Limbaje Formale și Compilatoare (LFC)

- Curs -

Ș.l.dr.ing Octavian MACHIDON

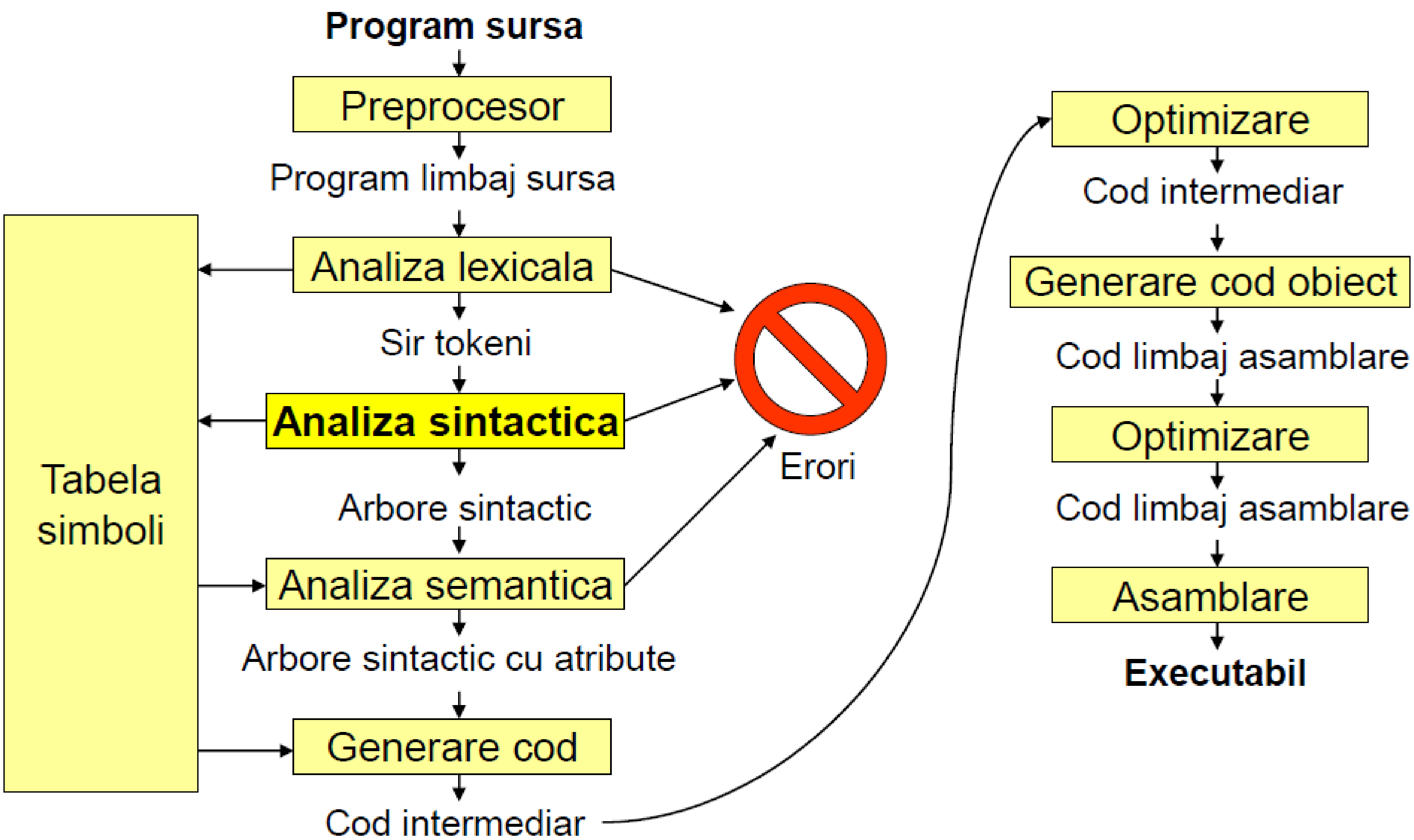
octavian.machidon@unitbv.ro

Universitatea Transilvania din Brașov

Astăzi

- Analiza sintactică
  - Introducere în analiza sintactică
  - Analiza sintactică LL
  - Recursivitate stângă
  - Parser recursiv
  - Factorizare stânga
  - Eliminarea ambiguităților

