Universitatea Ovidius Constanta

Lucrare de licenţă

Elemente de teoria compilarii si aplicaţii

Autor:
Neculai Stanciu

Îndrumator:
Lector dr. Dragoş Sburlan

O lucrare creată in scopul obţinerii cerinţelor
pentru obţinerea diplomei de licenţa

în domeniul informatica

Iunie 2013

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

UNIVERSITATEA OVIDIUS CONSTANTA

Elemente de teoria compilării

Facultatea de Matematica si Informatica
informatica

licenţa

de Neculai STANCIU

Mulţumiri

The acknowledgements and the people to thank go here, don't forget to include your project advisor…

Cuprins

Capitolul 1

Introducere

1.1 De ce teoria compilarii

Prima oara cand am auzit despre ce inseamna un compilator a fost in liceu cand am invatat despre limbajul de programare Pascal.De atunci pana in prezent am invatat o multime de alte limbaje.Cu toate acestea nu stim cu adevarat ce inseamna un limbaj de programare desi pe unele dintre ele stiam sa le folosesc.Desi aveam cunostinte de limbaje formale tot nu puteam sa imi dau seama cum sunt imbinate acele modele matematice (gramatici,automatele finite) intr-o aplicatie care sa generez cod.

Capitolul 2

Preliminare

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

$$L = \{w \in \{0, 1\}^* / |w| mod 3 = 0\}$$

este un alt mod de a specifica acelaşi limbaj.

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

2.1 Gramatici

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

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

2.1.1 Transformări asupra gramaticilor independente de context

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

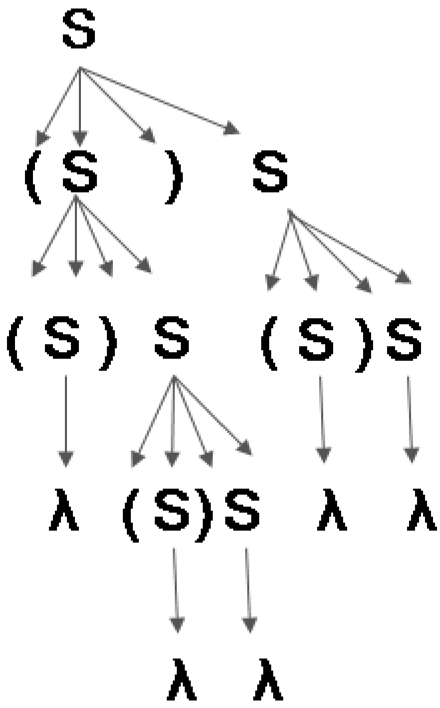
Un arbore de derivare este o reprezentare grafică pentru o secvenţă de derivări (de aplicări ale relaţiei $\Rightarrow$ între formele propoziţionale). Într-un arbore de derivare nu se mai poate identifica ordinea în care s-a făcut substituţia simbolilor neterminali. Fiecare nod interior arborelui, reprezintă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

$G = (\{S\}, \{(,)\}, \{S(S)S|\lambda\}, S)$

Fie următoarea secvenţă de derivări:

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

Se obţine arborele de derivare:



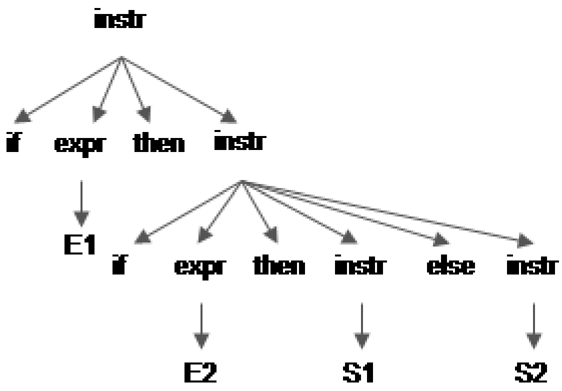
2.1.2 Eliminarea ambiguităţii

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

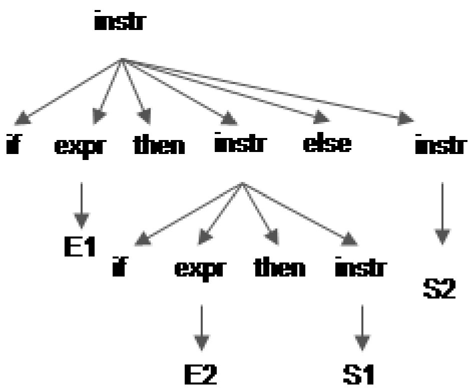
```
instr  if expresie then instr |
 if expresie then instr else instr |
 alte_instr
```

