TEHNICI DE COMPILARE

CURS 3

Gianina Georgescu

CUPRINSUL CURSULUI 3

ANALIZA SINTACTICĂ

- Sintaxa unui limbaj de programare
- Formalizarea sintaxei cu ajutorul gramaticilor independente de context
- Notația Backus-Naur
- Translatoare push-down: definiție, exemple
- Metode generale de analiză sintactică: top-down și bottom-up

ANALIZA SINTACTICĂ

Sintaxa unui limbaj de programare cuprinde mulțimea regulilor referitoare la:

- Structura unui program scris în limbajul respectiv
- Diferite tipuri de declarații (variabile, funcții/ proceduri, structuri etc.)
- Instrucțiuni
- Expresii

Sintaxa unui limbaj de programare poate fi formalizată de o gramatică independentă de context (prescurtat gic)

ANALIZA SINTACTICĂ – GRAMATICI INDEPENDENTE DE CONTEXT

Definiție. O gramatică independentă de context are o structură de forma:

$$G=(N,\Sigma,S,P), \text{ unde:}$$

- N este alfabetul neterminalilor
- Σ este alfabetul terminalilor
- $S \in N$ este simbolul de start
- P este mulțimea producțiilor de forma:

$$A \rightarrow x,\ A \in N,\ x \in (N \cup \Sigma)^*$$

GRAMATICI INDEPENDENTE DE CONTEXT

O **derivare** (într-un pas) în G, notată prin $x \Rightarrow_G y$ (sau prin $x \Rightarrow y$ când G este subînțeles), unde $x = zAu, y = zwu, A \to w \in P$, $z, u, w \in (N \cup \Sigma)^*$ (A este înlocuit de w).

Închiderea reflexivă și tranzitivă a relației $\Rightarrow$ se notează cu $\overset{*}{\Rightarrow}$ (reprezintă o derivare în zero sau mai mulți pași).

Închiderea tranzitivă a relației $\Rightarrow$ se notează cu $\overset{+}{\Rightarrow}$ (reprezintă o derivare în cel puțin un pas).

Dacă în derivarea $xAy \Rightarrow xzy$ (într-un pas), $A \to z \in P$ avem $x \in \Sigma^*$ (respectiv $y \in \Sigma^*$) spunem că este vorba despre o **derivare stângă** (respectiv **dreaptă**), notată $\underset{s}{\Rightarrow}$ (respectiv $\underset{d}{\Rightarrow}$).

Limbajul generat de gramatica $G = (N, \Sigma, S, P)$ este:

$$L(G) = \{w \in \Sigma^* | S \overset{*}{\Rightarrow} \mathrm{w}\}$$

GRAMATICI INDEPENDENTE DE CONTEXT

Un **arbore de derivare** în G este construit astfel:

- rădăcina este etichetată cu S
- dacă $A \in N$ este eticheta unui nod interior, atunci $Z_1, \dots, Z_m \in \Sigma \cup N$ sunt descendenții săi direcți, de la stânga la dreapta, dacă și numai dacă $A \to Z_1 \dots Z_m$ este o producție din P. Dacă $A \to \lambda$ este în P, atunci A va putea avea ca (unic) descendent pe λ.
- Unei derivări stângi sau drepte îi corespunde un unic arbore de derivare.
- Unui arbore de derivare îi pot corespunde mai multe derivări (nu neapărat stângi sau drepte)

GRAMATICI INDEPENDENTE DE CONTEXT

Definitie. Spunem că G este gramatică **ambiguă** dacă este adevărată cel puțin una dintre afirmațiile (echivalente):

1) $\exists w \in L(G), w$ are cel puțin două derivări stângi
2) $\exists w \in L(G), w$ are cel puțin două derivări drepte
3) $\exists w \in L(G), w$ are cel puțin doi arbori de derivare distincți

Observații. Pentru gramaticile neambigue se pot implementa algoritmi de analiză sintactică liniari.

La modul general, problema dacă o gic este ambiguă este nedecidabilă.

