)$ şi cele *sintetizate* $\mathcal{S}(X)$ pentru simbolul X se consideră disjuncte, oricare ar fi $X \in \Sigma$. De asemenea, notând $\mathcal{A}(X) = I(X) \cup \mathcal{S}(X)$, vom considera $\mathcal{A}(X) \cap \mathcal{A}(Y) = \Phi$, $\forall X, Y \in \Sigma$, $X \neq Y$, chiar dacă, uneori, atribute ale unor simboluri diferite au aceleaşi nume (ca în exemplele prezentate). Referirea la atributul α al simbolului X se face prin $X.\alpha$.

3.5. Simbolul de start Z al gramaticii are un atribut sintetizat desemnat a reprezenta *semantica* unei construcţii $w \in L(G)$.

4. $\mathcal{R} = \cup_{p\in P}\mathcal{R}(P)$ este mulţimea (finită) a *regulilor semantice* (reguli de evaluare a atributelor). Dacă producţia p este de forma: $X_{p0} \rightarrow X_{p1}X_{p2}\ldots X_{pnp}$ atunci o regulă semantică este de forma $X_{pk}.a = f(X_{pk1}.a_1, X_{pk2}.a_2,\ldots, X_{pkm}.a_m,)$ unde :

4.1. $f \in \mathcal{F}$, $f : type(a_1) \times type(a_2) \times \ldots \times type(a_m) \rightarrow type(a)$;

4.2. Pentru $k = 0$ atunci $a \in \mathcal{S}(X_{p0})$ cu precizarea că $\mathcal{R}(P)$ conţine câte o regulă şi numai una pentru fiecare atribut din $\mathcal{S}(X_{p0})$.

4.3. Pentru $k > 0$ atunci $a \in I(X_{pk})$ iar $\mathcal{R}(p)$ conţine câte o regulă şi numai una pentru fiecare atribut din $\cup_{1\le k\le np} I(X_{pk})$.

4.4. $a_i \in I(X_{p0}) \cup (\cup_{1\le k\le np}\mathcal{A}(X_{pk}))$, $1\le k\le np$.

□

Exemplul 7.1.1 Să punem în evidenţă elementele din definiţia gramaticii cu atribute pentru gramatica prezentată în paragraful 7.1.1.

1. $G = (\{N, L, B\}, \{0, 1, .\}, N, P)$ unde
$$P = \{N \rightarrow L,\ N \rightarrow L.L,\ L \rightarrow LB,\ L \rightarrow B,\ B \rightarrow 0,\ B \rightarrow 1\};$$
2. $\mathcal{M} = \{\mathbf{N}, \mathbf{Z}, \mathbf{Q}\}$, $\mathcal{F} = \{c_0, c_1, id, suc, minus, suma, exp\}$, unde c_0 şi c_1 sunt funcţiile constante 0 respectiv 1, *suc* este funcţia succesor, *minus*(x) = -x, *suma*(x, y) = x + y, *exp*(x) = 2^x. Domeniile acestor funcţii sunt adecvate regulilor semantice descrise.
3. $\mathcal{A}(N) = \mathcal{S}(N) = \{\mathbf{val}\}$, type(**val**) = **Q**
$\mathcal{S}(L) = \{\mathbf{val}, \mathbf{lung}\}$, type(**val**) = **Q**, type(**lung**) = **N**
$I(L) = \{\mathbf{scala}\}$, type(**scala**) = **Z**
$\mathcal{S}(B) = \{\mathbf{val}\}$, type(**val**) = **Q**
$I(B) = \{\mathbf{scala}\}$, type(**scala**) = **Z**

Observaţie. Aparent $I(L) = I(B) = \{\mathbf{scala}\}$. Trebue precizat însă că cele două atribute cu acelaşi nume sunt diferite: L.**scala** ≠ B.**scala** şi acest lucru se vede din definirea acestora prin regulile semantice. Acelaşi lucru este valabil pentru atributul **val** care apare la S, L şi B.

4. Regulile semantice sunt cele din tabela de la paragraful 7.1.1. Este uşor de verificat că au loc 6.1 – 6.4.

Exemplul 7.1.2 Să punem în evidenţă elementele din definiţia gramaticii cu atribute pentru gramatica prezentată în 7.1.2.

1. $G = (\{Z, D, I, V\}, \{begin, end, tip, use, ;, a, b, \ldots\}, S, P)$ unde
$P = \{Z \rightarrow$ begin D;Iend, D $\rightarrow$ D;type V, D $\rightarrow$ type V, I $\rightarrow$ I; use V,
I $\rightarrow$ use V, V $\rightarrow$v, V $\rightarrow$ V, v$\}$.
2. $\mathcal{M} = \{Bool, 2^{Var}, 2^{Var \cup \{eroare\}}, \{dec, inst\}\}$ unde Bool = {true, false}. $\mathcal{F}$ conţine toate funcţiile folosite în tabelul ce defineşte regulile semantice.

3. $\mathcal{A}(Z) = \mathcal{S}(Z) = \{$**Răspuns**$\}$, type(**Răspuns**) = Bool
   $\mathcal{A}(D) = \mathcal{S}(D) = \{$**Dec**$\}$, type (**Dec**) = $2^{Var \cup \{eroare\}}$
   $\mathcal{S}(I) = \{$**Răspuns**$\}$, type(**Răspuns**) = Bool
   $\mathcal{I}(I) = \{$**Tabel**$\}$, type(**Tabel**) = 2^{Var}
   $\mathcal{S}(V) = \{$**Lista**$\}$, type(**Lista**) = $2^{Var \cup \{eroare\}}$)
   $\mathcal{I}(V) = \{$**Loc**$\}$, type(**Loc**) = $\{dec, inst\}$
4. Regulile semantice sunt cele din tabela de la 7.1.2.

6.2 Grafuri ataşate unei gramatici cu atribute

În figurile 7.1.2, 7.1.4 şi 7.1.6 sunt puse în evidenţă nişte grafuri ataşate arborilor de derivare pentru anumite cuvinte generate de gramatica suport a gramaticii cu atribute. Aceste grafuri (şi altele pe care le vom defini în continuare) joacă un rol important în verificarea corectitudinii definirii regulilor semantice. În esenţă o regulă semantică defineşte o relaţie de dependenţă între atributele din gramatica respectivă, iar această relaţie defineşte un graf.

6.2.1 Graful DG_p

Definiţia 7.2.1 Fie $\mathcal{GA} = (G, \mathcal{D}, \mathcal{A}, \mathcal{R})$ o gramatică cu atribute şi producţia p dată de $X_0 \rightarrow X_1X_2 \ldots X_{np}$. Graful $\mathcal{DG}_p = (\mathcal{X}, \mathcal{E})$ asociat producţiei p, numit *graful de dependenţă* al producţiei p, este graful în care:

- nodurile sunt toate aparițiile atributelor simbolurilor din producţia p, adică $\mathcal{X} = \cup_{1 \le i \le np} A(X_i)$;
- arcele sunt de forma $(X_i.\alpha, X_j.\beta)$ dacă $X_j.\beta$ depinde printr-o regulă semantică de $X_i.\alpha$ (existentă în $\mathcal{R}(p)$), adică:

$$\mathcal{E} = \{(X_i.\alpha, X_j.\beta) \mid X_j.\beta = f(..., X_i.\alpha, ...) \in \mathcal{R}(p)\}$$

Exemplul 7.2.1 Fie gramatica din exemplul 7.1.1. Atunci, grafurile ataşate producţiilor gramaticii sunt prezentate în figurile 7.2.1 – 7.2.6

$\mathcal{DG}_1$:

Figura 7.2.1

$\mathcal{DG}_2$:

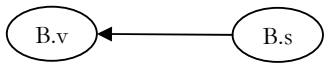


Figura 7.2.2

$\mathcal{DG}_3$:

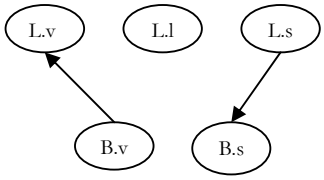


Figura 7.2.3

$\mathcal{DG}_4$:

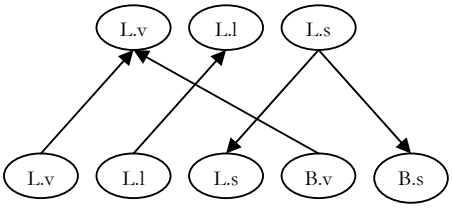


Figura 7.2.4

$\mathcal{DG}_5$:

$\mathcal{DG}_6$:

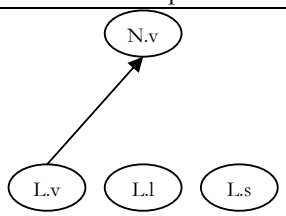


Figura 7.2.5

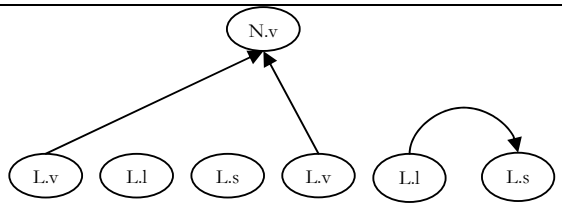


Figura 7.2.6

Cititorul este invitat să construiască grafurile $\mathcal{DG}_p$ pentru gramatica din exemplul 7.1.2.