Să construim arborele de derivare pentru propoziţia :

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



Pentru această propoziţie mai există însă un arbore de derivare.

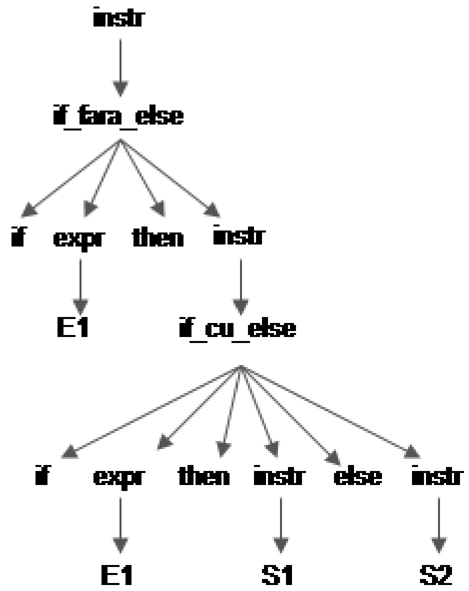


În toate limbajele de programare care acceptă construcţii de tip if then else se consideră cu sens prima derivare în care fiecare clauza else este atribuită instrucţiunii if cea mai interioară. Rezultă deci condiţia pe care trebuie să o satisfacă o instrucţiune if. Instrucţiunea cuprinsă între then şi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

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

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

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



Se numeşte producţie ambiguă o producţie care are în partea dreaptă mai multe apariţii ale aceluiaşi simbol neterminal.Existenţa unei producţii ambigue nu implică faptul c gramatica este ambiguă.

2.1.3 Eliminarea recursivităţii stânga

O gramatică este recursivă stânga dacă există un neterminal A astfel încât există o derivare A $\Rightarrow^*$ Aβ pentru $\beta \in (T \cup N)^*$. O analiză sintactică descendentă deterministă nu poate să opereze cu o astfel de gramatică, deci este necesară o transformare. Să considerăm întâi cazul cel mai simplu pentru care în gramatică există producţii de forma $A \rightarrow A\beta|\alpha$. În acest caz limbajul generat este de forma $\alpha\beta$ cu n 0. Acelaşi limbaj poate sâ fie generat de către gramatica: A αA', A' βA'— λ.
Să considerăm de exemplu gramatica expresiilor aritmetice :
$E \rightarrow E + T|T, T \rightarrow T * F|F, F \rightarrow (E)|a$
Se observă ca pentru un şir de forma a+a*a,examinand numai primul simbol terminal (a) nu este clar cu ce producţie dintre producţiile pentru E trebuie să se înceapă derivarea.Aplicând ideea anterioară se obţine:
$E \rightarrow TE', E' \rightarrow +TE'|\lambda, T \rightarrow FT', T' \rightarrow *FT'|\lambda, F \rightarrow (E)|a$
În acest caz derivarea va începe sigur prin aplicarea producţiei TE' şi se obţine derivarea $\Rightarrow$ TE' $\Rightarrow$ FT''. În acest moment se vede că pentru F trebuie să se aplice producţia F $\rightarrow$ a. Deci se obt̆ine $\Rightarrow$ aT''. Urmează simbolul terminal + datorită căruia pentru T' se va aplica producţia T' λ, etc.