Exemplu. Gramatica cu producțiile $E \to E + E \mid n$ este ambiguă

EXEMPLU DE GRAMATICĂ INDEPENDENTĂ DE CONTEXT CE REDĂ SINTAXA PENTRU EXPRESII

Cu notația Backus-Naur

$$< expr >::=< term > " + " < expr > \mid < term >$$
$$< term >::=< factor > " * " < term > \mid < factor >$$
$$< factor >::=< const > \mid < var > \mid "(" < expr > ")"$$
$$< const >::=< digit > \mid < digit >< const >$$
$$< digit >::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"$$

EXEMPLU DE GRAMATICĂ INDEPENDENTĂ DE CONTEXT CE REDĂ SINTAXA PENTRU EXPRESII

Cu notația folosită de *bison*

```
input:   /* empty string */
       | input line
;
line:    `\n`
       | exp `\n`
;
exp:     NUM
       | exp`+`exp
       | exp`-`exp
       | exp`*`exp
       | exp`/`exp
       | `(`exp`)`
;
```

(*input*, *line*, *exp* sunt neterminali, iar NUM este token declarat într-o secțiune specială)

ELIMINAREA AMBIGUITĂȚII PENTRU ANUMITE GRAMATICI INDEPENDENTE DE CONTEXT

1. Gramatica ambiguă G_1 generează expresii formate cu ajutorul operatorului *op* și are producțiile:

$$Exp \rightarrow Exp\ op\ Exp \mid n$$

Dacă operatorul *op* este asociativ la stânga, atunci o expresie generată de G_1 va fi procesată corect într-un parser de tip *LR* (care are asociată o gramatică neambiguă) dacă înlocuim producțiile de mai sus cu producțiile (care definesc o gramatică neambiguă echivalentă cu G_1):

$$Exp \rightarrow Exp\ op\ Exp' \mid Exp'$$
$$Exp' \rightarrow n$$

Dacă *op* este asociativ la dreapta, atunci înlocuim producțiile lui G_1 cu producțiile (care definesc o gramtică neambiguă echivalentă cu G_1):

$$Exp \rightarrow Exp'\ op\ Exp \mid Exp'$$
$$Exp' \rightarrow n$$

ELIMINAREA AMBIGUITĂȚII PENTRU ANUMITE GRAMATICI INDEPENDENTE DE CONTEXT

2. Fie G_2 gramatica (ambiguă) care generează expresii formate cu ajutorul operatorilor + și *, cu producțiile:

$$Exp \rightarrow Exp + Exp \mid Exp * Exp \mid n$$

Următoarea gramatică este neambiguă, este echivalentă cu G_2, exprimă asociativitatea la stânga a operatorilor '+' și '*', precum și precedența mai mare a lui '*' față de '+' atunci când este folosită de un parser LR:

$$Exp \rightarrow Exp + Exp' \mid Exp'$$
$$Exp' \rightarrow Exp' * Exp'' \mid Exp''$$
$$Exp'' \rightarrow n$$

ANALIZA SINTACTICĂ

La modul formal, pentru o gic $G = (N, \Sigma, S, P)$ și $w \in \Sigma^*$, a analiza sintactic pe w înseamnă a decide algoritmic dacă $w \in L(G)$. În caz afirmativ se furnizează o derivare, de regulă stângă sau dreaptă, a lui w.

În cazul unui limbaj de programare $\mathcal{L}$ pentru care sintaxa este formalizată de gic $G = (N, \Sigma, S, P)$:

- N reprezintă diferitele categorii sintactice (declarații, instrucțiuni, expresii, constante etc.)
- Σ conține toate tipurile de token-i
- P conține toate regulile sintactice (regulile de formare a instrucțiunilor, declarațiilor etc.), scrise în format Backus-Naur, în format *bison* etc.
- Pentru a verifica dacă $w \in \Sigma^*$ (w corect din punct de vedere lexical) este în $L(G)$ se folosește un automat push-down care corespunde lui G. Dacă $w \in L(G)$, spunem că w este **corect din punct de vedere sintactic**.
- În cazul în care dorim să obținem și o derivare a lui w se utilizează un **translator stivă** care corespunde lui G.