6.2.2 Graful BG_p

Definiţia 7.2.2 Fie $\mathcal{GA}$ = (G, $\mathcal{D}$, $\mathcal{A}$, $\mathcal{R}$) o gramatică cu atribute şi producţia p dată de $X_0 \rightarrow X_1X_2 \ldots X_{np}$. Graful $\mathcal{BG}_p$ asociat producţiei p este graful în care:

- nodurile sunt X_1, X_2, ..., X_{np}, privite ca elemente distincte(dacă un simbol se repetă în p, atunci apariţiile lui se numerotează);
- arcele sunt toate perechile (X_i, X_j) şi numai acelea, pentru care există un atribut moştenit al lui X_j care depinde (printr-o regulă semantică) de un atribut sintetizat al lui X_i , $1 \le i, j \le n_p$.

Exemplul 7.2.2 Să considerăm gramatica din exemplul 7.1.2. Atunci graful $\mathcal{BG}_1$, corespunzător producţiei `Z → begin D;I end`, este dat în figura 7.2.7. Pentru gramatica din exemplul 7.1.1 graful $\mathcal{BG}_6$ este dat în figura 7.2.8.

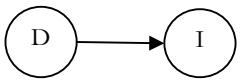


Figura 7.2.7

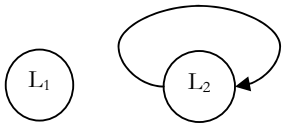


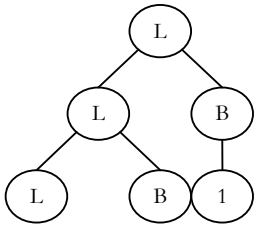
Figura 7.2.8

6.2.3 Graful DG_t

Definiţia 7.2.3 Fie $\mathcal{GA}$ = (G, $\mathcal{D}$, $\mathcal{A}$, $\mathcal{R}$) o gramatică cu atribute şi t un arbore de derivare în gramatica suport G. Graful $\mathcal{DG}_t$ ataşat arborelui t se defineşte recursiv astfel:

- Dacă t = p ∈ P atunci $\mathcal{DG}_t = \mathcal{DG}_p$;
- Dacă t este un arbore de adâncime mai mare ca 1, cu rădăcina etichetată X_0 iar producţia ce se aplică în rădăcină este $p = X_0 \rightarrow X_1X_2...X_{np}$, atunci $\mathcal{DG}_t$ se obţine din $\mathcal{DG}_p$ şi $\mathcal{DG}_{ti}$ $1 \le i \le n_p$, prin alipire: nodurile din $\mathcal{DG}_p$ corespunzătoare atributelor lui X_i se identifică respectiv cu nodurile din $\mathcal{DG}_{ti}$ corespunzătoare aceloraşi atribute ale lui X_i. În cazul în care un anume X_i este terminal, graful $\mathcal{DG}_{ti}$ corespunzător are ca noduri atributele lui X_i şi nu are nici un arc.

Exemplul 7.2.3 Pentru gramatica din exemplul 7.1.1 graful $\mathcal{DG}_t$ pentru arborele din figura 7.2.9 este dat în figura 7.2.10. El se obţine din grafurile $\mathcal{DG}_4$ (corespunzător producţiei din rădăcină) la care se alipesc grafurile $\mathcal{DG}_4$ şi $\mathcal{DG}_2$ ca în figura 7.2.11.



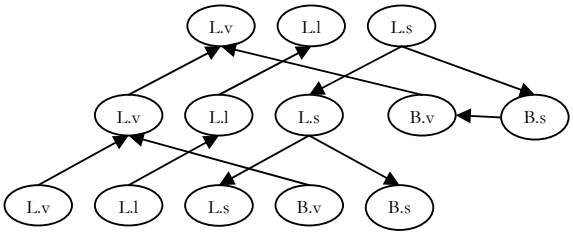


Figura 7.2.9 **Figura 7.2.10**

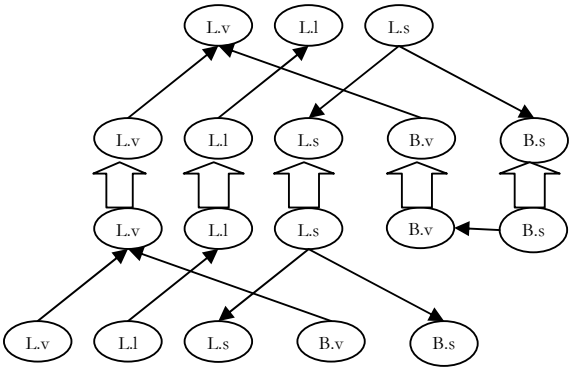


Figura 7.2.11

Pentru arborele din figura 7.1.1 graful $\mathcal{DG}_t$ este obţinut în acelaşi mod şi este dat în figura 7.2.12.

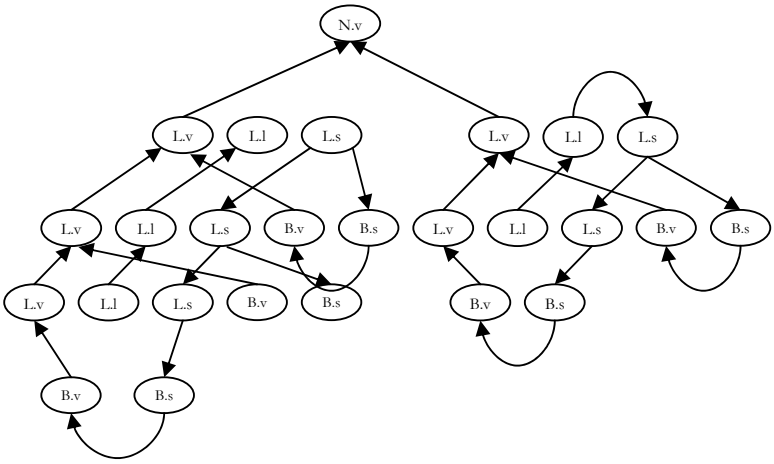


Figura 7.2.12

Graful $\mathcal{DG}_t$ pentru un arbore de derivare t complet (în care rădăcina este etichetată cu simbolul de start Z iar frunzele sunt etichetate cu terminali) se obţine, se înţelege, în acelaşi mod. Dacă acest graf nu are circuite, atunci el defineşte o ordine parţială în mulţimea aparţiilor atributelor în arborele t: dacă (X.α, Y.β) este un arc în $\mathcal{DG}_t$, atunci, pentru a putea calcula valoarea lui Y.β avem nevoie de valoarea lui X.α.

Definiţia 7.2.4 O gramatică cu atribute $\mathcal{GA}$ este *circulară* (sau nu este bine definită) dacă există măcar un arbore de derivare t pentru care graful $\mathcal{DG}_t$ are circuite. O gramatică cu atribute $\mathcal{GA}$ este *necirculară* (este *bine definită*) dacă pentru orice arbore de derivare t graful $\mathcal{DG}_t$ nu are circuite.

Pentru o gramatică cu atribute necirculară, dat un arbore de derivare t, deci un cuvânt w $\in$ L(G), se pot calcula valorile tuturor aparițiilor (instanţelor) atributelor din arborele t, deci se poate determina semantica lui w. Aşadar, două din problemele importante ce se pun referitor la o gramatică cu atribute sunt:

- *problema circularităţii*: dată $\mathcal{GA}$, este ea necirculară?
- *problema evaluării atributelor*: dată $\mathcal{GA}$ necirculară, să se construiască o procedură care, atunci când este dat un arbore t, să determine valorile tuturor aparițiilor atributelor etichetelor arborelui.

Înainte de a aborda problema circularităţii, să mai definim nişte grafuri ce vor fi utile rezolvării acesteia.