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

Se aranjează neterminalele în ordinea A1, ..., An
pentru i = 1 până la n executa
pentru j = 1 până la i - 1 executa
înlocuieşte fiecare producţie de forma $A_i \rightarrow A_j\beta$ *cu producţiile*
$A_i \rightarrow \alpha_1\beta|\alpha_2\beta|...|\alpha_k\beta$
unde $A_j \rightarrow \alpha_1|\alpha_2|\alpha_3|...|\alpha_k$
sunt toate producţiile pentru A_j
elimină recursivitatea stângă între producţiile A_i

2.1.4 Factorizare stânga

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

Să presupunem că încercăm să construim şirul derivărilor pentru a b a c a pornind de la simbolul de start al gramaticii. Din recunoaşterea simbolului a la începutul şirului nu se poate încă trage concluzia care dintre cele doua producţii corespunzătoare neterminalului S trebuie să fie luata în considerare (abia la întâlnirea caracterului b pe şirul de intrare se poate face o alegere corectă). În general pentru producţia $A \rightarrow \alpha\beta_1|\alpha\beta_2$ dacă se recunoaşte la intrare un şir nevid derivat din α nu se poate ştii dacă trebuie aleasă prima sau a doua producţie. Corespunzător este utilă transformarea: A $\rightarrow \alpha$ A', A'$\rightarrow \beta_1|\beta_2$.
Algoritmul de factorizare funcţionează în modul următor. Pentru fiecare neterminal A se caută cel mai lung prefix α comun pentru două sau mai multe dintre producţiile corespunzătoare neterminalului A. Dacă $\alpha \neq \lambda$ atunci se înlocuiesc producţiile de forma $A \rightarrow \alpha\beta_1$—$\alpha\beta_2$— ... — $\alpha\beta_n$— δ (unde δ reprezintă alternativele care nu încep cu α) cu :
A $\rightarrow \alpha A'\delta$ A' $\rightarrow \beta_1|\beta_2|...|\beta_n$ A' este un nou neterminal. Se aplică în mod repetat această transformare până când nu mai există două alternative producţii cu un prefix comun

pentru acelaşi simbol neterminal. Reluând exemplul considerat se obţine :
S → AX
X → bS | λ
A → BY
Y → cA | λ
B → a | dSd

Deci în analiza şirului a b a la întâlnirea simbolului b pentru neterminalul Y se va utiliza producţia Y → λ, în acest mod rezultă şirul de derivări :
S ⇒ AX ⇒ BYX ⇒ aYX ⇒ ...

2.2 Automate finite

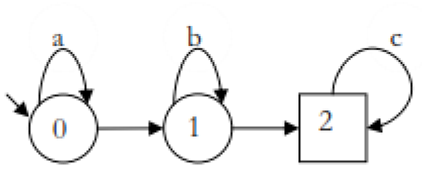
Un automat finit este sistemul A=(Q,Σ,δ,q_0,F) unde Q si F sunt nulţimi,nevide, numite mulţimea starilor respectiv *alfabetul de intrare*, $q_0 \in Q$ este *starea initiala*,F⊆Q este *multimea starilor finale* iar δ este o funcţie δ : $Qx(\Sigma \cup \varepsilon) \rightarrow 2^Q$, numita *funcţia de tranziţie* (unde prin 2^Q s-a notat mulţimea parţilor lui Q).

Modelul prezentat mai sus este cel cunoscut in literatura şi sub denumirea de automat nedeterminist cu ε - tranziţii. 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 starile care nu sunt finale sa le reprezentăm prin cercuri iar cele finale prin pătrate.De asemenea ε -tranziţiile sunt reprezentate prin arce neetichetate.
Exemple de automate
Fie Q={0,1,2 } Σ ={a,b,c } F={2},q_0=0,iar δ este dată astfel:

Tabela de tranziţie:

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

Graful de tranziţie:

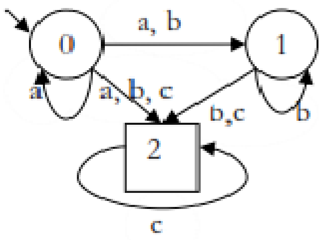


Fie Q={0,1,2 },Σ={a,b }, F={2 },q_0=0,iar δ este:

Tabela de tranziţie:

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

Graful de tranziţie:



Definitie Un automat A=(Q,Σ,δ,q_0,F) se numeste:

1. *nedeterminist* (fara ε - tranzitii),daca $\delta(q,\varepsilon) = \emptyset$,$\forall q \in Q$
2. *determinist*,dacă $\delta(q,\varepsilon) = \emptyset$,$\forall q \in Q$ şi $|\delta(q,a)| \leq 1$, $\forall q \in Q$,$\forall a \in \Sigma$

2.3 Expresii regulate

Fie Σ un alfabet, simbolurile $\varepsilon, \emptyset, |, \bullet, *,), ($ care nu aparţin lui Σ şi E un cuvânt peste alfabetul $\Sigma \cup \{\varepsilon, \emptyset, |, \bullet, *,), (\}$.O expresie regulată peste Σ se defineşte inductiv astfel:

1. E este un *atom regulat* peste Σ daca E este un simbol $\Sigma \cup \{\varepsilon, \emptyset\}$ sau este de forma (E_1) unde E_1 este o expresie regulata peste Σ;
2. E este *factor regulat* peste Σ daca E este un atom regulat peste 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 E_1 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 $E_1|E_2$, unde E_1 este o expresie regulat, iar E_2 este un termen regulat peste Σ.

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

Capitolul 3

Algoritmi de parsare

3.1 Problema parsării

Problema recunoaşterii în gramatici independente de context este următoarea: Dată o gramatică G = (V, T, S, P) şi un cuvânt w$\in T^*$, care este răspunsul la întrebarea w$\in$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$\in$L(G) este afirmativ, se cere arborele sintactic (o reprezentare a sa) pentru w.

3.2 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 3.1):

- 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.

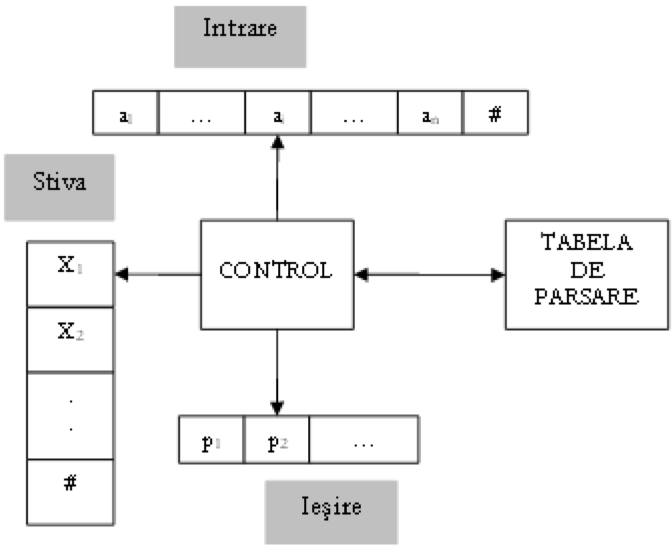


FIGURE 3.1: Reprezentarea unui parser descendent

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.

3.2.1 Gramaticile LL(1)

Pentru ca un parser SLL(k) să poat 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 $FIRST_1$ i $FOLLOW_1$ prin FIRST respectiv FOLLOW. Aşadar, dacă $\alpha \in \Sigma^+$, A$\in$V :
FIRST(α) = $\{a|a \in$T,$\alpha \overset{*}{\underset{st}{\Rightarrow}} au\} \cup$ if ($\alpha \overset{*}{\underset{st}{\Rightarrow}} \varepsilon$) then ε else $\emptyset$.
FOLLOW(A) = $\{a|a \in T \cup \varepsilon, S \overset{*}{\underset{st}{\Rightarrow}} uA\gamma,\ a \in FIRST(\gamma)\}$ Mai întâi să arătăm că gramaticile SLL(1) coincid cu gramaticile LL(1).