ANALIZA SINTACTICĂ – PARSER, AUTOMAT PUSH-DOWN

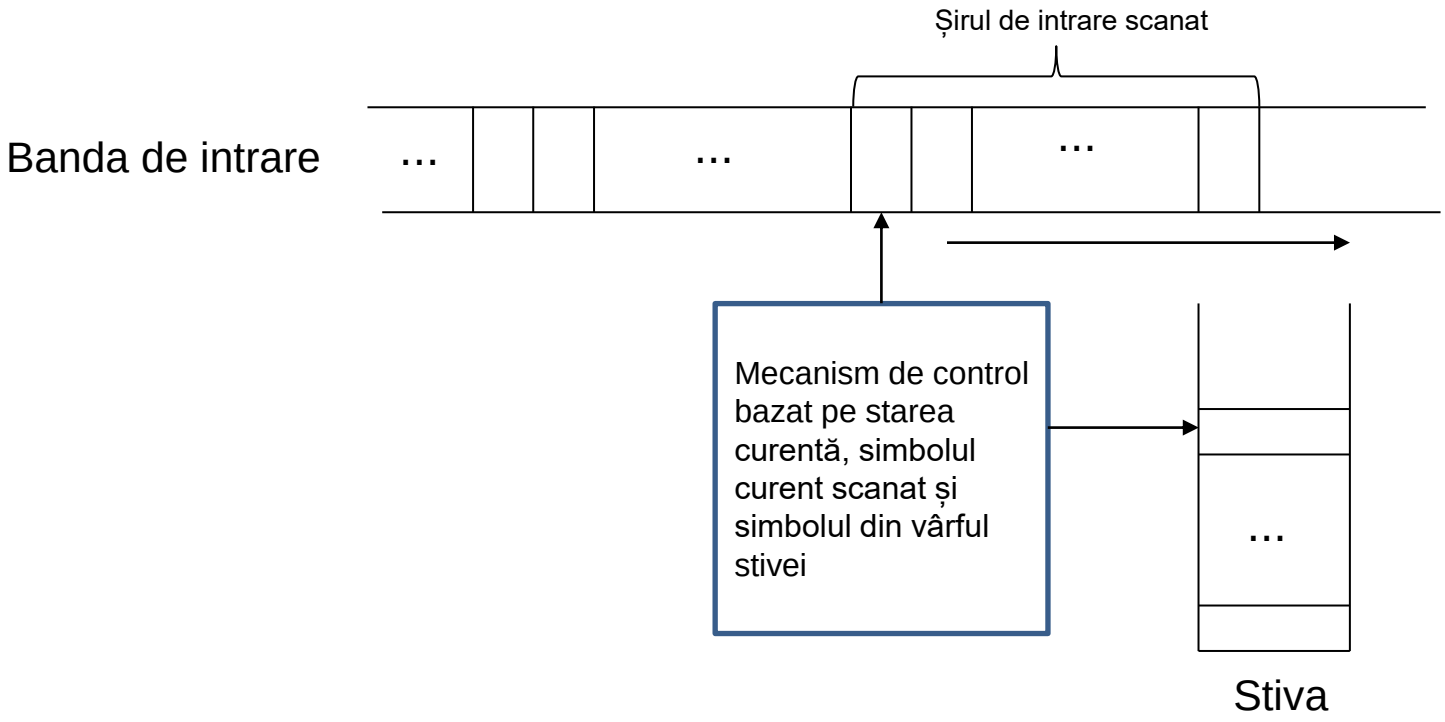
Analizorul sintactic (parserul) este de fapt implementarea unui **automat push-down.**