6.3 Metode de evaluare a atributelor

Un evaluator pentru o gramatică cu atribute este un program care, pentru un arbore de derivare t al gramaticii suport, determină valorile tuturor aparițiilor atributelor din arborele t. Un generator de evaluator este un program care, pentru o gramatică cu atribute ca intrare, o respinge (dacă ea este circulară) sau o acceptă şi construieşte pentru ea un evaluator.

În funcţie de modul cum sunt evaluate atributele, un evaluator poate fi:

- recursiv sau iterativ;
- aplicativ sau imperativ;
- tree-walking sau tree-free;
- pass-oriented sau visit-oriented;

Un tip de evaluator corespunde unei clase de gramatici cu atribute pentru care se poate construi acel tip de evaluator.Vom trece în revistă câteva metode de evaluare împreună cu clasele de gramatici pentru care se aplică aceste metode.

6.3.1 Evaluatorul rezultat din definiția gramaticii

Pentru a putea calcula valorile tuturor aparițiilor atributelor într-un arbore de derivare t, graful $\mathcal{DG}_t$ trebuie să fie liber de circuite, aşa cum a fost arătat în 7.2.3. Să considerăm cazul general al relațiilor pentru a defini o ordine de evaluare.

Dacă V este o mulţime finită iar $R \subseteq V \times V$ este o relație peste V, atunci (V, R) este un graf orientat. Să notăm prin R^+ închiderea tranzitivă a relației R. Atunci R este *necirculară* (sau *aciclică*, sau *ireflexivă*) dacă $\forall v \in V$, $(v, v) \notin R^+$. Relația $T \subseteq V \times V$ este o *ordine totală* pe V dacă :

- $T^+ = T$ (T este tranzitivă);
- T este necirculară;
- T este totală: $\forall x \in V, \forall y \in V$, $x = y$ sau $(x, y) \in T$ sau $(y, x) \in T$.

Pentru orice ordine totală T pe mulţimea V, există o enumerare a elementelor lui V, să zicem $v_1, v_2, \ldots, v_n$, astfel ca $(v_i, v_j) \in T$ dacă şi numai dacă $i < j$. Această enumerare se numeşte *secvență de evaluare*.

Spunem că o ordine totală T este o *ordine de evaluare* a relației R pe V, dacă $R \subseteq T$. Dacă $v_1, v_2, \ldots, v_n$ este secvența de evaluare pentru T, atunci, $(v_i, v_j) \in R$ implică $i < j$. Un element $v \in V$ se numeşte *element minimal* pentru R dacă nici un element din V nu este în relație cu v ($\forall w \in V$, $(w, v) \notin R$). Analog, $v \in V$ este *maximal* pentru R dacă el nu este în relație cu nici un element ($\forall w \in V$, $(v,w) \notin R$).

Teorema 7.4.1 (de evaluare) Fie R o relație pe V. R este necirculară dacă şi numai dacă R admite o ordine de evaluare.

Demonstrație. Dacă R admite o ordine de evaluare T, aceasta este necirculară (din definiția ordinii totale). Cum $R \subseteq T$ rezultă că şi R este necirculară. Invers, să presupunem că R este o relație necirculară pe V. Următorul algoritm, numit de *sortare topologică*, determină o secvență de evaluare pe R, ceea ce este suficient pentru o ordine de evaluare pe R:

Algoritmul 7.4.1

Intrare: Mulţimea V şi relația $R \subseteq V \times V$ necirculară;

Ieşire: O secvență de evaluare eval = $\{v_1, v_2, \ldots, v_n\}$, unde $|V| = n$;

Metoda:

```
1. eval = Φ; k = 0;
```

```
2. while (eval ≠ V) {
3.     Se alege v un element minimal din V – eval;
4.     k = k + 1;
5.     v_k = v;
6.     eval = eval∪{v_k};
   }
```

Să observăm că în V – eval există întotdeauna un element minimal pentru că R este necirculară pe V, deci şi restricţia sa la V – eval este necirculară. Algoritmul este finit iar din modul cum s-au adăugat elementele în eval rezultă că aceasta reprezintă o secvenţă de evaluare pe R.

□

Algoritmul 7.4.1 poate fi folosit pentru proiectarea unui evaluator pentru o gramatică cu atribute care este necirculară. Considerând un arbore de derivare t, aplicăm algoritmul precedent pentru relaţia ce defineşte graful $\mathcal{DG}_t$ pentru a afla ordinea de evaluare a atributelor în t.

Evaluatorul P1

Intrare: Gramatica cu atribute $\mathcal{GA} = (G, \mathcal{D}, \mathcal{A}, \mathcal{R})$ necirculară;
Graful $\mathcal{DG}_t = (\mathcal{N}, \mathcal{E})$ pentru un arbore t ($\mathcal{N}$ este mulţimea tuturor aparţiilor atributelor din t);

Ieşire: Valorile tuturor aparţiilor atributelor în t;

Metoda:

```
1. eval = Φ; k = 0;
2. while(eval ≠ N) {
3.     Se alege v un element minimal din N – eval;
4.     evaluare(v);//se aplica o regulă semantica
5.     k = k + 1;v_k = v; eval = eval ∪ {v_k};
   }
```

În linia 4 din algoritm se apelează o procedură care evaluează, conform cu regulile semantice, valorile atributului v. Acest lucru este posibil pentru că valorile atributului v depind de valorile unor atribute ce au fost calculate înainte, conform secvenţei de evaluare.

Evaluatorul tocmai prezentat este important pentru faptul că reprezintă o caracterizae a gramaticilor cu atribute necirculare: o gramatică cu atribute este necirculară dacă şi numai dacă P1 este un evaluator pentru aceasta. P1 este cunoscut în literatură ca şi "*defining evaluator*" – el exprimă faptul că gramatica respectivă este bine definită.

6.4 Traducere bazată pe sintaxă

Una din cele mai frecvente tehnici folosite în analiza semantică este cea a evaluării atributelor în timpul analizei sintactice ascendente (LR). Am văzut în paragraful precedent cum pot fi evaluate atributele în cadrul unui parser SLR(1). Dacă toate atributele gramaticii sunt sintetizate atunci lucrurile se simplifică: la fiecare reducere cu o producţie de forma $A \rightarrow \beta$ se aplică regulile semantice pentru evaluarea atributelor (sintetizate) ale lui A. Pentru analiza semantică a limbajelor de programare în acest mod, precum şi pentru obţinerea codului (intermediar) echivalent cu codul sursă, se folosesc *acţiunile semantice* ataşate regulilor sintactice, care sunt aplicate atunci când analizorul sintactic realizează o reducere. Se obţine astfel o structură care nu este altceva decât o gramatică cu atribute în care există un atribut care are ca valori secvenţe de cod intermediar. Aşa cum o să vedem în continuare, determinarea valorilor atributelor se poate face prin aplicarea uor proceduri specifice. Regulile semantice nu mai sunt chiar de forma pe care am specificat-o în definiţia gramaticii cu atribute ci poate apărea, pe lângă definiţii ale atributelor cu ajutorul unor funcţii calculabile, şi apelul unor proceduri

care generează cod intermediar. De aceea numim acţiuni semantice aceste proceduri ataşate regulilor sintactice. O astfel de gramatică cu atribute este cunoscută în literatură sub numele de *schemă de traducere bazată (orientată) pe sintaxă*. Schemele de traducere bazate pe sintaxă sunt deosebit de utile celor ce implementează limbaje de programare; într-o astfel de schemă poate fi implementată generarea de cod intermediar pornind de la structura sintactică a limbajului sursă. Odată obţinut codul intermediar, acesta urmează a fi transformat în cod obiect pentru o maşină anume. În acest paragraf vom prezenta mai întâi câteva tipuri de cod intermediar pentru ca apoi să vedem cum se abordează traducerea structurilor sintactice importante din limbajele de programare.

6.4.1 Reprezentări intermediare

Analiza sintactică oferă la ieşire reprezentarea programului ca arbore de derivare (parsare) în gramatica ce descrie sintaxa limbajului. Pentru a crea codul obiect echivalent cu cel sursă, în general se foloseşte o reprezentare intermediară a acestuia, un cod intermediar. Tipurile de cod intermediar utilizate în procesul de compilare sunt:

Notaţia postfix (Notaţia poloneză inversă). Această reprezentare se generează uşor de la arborele abstract printr-o traversare postordine, sau cu un parser bottom-up. Avantajul acestei reprezentări este acela că evaluarea expresiilor se face uşor şi eficient folosind o stivă. Instrucţiunile pot fi privite ca şi operatori (de o anume aritate), încât reprezentarea unui program se face uniform. De pildă, dacă notăm prin ? operatorul ternar if_then_else, atunci expresia

```
if a + b > c then x = a – c else x = a – b
```

are reprezentarea în notaţia postfix astfel:

```
a b + c > x a c - = x b c - = ?
```

Forma postfix a unui program se poate obţine folosind Yacc (vezi exemplul 6.1.3). Există şi un dezavantaj major în această abordare: dacă ne referim la reprezentarea operatorului if_then_else, atunci evaluarea expresiei presupune evaluarea tuturor subexpresiilor, chiar dacă acest lucru nu este necesar. Pentru optimizare se introduc nişte operatori de salt condiţionat în cadrul reprezentării.