Theorem 3.1. *O gramatică G = (V, T, S, P) este gramatică LL(1) dacă şi numai dacă pentru orice $A \in V$ şi pentru orice producţii $A \to \beta_1|\beta_2$ are loc:*

FIRST (β_1*FOLLOW (A)*)$\cap$ *FIRST*(β_2*FOLLOW(A)*)$=\emptyset$

3.2.2 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.
Intrare: Gramatica G=(V,T,S,P) redusă;
Iesire: FIRST(X),X$\in \Sigma$.
Metoda: Mulţimile FIRST sunt completate prin inspectarea regulilor gramaticii

1.for (X $\in$ Σ)

2. if (X $\in$T) FIRST(X) = X else FIRST(X)=$\emptyset$;

3.for (A $\to$ α β $\in$ P)

4. FIRST (A) = FIRST(A) $\cup$ $\{$ a $\}$;

5.FLAG = true;

6.while(FLAG) $\{$ // FLAG marcheaza schimbarile in FIRST

7. FLAG = false;

8. for(A $\to$ $X1$ $X2$ $\ldots$ Xn $\in$ P) $\{$

9. i = 1;

10. if((FIRST(X1)$\not\subseteq$FIRST(A)) $\{$

11. FIRST(A) = FIRST(A)$\cup$(FIRST($X1$);

12. FLAG = true;

$\}$//endif

13. while(i<n && Xi$\underset{st}{\overset{+}{\Rightarrow}}\varepsilon$)

14. if((FIRST(Xi+1) $\not\subseteq$ FIRST(A)) $\{$

15. FIRST(A) = FIRST(A)$\cup$FIRST(Xi+1);

16. FLAG = true; i++;

$\}$//endif

$\}$//endwhile

$\}$//endfor

$\}$//endwhile

17.for (A$\in$ V)

18.if(A $\underset{st}{\overset{+}{\Rightarrow}}\varepsilon$) FIRST(A) = FIRST(A)$\cup$$\{$ ε $\}$;

Algoritmul pentru descrierea mulţimilor FOLLOW:

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

Iesire: Mulţimile FOLLOW(A),A $\in$ V

Metoda:

```
1.for (A∈ Σ) FOLLOW(A) =∅ ;
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
```

3.2.3 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 : $V \times (T \cup \{ \# \}) \to \{(\beta,p)—p = A \to \beta \in P \} \cup \{eroare\}$

construită după algoritmul următor:

Intrare Gramatica G = (V, T, S,P); Mulţimile FIRST(β), FOLLOW(A), A$\to \beta \in$P.

Iesire 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)= ∅ ;
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( β ){
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)=∅) M(A,a)={eroare};
```

3.2.4 Analizatorul sintactic LL(1)

În continuare vom prezenta algoritmul de analiză sintactică LL(1):

Intrare Gramatica G = (V,T,S, P).

Tabela de analiză LL(1) notată M.

Cuvântul de intrare w#.

Iesire Analiza sintactică stângă π a lui w dacă w $\in$ L(G),eroare în caz contrar.

Metoda Sunt implementate tranziţiile 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")
14.{write("eroare"); exit(1);}
15.else {
// M(X,a)=(β,r), r=X→β ,β=Y1Y2...Yn
//β inlocuieste 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);
```

Exemplu de rulare a algoritmului pe o gramatică: Fie gramatica:

1. S $\rightarrow$ E
2. S $\rightarrow$ B
3. E $\rightarrow$ ε
4. B $\rightarrow$ a
5. B $\rightarrow$ begin SC end
6. C $\rightarrow$ ε
7. C $\rightarrow$;SC

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 gramatica 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

In continuare urmează 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 Analiza sintactică în gramatici LR

Vom prezenta în continuare 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: *Left to right scanning of the input, constructing a Rightmost derivation in reverse, using k symbols lookahead.* Aceast 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 analiz 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.