Definiție. Un **automat push down** are o structură de forma:

$$A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F), \text{ unde:}$$

- Q este mulțimea stărilor
- Σ este alfabetul automatului
- Γ este alfabetul stivei
- $\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to \mathcal{P}_{fin}(Q \times \Gamma^*)$
- q_0 este starea inițială a automatului
- $Z_0 \in \Gamma$ simbolul inițial al stivei
- $F \subseteq Q$ mulțimea stărilor finale

ANALIZA SINTACTICĂ – PARSER, AUTOMATUL PUSH-DOWN



ANALIZA SINTACTICĂ –AUTOMATUL PUSH-DOWN

- **Descriere instantanee** (instanță) a lui A

  $(p, x, a), p \in Q$ starea curentă a lui A

  $x \in \Sigma^*$ șirul curent scanat din intrare

  $\alpha \in \Gamma^*$ conținutul stivei

- **Mișcare a lui A:**

  $(p, ax, Z\alpha) \vdash (q, x, \beta\alpha)$ ddacă $(q, \beta) \in \delta(p, a, Z)$

  pentru $p, q \in Q, a \in \Sigma \cup \{\lambda\}, x \in \Sigma^*, \mathrm{Z} \in \Gamma, \alpha, \beta \in \Gamma^*$

- **Închiderea reflexivă și tranzitivă** a relației $\vdash$ este notată cu $\vdash^*$

- **Limbajul acceptat de A cu stări finale:**

$$L(A) = \{w \in \Sigma^* | (q_0, w, Z_0) \vdash^* (\mathrm{q}, \lambda, \alpha), q \in F, \alpha \in \Gamma^*\}$$

- **Limbajul acceptat de A cu vidarea stivei:**

$$L_\lambda(A) = \{w \in \Sigma^* | (q_0, w, Z_0) \vdash^* (\mathrm{q}, \lambda, \lambda), q \in Q\}$$

ANALIZA SINTACTICĂ – AUTOMATUL PUSH-DOWN

- **Propoziție:** Pentru un automat push-down cu stări finale există un automat push-down cu vidarea stivei echivalent și reciproc.

- **Automat push-down determinist:**

  Spunem că $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ este **determinist** dacă:

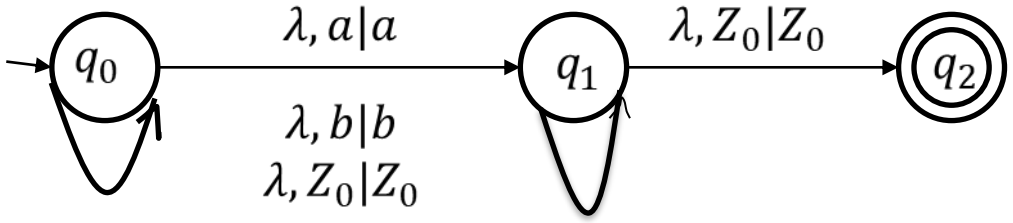
  $$|\delta(p, a, Z)| + |\delta(p, \lambda, Z)| \leq 1, \forall p \in Q, \forall a \in \Sigma, \forall Z \in \Gamma$$

- În cazul limbajelor de programare, este important ca parserul să fie construit pe baza unui automat determinist ce corespunde gramaticii (neambigue) a sintaxei limbajului.
- Algoritmii bazați pe automate deterministe sunt liniari

ANALIZA SINTACTICĂ – AUTOMATUL PUSH-DOWN

Exemplu: Automatul nedeterminist

$A = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, Z_0\}, \delta, q_0, Z_0, \{q_2\})$ cu stări finale



$a, Z_0|aZ_0$
$b, Z_0|bZ_0$
$a, a|aa$
$a, b|ab$
$b, a|ba$
$b, b|bb$

$a, a|\lambda$
$b, b|\lambda$

recunoaște limbajul $L(A) = \{ww^R | w \in \{a, b\}^*\}$

$(q_0, abba, Z_0) \vdash (q_0, bba, aZ_0) \vdash (q_0, ba, baZ_0) \vdash (q_1, ba, baZ_0) \vdash$
$(q_1, a, aZ_0) \vdash (q_1, \lambda, Z_0) \vdash (q_2, \lambda, Z_0) \vdash$ accept

ANALIZA SINTACTICĂ – TRANSLATORUL STIVĂ

Ca și în cazul translatoarelor finite, translatoarele stivă pot produce ieșiri pentru fiecare tranziție.