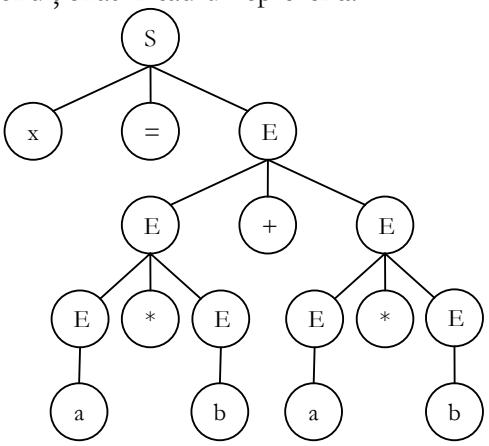


Figura 7.5.1

Arborele sintactic (abstract). Acesta se obţine din arborele (concret) de parsare în care neterminalii sunt înlocuiţi cu operatorii corespunzători (dacă ne referim la expresii). Să analizăm, spre exemplu, arborele de parsare (figura 7.5.1) pentru atribuirea `x = a * b + a * b` ce se obţine în gramatica:

$$S \rightarrow id = E \qquad E \rightarrow E + E \mid E * E \mid (E) \mid id$$

Arborele sintactic corespunzător este dat în figura 7.5.2.

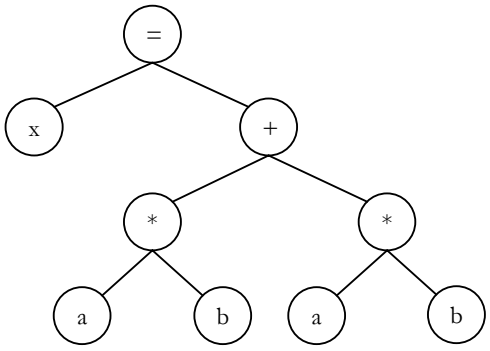


Figura 7.5.2

În general, arborele sintactic se poate defini astfel:

- Pentru o regulă sintactică de forma A → a (regulă ce produce terminali) arborele are un singur nod etichetat cu a.
- Pentru o regulă de forma A → B op C (regulă corespunzătoare operatorilor binari), arborele are o rădăcină etichetată cu op, un subarbore stâng corespunzător regulii ce rescrie pe B şi un subarbore drept corespunzător regulii ce rescrie pe C.
- Pentru o regulă de forma A → op B (regulă corespunzătoare operatorilor unari) arborele are o rădăcină etichetată cu op şi un subarbore (stâng) corespunzător regulii ce rescrie pe B.
- Pentru o regulă corespunzătoare unui operator ternar, de exemplu cea care descrie instrucţiunea if, S → `if` B `then` S `else` S, arborele sintactic are o rădăcină (etichetată cu numele operatorului, de exemplu `if`) şi trei subarbori corespunzători, în ordine, celor trei operanzi.

În acest mod, oricărui program scris într-un limbaj de programare îi putem ataşa un arbore sintactic. Acest arbore se poate obţine în procesul analizei sintactice ascendente aplicând acţiuni semantice corespunzătoare fiecărei reduceri efectuate de analizorul sintactic.

Codul cu trei adrese Această reprezentare este avantajoasă pentru cazul în care instrucţiunile limbajului intermediar au cel mult 3 variabile şi cel mult un operator. Acestea se traduc uşor în limbaj de asamblare. Reprezentarea se face prin quadruple în care, pe lângă adresele celor doi operanzi şi a rezultatului, este păstrat şi codul operaţiei. De pildă, instrucţiunea

delta := b*b – 4*a*c

se traduce prin codul:

T_1 = b*b
T_2 = 4*a
T_3 = T_2*c
T_4 = T_2 – T_3
delta = T_4

care este reprezentat de quadruplele:

	Operator	Argument 1	Argument 2	Rezultat
0	*	b	b	T_1
1	*	4	a	T_2
2	*	T_2	c	T_3
3	-	T_2	T_3	T_4

4	=	T_4	-	delta

Aici T_1, T_2,... sunt nume temporare care sunt adăugate la tabela simbolurilor pe măsură ce sunt create încât, în câmpurile relative la argumente şi rezultat, apar pointeri la tabela de simboluri corespunzătoare numelor ce apar acolo.

6.4.2 Traducerea codului în notație postfix

O schemă de traducere pentru obţinerea unui cod postfix este relativ uşor de obţinut. O să exemplificăm acest lucru pentru atribuiri cu expresii aritmetice generate de gramatica:

$$S \rightarrow A := E \qquad E \rightarrow E \text{ op } E \mid (E) \mid id \qquad A \rightarrow id$$

Dacă vom nota prin id.Loc pointerul către id în tabela de parsare (numele identificatorului) şi prin E.Cod traducerea expresiei E în cod intermediar în notaţia postfix, atunci schema de traducere este dată în tabela de mai jos, unde prin || am notat operaţia de concatenare a codurilor:

Regula sintactică	Acţiunea semantică
$S \rightarrow A := E$	S.Cod = A.Cod || E. Cod || :=
$E \rightarrow E_1$ op E_2	E.Cod = E_1.Cod || E_2.Cod ||op
$E \rightarrow (E_1)$	E.Cod = E_1.Cod
$E \rightarrow id$	E.Cod = id.Loc
$A \rightarrow id$	A.Cod = id.Loc

Această schemă de traducere poate fi implementată în cadrul unui parser bottom-up precizând care sunt acţiunile ce se aplică la fiecare reducere:

Regula sintactică	Acţiunea (segment de program)
$S \rightarrow A := E$	{ print ″:=″ }
$E \rightarrow E_1$ op E_2	{ print op }
$E \rightarrow (E_1)$	{ }
$E \rightarrow id$	{ print id.Loc }
$A \rightarrow id$	{ print id.Loc }

Traducerea textului `d := b + a*c` cu un parser ascendent este dată mai jos:

Stiva	Intrare	Acţiune
#	d := b+a*c#	Deplasare d
#d	:= b+a*c#	Reduce si **print d**
#A	:= b+a*c#	Deplasare :=
#A:=	b+a*c#	Deplasare b
#A:= b	+a*c#	Reduce si **print b**
#A:= E	+a*c#	Deplasare +
#A:= E+	a*c#	Deplasare a
#A:= E+a	*c#	Reduce si **print a**
#A:= E+E	*c#	Deplasare *
#A:= E+E*	c#	Deplasare c
#A:= E+E*c	#	Reduce si **print c**
#A:= E+E*E	#	Reduce si **print ***
#A:= E+E	#	Reduce si **print +**
#A:= E	#	Reduce si **print :=**
#S	#	ACCEPTARE

Aşadar, forma postfix a textului dat este `d b a c * + :=`.

6.4.3 Traducerea codului în arbore sintactic

Arborele de derivare (parsare) conţine, în general informaţii redundante (relativ la generarea de cod obiect) care pot fi eliminate obţinând astfel un arbore mai economic ce reprezintă codul sursă. Acest arbore se numeşte arbore sintactic (vezi figura 7.5.2).
Să arătăm cum se obţine arborele abstract pentru aceeaşi gramatică ce generează instrucţiunile de atribuire:

$S \rightarrow A := E \quad E \rightarrow E \text{ op } E \mid (E) \mid id \quad A \rightarrow id$

Să notăm prin A.Val, E.Val respectiv S.Val acest arbore corepunzător lui A, expresiilor E respectiv asignărilor S. Să considerăm `arbinar(op, st, dr)` o funcţie ce construieşte un arbore cu rădăcina `op` şi subarborii `st` respectiv `dr`, iar `frunza(id)` o funcţie ce construieşte arborele cu rădăcina `id` fără subarbori. Fiecare din aceste funcţii returnează pointeri la arborii construiţi. Atunci schema de traducere este următoarea:

Reguli sintactice	Acţiuni semantice
$S \rightarrow A := E$	S.Val = arbinar(:=, A.Val, E.Val)
$E \rightarrow E_1 \text{ op } E_2$	E.Val = arbinar(op, E_1.Val, E_2. Val)
$E \rightarrow (E_1)$	E.Val = E_1.Val
$E \rightarrow id$	E.Val = frunza(id.Loc)
$A \rightarrow id$	A.Val = frunza(id.Loc)