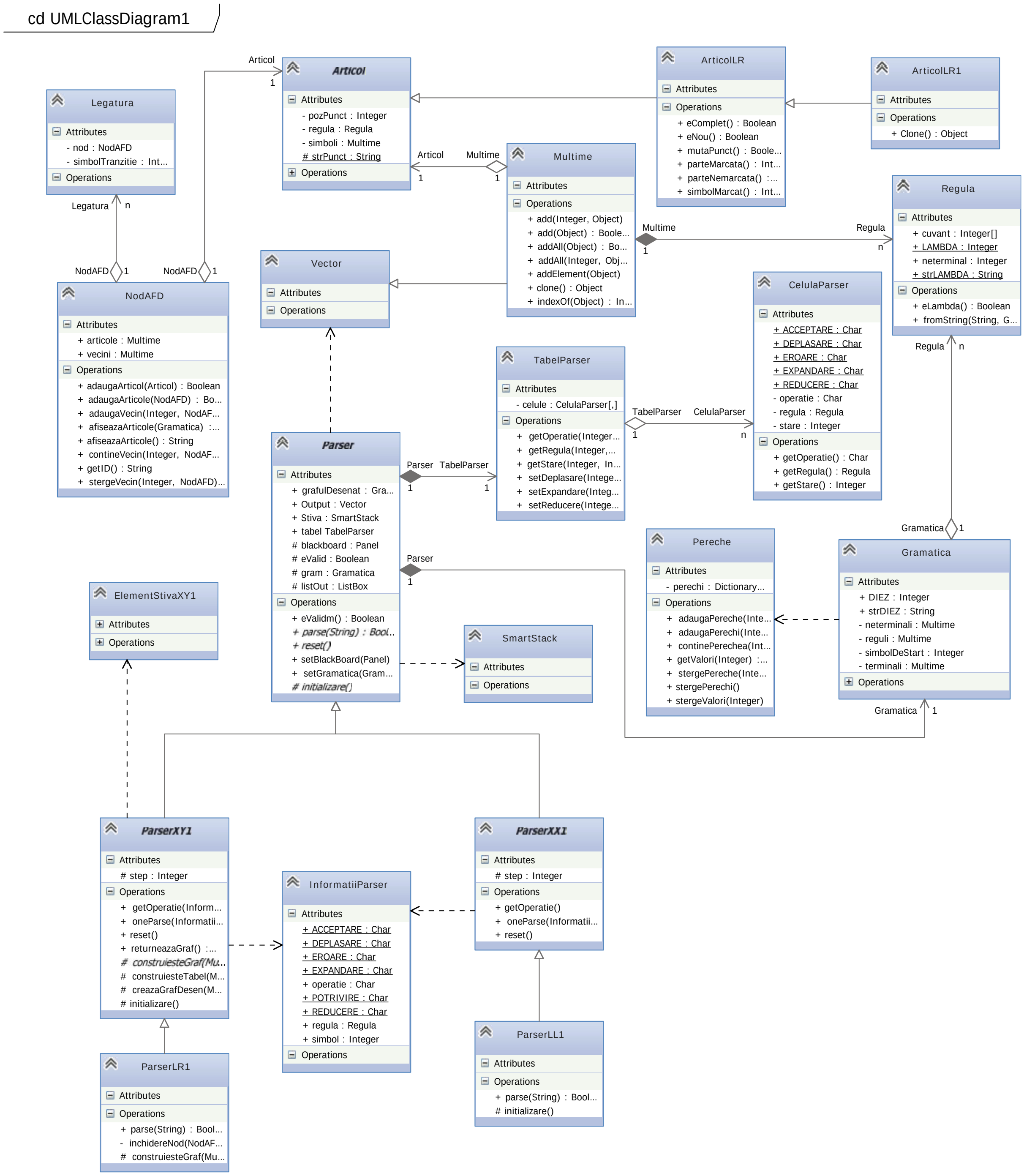
Dezavantajul principal al metodei este acela că determinarea tabelei de analiză necesită un volum mare de muncă; există însă generatoare de analizoare de tip LR, precum yacc sau bison, care produc un astfel de analizor.

Vom considera în continuare o gramatică G = (V, T, S, P) redusă şi gramatica augmentată G' = (V' , T' , S' , P') unde P' = P$\cup${S'$\rightarrow$ S }, V' = V $\cup${ 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.

3.3.1 O caracterizare a gramaticilor LR(1)

Capitolul 4

Aplicatia mea



Capitolul 5

Concluzii

Listă imagini