Definiție. Un **translator stivă** are o structură de forma:

$$T = (Q, V_i, V_e, \Gamma, \delta, q_0, Z_0, F), \text{unde:}$$

- Q este mulțimea stărilor
- V_i este alfabetul de intrare
- V_e este alfabetul de ieșire
- Γ este alfabetul stivei
- $\delta: Q \times (V_i \cup \{\lambda\}) \times \Gamma) \rightarrow \mathcal{P}_{fin}(Q \times \Gamma^* \times V_e^*)$
- $q_0 \in Q$ starea inițială
- $Z_0 \in \Gamma$ simbolul inițial al stivei
- $F \subseteq Q$ mulțimea stărilor finale (F poate fi mulțimea vidă)

ANALIZA SINTACTICĂ –TRANSLATORUL STIVĂ

- **Descriere instantanee** (instanță) a lui T

  $(p, x, \alpha, y), p \in Q$ starea curentă a lui T

  $x \in V_i^*$ șirul curent scanat din intrare

  $\alpha \in \Gamma^*$ conținutul stivei

  $y \in V_e^*$ șirul curent din ieșire

- **Mișcare a lui T:**

  $(p, ax, Z\alpha, y) \vdash (q, x, \beta\alpha, yy')$ ddacă $(q, \beta, y') \in \delta(p, a, Z)$

  pentru $p, q \in Q, a \in \Sigma \cup \{\lambda\}, x \in V_i^*, Z \in \Gamma, \alpha, \beta \in \Gamma^*,$

  $y, y' \in V_e^*$

- **Închiderea reflexivă și tranzitivă** a relației $\vdash$ este notată cu $\vdash^*$

ANALIZA SINTACTICĂ –TRANSLATORUL STIVĂ

Translatarea definită de T

Cu stări finale

- pentru un șir de intrare $x \in V_i^*$:
  $T(x) = \{y \in V_e^* | \ (q_0, x, Z_0, \lambda) \vdash^* (q, \lambda, \alpha, y), q \in F, \alpha \in \Gamma^*\}$
- pentru un limbaj $L \subseteq V_i^*$
  $T(L) = \bigcup_{x \in L} T(x)$
- translatarea definită de T cu stări finale (global)
  $\tau(T) = \{(x, y) | x \in V_i^*, y \in V_e^*, y \in T(x)\}$

Cu vidarea stivei

- pentru un șir de intrare $x \in V_i^*$:
  $T_\lambda(x) = \{y \in V_i^* | \ (q_0, x, Z_0, \lambda) \vdash^* (q, \lambda, \lambda, y), q \in Q\}$
- pentru un limbaj $L \subseteq V_i^*$
  $T_\lambda(L) = \bigcup_{x \in L} T_\lambda(x)$
- translatarea definită de T cu vidarea stivei (global)
  $\tau_\lambda(T) = \{(x, y) | x \in V_i^*, y \in V_e^*, y \in T_\lambda(x)\}$

ANALIZA SINTACTICĂ –TRANSLATORUL STIVĂ

Exemplu:

Fie gramatica ($G = (N, \Sigma, S, P)$ independentă de context, cu producțiile numerotate $1, \cdots, |P|$.

Translatorul următor:

$$T_G = (\{q\}, \Sigma, \{1, \dots, |P|\}, N \cup \Sigma, \delta, q, S, \emptyset),$$

unde $\delta(q, \lambda, A) = \{(q, \alpha, i) | i: A \to \alpha \in P\}$

$$\delta(q, a, a) = \{(q, \lambda, \lambda)\} \, \forall a \in \Sigma$$

are ca translatare

$$\tau_\lambda(T_G) = \{(w, \pi) | S \overset{\pi}{\Rightarrow}_s w, w \in L(G) \subseteq \Sigma^*, \pi \in \{1, \dots, |P|\}^*\}$$

Observații.