Traducerea aceluiaşi text `d := b + a*c`, ca în paragraful anterior, cu un parser ascendent este dată mai jos:

Stiva	Intrare	Acţiune
#	d := b+a*c#	Deplasare d
#d	:= b+a*c#	Reduce si **t_1=frunza(d)**
#A	:= b+a*c#	Deplasare :=
#A:=	b+a*c#	Deplasare b
#A:= b	+a*c#	Reduce si **t_2=frunza(b)**
#A:= E	+a*c#	Deplasare +
#A:= E+	a*c#	Deplasare a
#A:= E+a	*c#	Reduce si **t_3=frunza(a)**
#A:= E+E	*c#	Deplasare *
#A:= E+E*	c#	Deplasare c
#A:= E+E*c	#	Reduce si **t_4=frunza(c)**
#A:= E+E*E	#	Reduce si **t_5=arbinar(*,t_3,t_4)**
#A:= E+E	#	Reduce si **t_6=arbinar(+,t_2,t_5)**
#A:= E	#	Reduce si **t_7=arbinar(:=,t_1,t_6)**
#S	#	ACCEPTARE

Arborele construit este cel din figura 7.5.3. Traducerea codului obiect în arbore abstract este utilizată în proiectarea de interpretoare dar şi de compilatoare folosind generatoare de parsere yacc sau bison. In paragraful 7.6 prezentăm un exemplu de interpreter construit în acest mod.

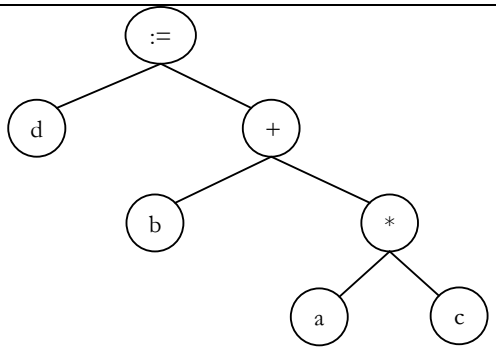


Figura 7.5.3

6.4.4 Traducerea codului în secvențe de quadruple

Codul cu trei adrese este o secvenţă de instrucţiuni simple care conţin cel mult trei referințe la o tabelă de simboluri. Odată transformat codul sursă în cod cu trei adrese, traducerea în cod maşină este relativ simplă deoarece, în general, există o corespondenţă între instrucţiunile codului intermediar cu trei adrese şi cele ale maşinii. Avantajul este că, până la acest nivel, proiectarea compilatorului nu depinde de maşina pentru care se implementează. Instrucţiunile din codul cu trei adrese se implementează ca şi structuri care conţin patru câmpuri: pe lângă cele trei adrese se reprezintă şi operatorul ce intervine în instrucţiune. Iată cele mai uzuale instrucţiuni ale codului cu trei adrese:

```
A = B op C
A = op B
goto L
if A goto L
if A rel B goto L
A = B[I]
A[I] = B
```

În acest paragraf dorim să dezvoltăm o schemă de traducere care să transforme un text sursă în cod intermediar cu trei adrese. De exemplu, codul:

```
while (a != b)
      if(a < b) then b = b - a else a = a - b
```

va fi transformat în codul intermediar următor:

```
101 if a != b goto 103
102 go to 111
103 if a < b goto 105
104 goto 108
105 T1 = b - a
106 b = T1
107 goto 101
108 T2 = a - b
109 a = T2
110 goto 101
111 ...
```

Vom considera limbajul descris de următoarea gramatică:

$S \rightarrow A$
| begin L end
| if B then S
| if B then S else S
| while B do S

$L \rightarrow L$; S

| S

A → id = E

E → E + E |E – E |E * E| E / E | - E | (E) | id

B → B or B | B and B | not B | (B)

| id < id | id > id | id == id | id != id | id

Acestei sintaxe îi vom adăuga o semantică (acțiuni semantice) care să permită traducerea programelor în cod intermediar.

Traducerea instrucţiunilor de atribuire

Vom începe descrierea schemei de traducere cu cea pentru instrucțiunile de atribuire. Intenţia este de a traduce un text de forma `delta = b*b - 4*a*c` în secvenţa:

T_1 = b * b
T_2 = 4*a
T_3 = T_2*c
T_4 = T_1- T_3
delta = T_4

Pentru a obține această traducere să definim niște atribute pentru simbolurile E şi id:

- E.Nume este numele ce păstrează valoarea expresiei E. Pentru evaluarea expresiei vom folosi nume temporare care să păstreze valorile subexpresiilor. Aceste nume le vom nota cu T_1, T_2,… şi ele sunt furnizate, în această ordine, de apelul succesiv al unei funcții pe care o notăm `newtemp()`şi care se poate implementa uşor.
- id.Nume este numele instanţei identificatorului id respectiv valoarea constantei atunci când id este instanţiat cu o constantă.

Pentru obținerea unei instrucțiuni din codul intermediar folosim o funcţie pe care o notăm GEN(I) unde I este o instrucțiune. De pildă, GEN(E.Nume = E_1.Nume + E_2.Nume) va genera instrucțiunea T_1 = b * b dacă E.Nume este T_1, E_1.Nume şi E_2.Nume sunt b. Cu acestea schema de traducere se descrie astfel:

Reguli sintactice	Acţiuni semantice
A → id = E	GEN(id.Nume = E.Nume)
E → E_1 + E_2	E.Nume = `newtemp()` GEN(E.Nume = E_1.Nume + E_2.Nume)
E → E_1 - E_2	E.Nume = `newtemp()` GEN(E.Nume = E_1.Nume - E_2.Nume)
E → E_1 * E_2	E.Nume = `newtemp()` GEN(E.Nume = E_1.Nume * E_2.Nume)
E → E_1 / E_2	E.Nume = `newtemp()` GEN(E.Nume = E_1.Nume / E_2.Nume)
E → -E_1	E.Nume = `newtemp()` GEN(E.Nume = - E_1.Nume)
E → (E_1)	E.Nume = E_1.Nume
E → id	E.Nume = id.Nume

Codul intermediar se obține în timpul parsării folosind un parser bottom-up împreună cu această schemă de traducere: de fiecare dată când se aplică o reducere (regulă sintactică), se aplică şi acțiunile semantice corespunzătoare. Vom exemplifica obținerea codului intermediar pentru instrucțiunea `delta = b*b - 4*a*c`. Cuvântul ce trebuie

analizat este id = id*id – id*id*id unde id.Nume are respectiv valorile delta, b, b, 4, a, c. Aceste valori sunt luate din tabela identificatorilor şi a constantelor construite de analizorul lexical. De fapt, id.Nume pot fi reprezentaţi ca şi pointeri la tabela simbolurilor şi pot fi păstraţi în stiva de parsare împreună cu simbolurile corespunzătoare: o intrare în stivă este o pereche (X, X.Nume). În tabela următoare este dată întreaga analiză sintactică şi semantică:

Intrare	Simbol stivă	Simbol.Nume	Acţiune	Cod
delta = b*b-4*a*c#			*deplasare*	
= b*b-4*a*c#	id	delta	*deplasare*	
b*b-4*a*c#	id=	delta_	*deplasare*	
*b-4*a*c#	id=id	delta_b	reducere	
*b-4*a*c#	id=E	delta_b	*deplasare*	
b-4*a*c#	id=E*	delta_b_	*deplasare*	
-4*a*c#	id=E*id	delta_b_b	reducere	
-4*a*c#	id=E*E	delta_b_b	reducere	T_1=b*b
-4*a*c#	id=E	delta_T_1	*deplasare*	
4*a*c#	id=E-	delta_T_1_	*deplasare*	
*a*c#	id=E-id	delta_T_1_4	reducere	
*a*c#	id=E-E	delta_T_1_4	*deplasare*	
a*c#	id=E-E*	delta_T_1_4_	*deplasare*	
*c#	id=E-E*id	delta_T_1_4_a	reducere	
*c#	id=E-E*E	delta_T_1_4_a	reducere	T_2=4*a
*c#	id=E-E	delta_T_1_ T_2	*deplasare*	
c#	id=E-E*	delta_T_1_ T_2_	*deplasare*	
#	id=E-E*id	delta_T_1_ T_2_c	reducere	
#	id=E-E*E	delta_T_1_ T_2_c	reducere	T_3= T_2*c
#	id=E-E	delta_T_1_ T_3	reducere	T_4= T_1-T_3
#	id=E	delta_T_4	reducere	delta = T_4
#	A			