— Translatorul de mai sus simulează (într-un mod ”foarte nedeterminist”) derivările stângi din G

— i reprezintă numărul de ordine al producției aplicate în ordinea dată, iar π succesiunea de producții aplicate pentru derivarea lui w din simbolul de start S într-o derivare stângă

METODE DE ANALIZĂ SINTACTICĂ

A. METODA TOP-DOWN

Fie $(G = (N, \Sigma, S, P)$ gic și $w \in \Sigma^*$. În cazul metodei top-down

- Se pornește de la simbolul de start S
- În forma sentențială curentă (inițal S) se alege un neterminal A și o producție $A \to \alpha$, se înlocuiește A cu α (alegerea producției se face conform unor criterii prestabilite sau pur și simplu în ordinea în care sunt listate producțiile gramaticii)
- Dacă nu mai există nicio alternativă pentru A(nicio producție $A \to \alpha$ care să nu fi fost analizată), atunci se revine la un pas anterior
- Dacă la final nici pentru S nu mai există alternative, atunci $w \notin L(G)$
- Altfel, dacă forma sentențială curentă este egală cu w, atunci $w \in L(G)$ și se furnizează o derivare stângă a sa
- Există gramatici pentru care metoda descrisă mai sus nu funcționează
- Denumirea top-down provine de la faptul că w este obținut ca și cum s-ar construi arborele lui de derivare de sus în jos, dacă $w \in L(G)$

METODE DE ANALIZĂ SINTACTICĂ

B. METODA BOTTOM-UP

Fie $(G = (N, \Sigma, S, P)$ gic și $w \in \Sigma^*$. În cazul metodei bottom-up:

- Se pornește de la șirul analizat, w
- În forma sentențială curentă $\alpha \in N \cup \Sigma)^*$(inițal $\alpha = w$) se identifică un subșir β, astfel ca $\alpha = \alpha'\beta\alpha'', \beta \in (N \cup \Sigma)^*,$ pentru care există $A \to \beta \in P$; se înlocuiește β cu A (operație care se numește **reducere**, inversă derivării), obținându-se șirul curent $\alpha' A \alpha''$ (identificarea lui β se poate face în conformitate cu anumite criterii, sau este considerat arbitrar)
- Se repetă secvența de mai sus până când:
  - se ajunge la forma sentențială S, caz în care $w \in L(G)$, STOP
  - nu mai există în forma sentențială curentă niciun subșir care să poată fi redus (înlocuit de un neterminal); în acest caz se încearcă revenirea la un pas anterior
  - dacă nu mai există alternative, atunci $w \notin L(G)$, STOP
- În cazul $w \in L(G)$, se furnizează o derivare dreaptă a sa
- Există gramatici pentru care metoda generală descrisă mai sus nu funcționează
- Denumirea **bottom-up** provine de la faptul că dacă $w \in L(G)$, prin aplicarea acestei metode este ca și cum arborele sintactic pentru w s-ar parcurge de jos în sus

METODE DE ANALIZĂ SINTACTICĂ

Pentru cele două metode descrise mai sus la modul general, există mai multe variante de algoritmi:

- exponențiali; nu se folosesc în practică
- polinomiali (cum este algoritmul CYK, de ordin $O(n^3)$)
- liniari; aceștia sunt cei mai utilizați de compilatoare
  - metode liniare de tip top-down: algoritmi de tip LL
  - metode liniare de tip bottom-up: algoritmi de tip LR

METODA DE ANALIZĂ SINTACTICĂ TOP-DOWN

Fie $G=(N, \Sigma, S, P)$ o gramatică independentă de context și $w \in \Sigma^*$ un cuvânt peste Σ. Pentru a verifica dacă $w \in L(G)$ cu metoda top-down se pornește de la simbolul de start, S, și se generează toți arborii de derivare până când (eventual) găsim unul care corespunde lui w. Dacă $w \in L(G)$ atunci se va genera o derivare stângă a sa.
Există mai multe abordări în rezolvarea acestei probleme.