Traducerea expresiilor logice

Evaluarea expresiilor logice poate fi făcută în diverse moduri. Dacă se codifică valorile de adevăr prin valori numerice (ca în limbajul C), atunci evaluarea se face analog cu evaluarea expresiilor aritmetice. În acest caz o schemă de traducere pentru expresii logice poate fi obţinută uşor din schema de traducere a expresiilor aritmetice. O altă modalitate este aceea de a traduce expresia logică într-o succesiune de instrucţiuni (cod intermediar) de forma celor care conţin goto sau if şi de a obţine valorile *adevărat* repectiv *fals* prin atingerea unor poziţii în acest cod intermediar. De exemplu, expresia:

a < b or b > c and not d

ar putea fi tradusă în codul:

```
101 if a < b goto ?
102 goto 103
103 if b > c goto 105
104 goto ?
105 if d goto ?
106 goto ?
```

cu precizarea că ieşirile (ţintele lui goto) din acest cod la liniile 101 şi 106 sunt corespunzătoare valorii *adevărat* pentru expresia dată iar cele de la 104 şi 105 sunt corespunzătoare valorii *fals*. Dacă această expresie apare în contextul unui „program”, de pildă:

if(a < b or b > c and not d) then x = x + y else x = x - y

atunci traducerea acestuia ar putea fi:

```
101 if a < b goto 107
102 goto 103
103 if b > c goto 105
104 goto 110
105 if d goto 110
106 goto 107
107 T1 = x + y
108 x = T1
109 goto 113
110 T2 = x – y
111 x = T2
112 goto 113
113 …
```

Vom indica în continuare o schemă de traducere a unei expresii logice într-o secvenţă de cod intermediar ce conţine instrucţiuni de salt condiţionat (if A goto L, if A rel B goto L) şi salt necondiţionat (goto L). În acest cod, unele din instrucţiuni vor avea ţintele L către o locaţie dintr-o mulţime notată B.True în cazul în care expresia B are valoarea *adevărat* iar altele către o locaţie dintr-o mulţime B.False în cazul în care expresia B are valoarea *fals*. B.True şi B.False vor fi aşadar atribute ale simbolului B, care au ca valori liste de pointeri către quadruplele ce formează codul intermediar. Dacă, de exemplu, avem expresia B = B_1 or B_2 atunci B.True este, deocamdată, B_1.True pentru că, odată ce am evaluat expresia B_1 şi am constatat că are valoarea *adevărat*, atunci E are valoarea *adevărat*. Dacă B_1 are valoarea *fals*, atunci trebuie evaluată expresia B_2 pentru a obţine valoarea lui B, încât B.False va fi B_1.False iar B. True este reuniunea celor două mulţimi (liste) B_1.True şi B_2.True. Aici apare următoarea problemă: in quadruplele din B_1.False trebuie ca ţinta L să aibă ca valoare pointerul la quadrupla de început al codului lui B_2. Ori acest pointer nu este cunoscut înainte de a termina generarea codului pentru B_1. Soluţia este de a genera instrucţiuni goto care nu au completată ţinta L; aceasta va fi stabilită atunci când se face o reducere cu regula B → B_1 or B_2 pentru că în acest punct se cunoaşte codul generat atât pentru B_1, cât şi pentru B_2. Asemănător se petrec lucrurile la expresiile de forma B_1 and B_2. sau la descrierea semanticii instrucţiunilor if sau while. Pentru a descrie o schemă de traducere în această idee, avem nevoie de nişte funcţii (vom păstra denumirea din AhU83):

- `makelist(i)`: funcţia creează o nouă listă de quadruple care conţine doar lista cu indexul i (pointer la tabloul quadruplelor ce se generează) şi returnează pointer la lista creată;
- `merge(`p_1`, `p_2`)`: funcţia concatenează listele pointate de p_1 şi p_2 şi returnează pointer la noua listă obţinută;
- `backpatch(p, i)`: funcţia completează cu i ţintele instrucţiunilor goto din quadruplele listei pointată de p.

Vom nota de asemenea cu NEXTQUAD un pointer ce păstrează, la un moment dat, numărul primei quadruple ce trebuie generată, şi care va permite identificarea punctului de intrare în codul unei subexpresii. Acest lucru se poate realiza prin introducerea unui neterminal M în sintaxa expresiilor, care să marcheze începutul codului unei subexpresii prin atributul M.Quad (=NEXTQUAD). Astfel, regulile sintactice B → B or B şi B → B and B se transformă în B → B or M B, B → B and M B şi M → ε. Cu aceste pregătiri, schema de traducere pentru expresii logice este următoarea (am notat prin rel un operator relaţional):

Reguli sintactice	Acţiuni semantice
$B \rightarrow B_1$ or M B_2	backpatch(B_1.False, M.Quad); B.True = merge(B_1.True, B_2.True); B.False = B_2.False;
$B \rightarrow B_1$ or M B_2	backpatch(B_1.True, M.Quad); B.True = B_2.True; B.False = merge(B_1.False, B_2.False);
$B \rightarrow$ not B_1	B.True = B_1.False ; B.False = B_1.True;
$B \rightarrow (B_1)$	B.True = B_1.True; B.False = B_1.False;
$B \rightarrow id_1$ rel id_2	B.True = makelist(NEXTQUAD); B.False = makelist(NEXTQUAD + 1); GEN(if id_1.Nume rel id_2.Nume go to …); GEN(go to …);
$B \rightarrow$ id	B.True = makelist(NEXTQUAD); B.False = makelist(NEXTQUAD + 1); GEN(if id.Nume go to …); GEN(go to …);
$M \rightarrow \varepsilon$	M.Quad = NEXTQUAD;

Să observăm că am tratat aici un caz particular, când expresiile relaţionale sunt doar de forma id rel id. Cititorul este invitat să trateze cazul general al sintaxei $B \rightarrow E$ rel E unde E este o expresie aritmetică.

Dacă realizăm analiza sintactică a expresiei

a < b or b > c and not d

şi aplicăm acţiunile semantice la fiecare reducere, obţinem rezultatele din tabela de mai jos. Pentru variabila B vom nota prin (q1, q2, …; p1, p2, …) valorile atributelor B.True respectiv B.False.

Intrare	Simbol stivă	Valorile atributelor	Cod
a < b or b > c and not d #			
< b or b > c and not d #	id	a	
b or b > c and not d #	id <	a_	
or b > c and not d #	id < id	a_b	(1)
or b > c and not d #	B	(101; 102)	
b > c and not d #	B or	(101; 102) _	
b > c and not d #	B or M	(101; 102) _103	
> c and not d #	B or M id	(101; 102) _103b	
c and not d #	B or M id >	(101; 102) _103b_	
and not d #	B or M id > id	(101; 102) _103b_c	(2)
and not d #	B or M B	(101; 102) 103(103; 104)	
not d #	B or M B and	(101; 102) 103(103; 104)_	
not d #	B or M B and M	(101; 102) 103(103; 104)_105	
d #	B or M B and M not	(101; 102) 103(103; 104)_105_	
#	B or M B and M not id	(101; 102) 103(103; 104)_105_d	(3)
#	B or M B and M not B	(101; 102) 103(103; 104)_105_ (105;106)	
#	B or M B and M B	(101; 102) 103(103; 104)_105 (106;105)	(4)
#	B or M B	(101; 102) 103(106; 104,105)	(5)
#	B	(101, 106; 104,105)	

(1) 101 if a<b goto…
102 goto…

(2) 103 if b < c goto…
104 goto…
(3) 105 if d goto…
106 goto…
(4) În lista{103} se pune 105 ca ţintă pentru goto
Se determină conform acţiunilor semantice B.True şi B.False
(5) În lista{102} se pune 103 ca ţintă pentru goto
Se determină conform acţiunilor semantice B.True şi B.False

Urmare a acestor operaţii codul obţinut este:

101 if a < b goto …
102 goto 103
103 if b > c goto 105
104 goto …
105 if d goto …
106 goto …

iar listele ataşate lui B, corespunzătoare valorilor *adevărat* respectiv *fals*, sunt:
B.True = {101, 106}, B. False = {104, 105}.

Traducerea instrucţiunilor

Să dezvoltăm mai întâi o schemă de traducere pentru instrucţiunile de control. Dacă ar trebui să traducem o instrucţiune de forma **if B then S else S**, atunci codul obţinut trebuie să conţină codul pentru B, urmat de codul pentru primul S apoi codul pentru al doilea S. În determinarea codului pentru B se obţin, aşa cum am văzut mai sus, listele B.True şi B.False. Ţintele pentru goto din B.True trebuie să coincidă cu eticheta de început a codului pentru primul S iar cele din B.False trebuie să coincidă cu eticheta de început a codului celui de-al doilea S. De asemenea, în codul obţinut trebuie să precizăm şi faptul că, după execuţia instrucţiunilor desemnate de S (atât primul, cât şi al doilea) urmează instrucţiunea de după if. De aceea vom considera un atribut S.Next care reprezintă un pointer la o listă de quadruple ce conţin goto, condiţionat sau nu, la instrucţiunea ce urmează după S în ordinea execuţiei. Mecanismul de completare a acestor ţinte va fi cel de la traducerea expresiilor logice încât va trebui şi aici să considerăm un neterminal ce marchează începutul codului lui S şi un neterminal N care printr-un atribut N.Next permite saltul după execuţia ramurii then a instrucţiunii if. Aşadar, schema de traducere pentru instrucţiunile if se poate descrie astfel:

1. $S \rightarrow$ if B then M_1 S_1 N else M_2 S_2

```
backpatch(B.True, M1.Quad);
backpatch(B.False, M2.Quad);
S.Next = merge(S1.Next, N.Next, S2.Next);
```

2. $N \rightarrow \varepsilon$

```
N.Next = makelist(NEXTQUAD);
GEN(goto…);
```

3. $M \rightarrow \varepsilon$

```
M.Quad = NEXTQUAD;
```

4. $S \rightarrow$ if B then M S_1

```
backpatch(B.True, M.Quad);
S.Next = merge(B.False, S1.Next);
```

În mod analog tratăm cazul instrucţiunii while. Dacă sintaxa este **while B do S** atunci va trebui să stabilim în quadruplele din B.True din codul lui B, ca ţintă, începutul codului lui S, iar în quadruplele din B.False ţinta trebuie să fie începutul codului instrucţiunii ce urmează lui while. De asemenea, trebuie să ne asigurăm că, după execuţia lui S, se reia evaluarea lui B. Atunci, traducerea acestei instrucţiuni se face după schema:

5. $S \rightarrow$ while M_1 B do M_2 S_1

```
backpatch(S1.Next, M1.Quad);
backpatch(B.True, M2.Quad);
S.Next = B.False;
GEN(goto M1.Quad);
```

Să observăm că apare pentru prima dată generarea unei instrucţiuni goto care are ţinta fixată: goto M_1.Quad. Asta asigură faptul că, în toate cazurile, după execuţia lui S_1 urmează evaluarea din nou a expresiei B.
Pentru traducerea celorlalte instrucţiuni din limbajul specificat la începutul paragrafului lucrurile sunt cât se poate de simple:

6. $S \rightarrow$ begin L end

```
S.Next = L.Next;
```

7. $S \rightarrow A$

```
S.Next = makelist(); // lista vidă
```

8. $L \rightarrow L_1$; M S

```
backpatch(L1.Next, M.Quad);
L.Next = S.Next;
```

9. $L \rightarrow S$

```
L.Next = S.Next;
```

Să mai observăm că doar în cazurile $N \rightarrow \varepsilon$ şi la instrucţiunea while sunt generate noi quadruple; codul este asociat în rest expresiilor aritmetice sau logice. Traducerea instrucţiunilor înseamnă în plus asocierea ţintelor pentru goto care au fost generate fără a fi complete (apelul funcţiei backpatch).
Dacă aplicăm un parser bottom-up pentru textul:

```
while (a != b)
  if(a < b)then
      b = b - a
  else
      a = a - b
```

şi realizăm inclusiv traducerea sa conform schemei de traducere prezentată, obţinem un cod intermediar de forma celui anunţat la începutul acestui paragraf:

```
101 if a != b goto 103
102 go to 111
103 if a < b goto 105
104 goto 108
105 T1 = b - a
106 b = T1
107 goto 101
108 T2 = a - b
109 a = T2
110 goto 101
111 …
```

6.5 Proiectarea unui interpreter cu flex şi yacc

Vom descrie în acest paragraf modul cum se poate proiecta un interpreter folosind instrumentele flex şi bison. Dorim să interpretăm programe de forma:

```
a = 36;
b = 42;
while (a != b)
  if(a > b) a = a - b;
  else b = b - a;
print a;
```

Pentru aceasta vom descrie fişierele de intrare pentru flex respectiv bison. Fişierul pentru flex descrie unităţile lexicale care vor fi transmise analizorului sintactic construit de bison. Fisierul flex este următorul:

```
/* Fisierul glang.l intrare pentru flex. Se foloseste comanda
      flex glang.l dupa comanda bison –d glang.y   */
%{
#include <stdlib.h>
#include "glang.h"
#include "glang.tab.h"
void yyerror(char *);
%}

%%

[a-z]       { yylval.sIndex = *yytext - 'a'; return VAR; }

[0-9]+      { yylval.iVal = atoi(yytext);  return INT; }
[-()<>=+*/;{}.] { return *yytext;}
">="            { return GE;   }
"<="            { return LE;   }
"=="            { return EQ;   }
"!="            { return NE;   }
"while"         { return WHILE;      }
"if"            { return IF;   }
"else"          { return ELSE;       }
"print"         { return PRINT;      }

[ \t\n]+        ; /* Spatiile sunt ignorate */

.               yyerror(" Caracter ilegal!\n");
%%
int yywrap(void) {
    return 1;
}
```

Fişierul glang.h descrie structurile de date folosite pentru reprezentarea constantelor, a identificatorilor precum şi a tipurilor de noduri din arborele abstract care va fi costruit de anlizorul sintactic. Identificatorii sunt definiţi aici ca fiind litere şi de aceea am folosit `extern int sym[26]` ca tabelă a identificatorilor. O să indicăm mai târziu cum se poate aborda cazul general.

```
/* Fisierul glang.h     */
typedef enum { typeCon, typeId, typeOper } nodeEnum;

/* constante */
typedef struct {
    int val;            /* valoarea constantei */
} conNodeType;

/* identificatori */
```

```
typedef struct {
    int i;        /* indice in tabela de simboluri */
} idNodeType;

/* operatori */
typedef struct {
    int oper;                   /* operator */
    int nops;                   /* nr. operanzi */
    struct nodeTypeTag *op[1];  /* operanzi */
} oprNodeType;

typedef struct nodeTypeTag {
    nodeEnum type;              /* tipul nodului */
    union {
        conNodeType con;        /* constante */
        idNodeType id;          /* identificatori */
        oprNodeType opr;        /* operatori */
    };
} nodeType;
extern int sym[26];
```

Fişierul glang.tab.h este obţinut de către yacc (bison) atunci când se foloseşte opţiunea –d. Acest fişier defineşte codurile numerice pentru unităţile lexicale:

```
/* Fisierul glang.tab.h creat de bison cu optiunea -d  */
# ifndef BISON_GLANG_TAB_H
# define BISON_GLANG_TAB_H
#ifndef YYSTYPE
typedef union {
    int iVal;             /* valoare intreaga */
    char sIndex;          /* index in tabela de simboluri */
    nodeType *nodPtr;     /* pointer la un nod in arbore */
} yystype;
# define YYSTYPE yystype
# define YYSTYPE_IS_TRIVIAL 1
#endif
# define    INT   257
# define    VAR   258
# define    WHILE 259
# define    IF    260
# define    PRINT 261
# define    IFX   262
# define    ELSE  263
# define    GE    264
# define    LE    265
# define    EQ    266
# define    NE    267
# define    UMINUS      268

extern YYSTYPE yylval;
#endif /* not BISON_GLANG_TAB_H */
```

Fişierul de intrare pentru bison descrie sintaxa limbajului de programare împreună cu acţiunile semantice ce trebuiesc aplicate în cazul reducerilor. Aici este vorba de apelul funcţiei de construire a arborelui abstract corespunzător. Iată mai jos fişierul de intrare pentru bison:

```
/* Fisierul glang.y intrare pentru bison. Se foloseste
   comanda : bison -d glang.y         */
%{
```

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "glang.h"

/* prototipurile functiilor */
nodeType *nodOper(int oper, int nops, ...);
            /* nod operator in arbore     */
nodeType *nodId(int i);
            /* nod frunza identificator   */
nodeType *nodCon(int value);
            /* nod frunza constanta       */
void freeNode(nodeType *p);
            /* eliberare memorie          */
int interpret(nodeType *p);
            /* functia de interpretare    */
int yylex(void);
            /* functia creata de flex     */

void yyerror(char *s);
int sym[26];                /* tabela de simboluri */
%}

%union {
    int iVal;               /* valoare intreaga */
    char sIndex;            /* index in tabela de simboluri */
    nodeType *nodPtr;       /* pointer la un nod in arbore */
};