1) Algoritm Breadth-First. Este cel mai general algoritm care poate fi utilizat pentru orice tip de gramatică independentă de context. Se folosește o coadă în care se introduc formele sentențiale curente. Inițial, coada conține simbolul de start, S. Algoritmul este foarte lent. În cazurile cele mai defavorabile atât timpul, cât și spațiul necesitate pot fi exponențiale, în funcție de lungimea șirului analizat.

2) Algoritm Breadth-First cu derivare stângă. În acest caz se reduce din efortul de calcul, însă algoritmul nu funcționează pentru gramatici independente de context recursive la stânga (în care există derivări de forma $A \overset{+}{\Rightarrow} A\alpha$), deoarece se poate intra în cicluri infinite.

INPUT: gramatica independentă de context $G = (N, \Sigma, S, P)$, nerecursivă la stânga,
$w \in \Sigma^*, n = |w|$

ALGORITM

Top_Down_Breadth_First_Left() {
$S \Rightarrow Q$ // initializare; Q este o coadă
while (Q este nevidă)
{ $z \Leftarrow Q$;
Fie $z = xAu, x \in \Sigma^*, A \in N, u \in (N \cup \Sigma)^*$;
if (x este sufix al lui w) //daca $x = \lambda$ atunci x este considerat sufix al lui w
{// alege o productie pentru A
for (fiecare productie $A \longrightarrow v$ din P)
{
if ($xvu == w$) accept();
// sirurile terminale $xvu \neq w$ nu sunt introduse in coada
if ($|xvu|_N \geq 1$ && $|xvu|_\Sigma \leq n$) $xvu \Rightarrow Q$;
} // end_for
} //end if
}//end_while
reject(); // coada este vida
} // end_ Top_Down_Breadth_First_Left

3) Algoritm Deapth-First pentru derivări stângi. Algoritmul folosește backtracking și inițial pune pe stivă simbolul $. getNextToken() este o funcție care furnizează următorul token. În cazul în care lucrăm cu o gramatică cu simboluri terminale abstracte și facem analiza pentru un șir w de lungime n, atunci această funcție ne furnizează următorul simbol al lui w (adică este o banală incrementare a unui indice; când acest indice devine n+1, atunci s-au citit toate simbolurile lui w).

Algoritmul de mai jos se termină întotdeauna atunci când G nu este recursivă la stânga. Algoritmul se încheie atunci când în interiorul lui while se apelează funcția reject(), caz în care $w \notin L(G)$, sau atunci cânde se apelează funcția accept(), caz în care $w \in L(G)$.

INPUT: gramatica independentă de context $G=(N, \Sigma, S, P)$ și $w=z\#$, unde $z \in \Sigma^*$, $\#=EOF$ (dacă G formalizează sintaxa unui limbaj de programare, atunci Σ reprezintă mulțimea categoriilor de atomi lexicali (token-ii) acceptați de limbajul respectiv, iar EOF marchează sfârșitul fișerului care conține programul sursă, program reprezentat de z)

ALGORITM TOP-DOWN CU DEPTH-FIRST (BACKTRACKING)

```
// initializare
push($);
X← S;
a ← getNextToken();
void Top_Down_Depth_First_Left(){
while(true)
 {if (X este neterminal)
    {
      if ( mai exista productie X→ Y_1 ⋯ Y_k din P neanalizata)
        {
         push Y_k; ⋯ ;  push Y_1;
         X ← pop();
        }
      else if ( X == S ) reject();  // am analizat toate alternativele pentru S
    }
  else   // X este terminal sau $
      if (X== $ && a == EOF) accept();
      else
          if (X== a)
             { a  ← getNextToken();
               X ← pop();
             }
           else  //daca token-ul curent este diferit de simbolul din varful stivei
                 // revenim la pasul anterior
                Top_Down_Depth_First_Left();
 } // end_while
} // end_ Top_Down_Depth_First_Left
```