%token <iVal> INT
%token <sIndex> VAR
%token WHILE IF PRINT
%nonassoc IFX     /* rezolvarea ambiguitatii if-then-else */
%nonassoc ELSE
%left GE LE EQ NE '>' '<'
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <nodPtr> stmt expr stmt_list

%%
program:  function           { exit(0); }
        ;
function: function stmt      { interpret($2); freeNode($2); }
        | /* NULL */
        ;

stmt:     ';'           { $$ = nodOper(';', 2, NULL, NULL); }
        | expr ';'         { $$ = $1; }
        | PRINT expr ';'   { $$ = nodOper(PRINT, 1, $2); }
        | VAR '=' expr ';'
                  {$$ = nodOper('=', 2, nodId($1), $3);}
        | WHILE '(' expr ')' stmt
                  { $$ = nodOper(WHILE, 2, $3, $5); }
        | IF '(' expr ')' stmt %prec IFX
                  { $$ = nodOper(IF, 2, $3, $5); }
        | IF '(' expr ')' stmt ELSE stmt
                  { $$ = nodOper(IF, 3, $3, $5, $7); }
        | '{' stmt_list '}' { $$ = $2; }
        ;
```

```
stmt_list:
          stmt              { $$ = $1; }
        | stmt_list stmt    { $$ = nodOper(';', 2, $1, $2); }
        ;
expr:
          INT              { $$ = nodCon($1); }
        | VAR              { $$ = nodId($1); }
        | '-' expr %prec UMINUS
                           { $$ = nodOper(UMINUS, 1, $2); }
        | expr '+' expr
                           { $$ = nodOper('+', 2, $1, $3); }
        | expr '-' expr
                           { $$ = nodOper('-', 2, $1, $3); }
        | expr '*' expr
                           { $$ = nodOper('*', 2, $1, $3); }
        | expr '/' expr
                           { $$ = nodOper('/', 2, $1, $3); }
        | expr '<' expr
                           { $$ = nodOper('<', 2, $1, $3); }
        | expr '>' expr
                           { $$ = nodOper('>', 2, $1, $3); }
        | expr GE expr
                           { $$ = nodOper(GE, 2, $1, $3); }
        | expr LE expr
                           { $$ = nodOper(LE, 2, $1, $3); }
        | expr NE expr
                           { $$ = nodOper(NE, 2, $1, $3); }
        | expr EQ expr
                           { $$ = nodOper(EQ, 2, $1, $3); }
        | '(' expr ')'
                           { $$ = $2; }
        ;
%%

#define SIZEOF_NODETYPE ((char *)&p->con - (char *)p)

nodeType *nodCon(int value) {
    nodeType *p;
    size_t nodeSize;

    /* alocare memorie pentru noul nod */
    nodeSize = SIZEOF_NODETYPE + sizeof(conNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copiere valoare constanta */
    p->type = typeCon;
    p->con.val = value;

    return p;
}

nodeType *nodId(int i) {
    nodeType *p;
    size_t nodeSize;

    /* alocare memorie pentru noul nod */
    nodeSize = SIZEOF_NODETYPE + sizeof(idNodeType);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");
```

```
    /* copiere valoare indice */
    p->type = typeId;
    p->id.i = i;

    return p;
}

nodeType *nodOper(int oper, int nops, ...) {
    va_list ap;
    nodeType *p;
    size_t nodeSize;
    int i;
    /* allocare memorie pentru noul nod */
    nodeSize = SIZEOF_NODETYPE + sizeof(oprNodeType) +
        (nops - 1) * sizeof(nodeType*);
    if ((p = malloc(nodeSize)) == NULL)
        yyerror("out of memory");

    /* copiere informatii functie de nops */
    p->type = typeOper;
    p->opr.oper = oper;
    p->opr.nops = nops;
    va_start(ap, nops);
    for (i = 0; i < nops; i++)
        p->opr.op[i] = va_arg(ap, nodeType*);
    va_end(ap);
    return p;
}

void freeNode(nodeType *p) {
    int i;

    if (!p) return;
    if (p->type == typeOper) {
        for (i = 0; i < p->opr.nops; i++)
            freeNode(p->opr.op[i]);
    }
    free (p);
}

void yyerror(char *s) {
    fprintf(stdout, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

Conform regulilor următoare (din fişierul tocmai prezentat):

```
function: function stmt
          { interpret($2); freeNode($2); }
| /* NULL */
;
```

un program în limbajul sursă descris este un şir de instrucţiuni (stmt). Fiecare din aceste instrucţiuni este interpretată: apelul funcţiei `interpret` pentru argumentul `$2` care este valoarea atributului lui stm, adică arborele construit pentru stmt, face ca această instrucţiune să fie executată. Prin `freeNode($2)` se eliberează memoria alocată la construirea arborelui amintit. Funcţia `interpret()` este prezentată mai jos :

```
/* Functia interpret(nodeType *p)  */
#include <stdio.h>
#include "glang.h"
#include "glang.tab.h"

int interpret(nodeType *p) {
    if (!p) return 0;
    switch(p->type) {
    case typeCon:       return p->con.val;
    case typeId:        return sym[p->id.i];
    case typeOper:
        switch(p->opr.oper) {
        case WHILE:     while(interpret(p->opr.op[0]))
                         interpret(p->opr.op[1]); return 0;
        case IF:        if (interpret(p->opr.op[0]))
                            interpret(p->opr.op[1]);
                        else if (p->opr.nops > 2)
                            interpret(p->opr.op[2]);
                        return 0;
        case PRINT:  printf("%d\n", interpret(p->opr.op[0]));
                     return 0;
        case ';':    interpret(p->opr.op[0]);
                     return interpret(p->opr.op[1]);
        case '=':    return sym[p->opr.op[0]->id.i] =
                        interpret(p->opr.op[1]);
        case UMINUS: return -interpret(p->opr.op[0]);
        case '+':    return interpret(p->opr.op[0]) +
                        interpret(p->opr.op[1]);
        case '-':    return interpret(p->opr.op[0]) -
                        interpret(p->opr.op[1]);
        case '*':    return interpret(p->opr.op[0]) *
                        interpret(p->opr.op[1]);
        case '/':    return interpret(p->opr.op[0]) /
                        interpret(p->opr.op[1]);
        case '<':    return interpret(p->opr.op[0]) <
                        interpret(p->opr.op[1]);
        case '>':    return interpret(p->opr.op[0]) >
                        interpret(p->opr.op[1]);
        case GE:     return interpret(p->opr.op[0]) >=
                        interpret(p->opr.op[1]);
        case LE:     return interpret(p->opr.op[0]) <=
                        interpret(p->opr.op[1]);
        case NE:     return interpret(p->opr.op[0]) !=
                        interpret(p->opr.op[1]);
        case EQ:     return interpret(p->opr.op[0]) ==
                        interpret(p->opr.op[1]);
        }
    }
    return 0;
}
```

În urma lansării comenzilor bison –d glang.y şi flex glang.l se obţin fişierele lexyy.c (analizorul lexical) şi glang.tab.c (analizorul sintactic) care, împreună cu interpreterul din fişierul interpret.c, vor fi compilate cu un compilator de limbaj C şi se obţine astfel un interpreter pentru limbajul sursă descris. Dacă lansăm acest interpreter pentru programul prezentat la începutul secţiunii, se obţine rezultatul 6.

Bibliografie

[AnG97] Andrei Şt., Grigoraş Gh. : *Tehnici de compilare. Lucrări de laborator*. Editura Universităţii „Al.I.Cuza", Iaşi, 1995

[App97] Appel, A., *Modern Compiler Implementation in C*, Cambridge University Press, 1997

[ASU86] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, U.S.A., 1986

[Gri86] Grigoraş, Gh.: *Limbaje formale şi tehnici de compilare*. Editura Universităţii „Al.I.Cuza", Iaşi, 1986

[Gri05] Grigoraş, Gh.: Constructia compilatoarelor - Algoritmi fundamentali, Editura Universitatii "Al. I. Cuza" Iasi, ISBN 973-703-084-2, 274 pg., 2005.

[JuA97] Jucan, T, Andrei, Şt.: *Limbaje formale şi teoria automatelor. Culegere de probleme*. Editura Universităţii „Al.I.Cuza", Iaşi, 1997

[Juc99] Jucan, T., *Limbaje formale şi automate*, Editura MatrixRom, 1999.

[Şer97] Şerbănaţi, L.D., *Limbaje de programare şi compilatoare*, Editura Academiei, Bucureşti, 1987.