Structura detaliată

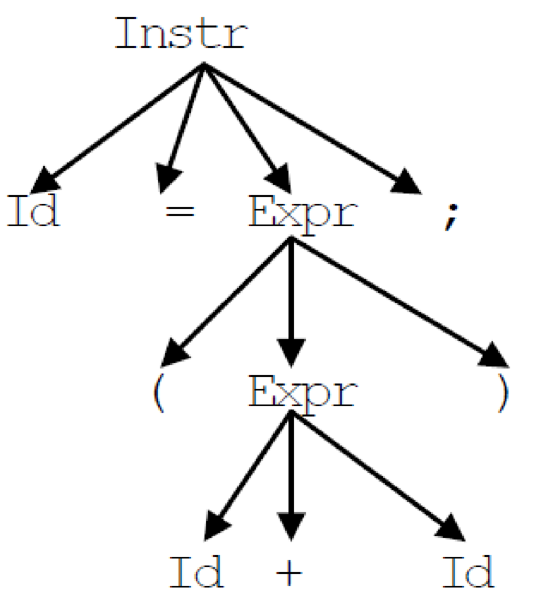


Analiza sintactică

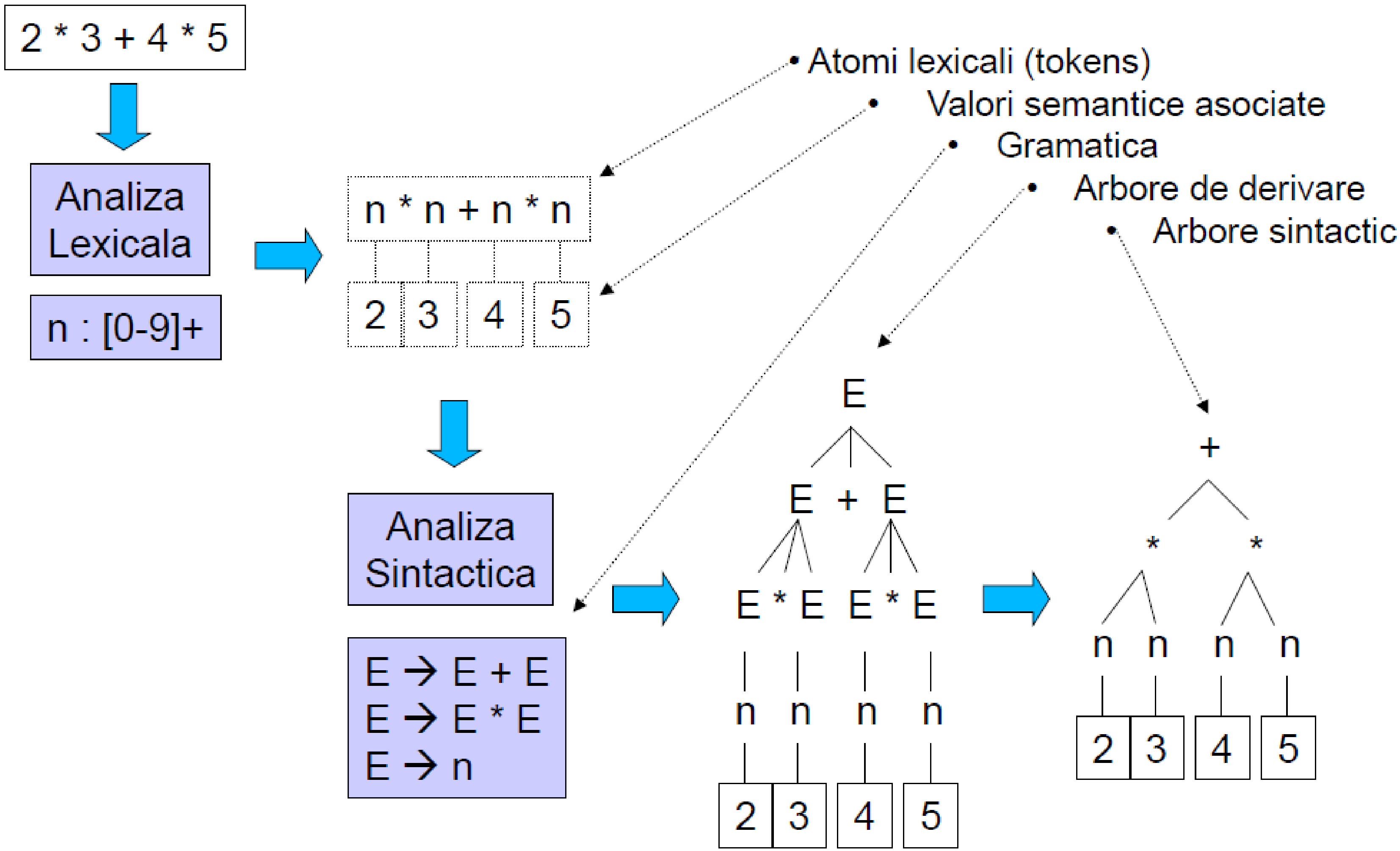
- Verifica formarea corecta (cf. gramaticii) a constructiilor din limbaj
  - Analiza lexicală – “cuvinte”
  - Analiza sintactica – “propozitii”
- Primeste un sir de atomi lexicali, construieste un arbore de derivare
  - Structura utila in final este un arbore sintactic
- Folosita in front-end-ul unui interpretor / compilator
  - Dar si de catre IDE: syntax highlight, navigare prin cod, refactoring

Exemplu

- ALFA = (BETA + GAMA);
- id = (id + id) ;
- Instr $\rightarrow$ id = Expr ;
  Expr $\rightarrow$ Expr + Expr
  | Expr * Expr
  | (Expr)
  | id



Arbore de derivare / sintactic



Arborele de derivare (parsare)

- Fie gramatica:

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$

- Arborele de derivare pentru

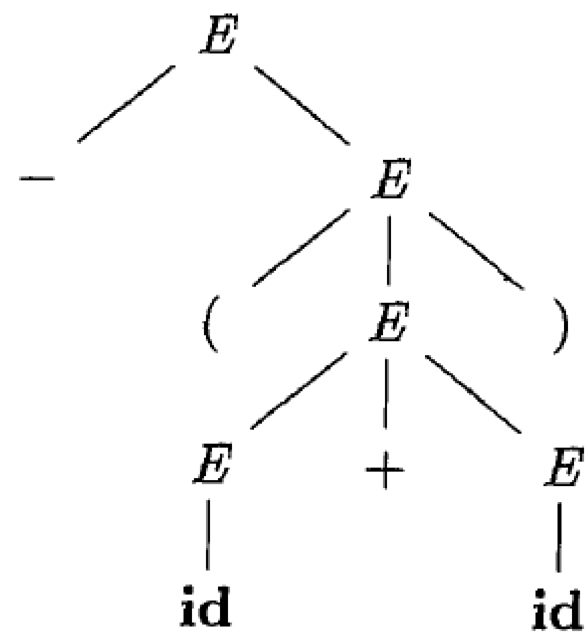
  - (id + id)

  corespunde celor 2 derivari posibile:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

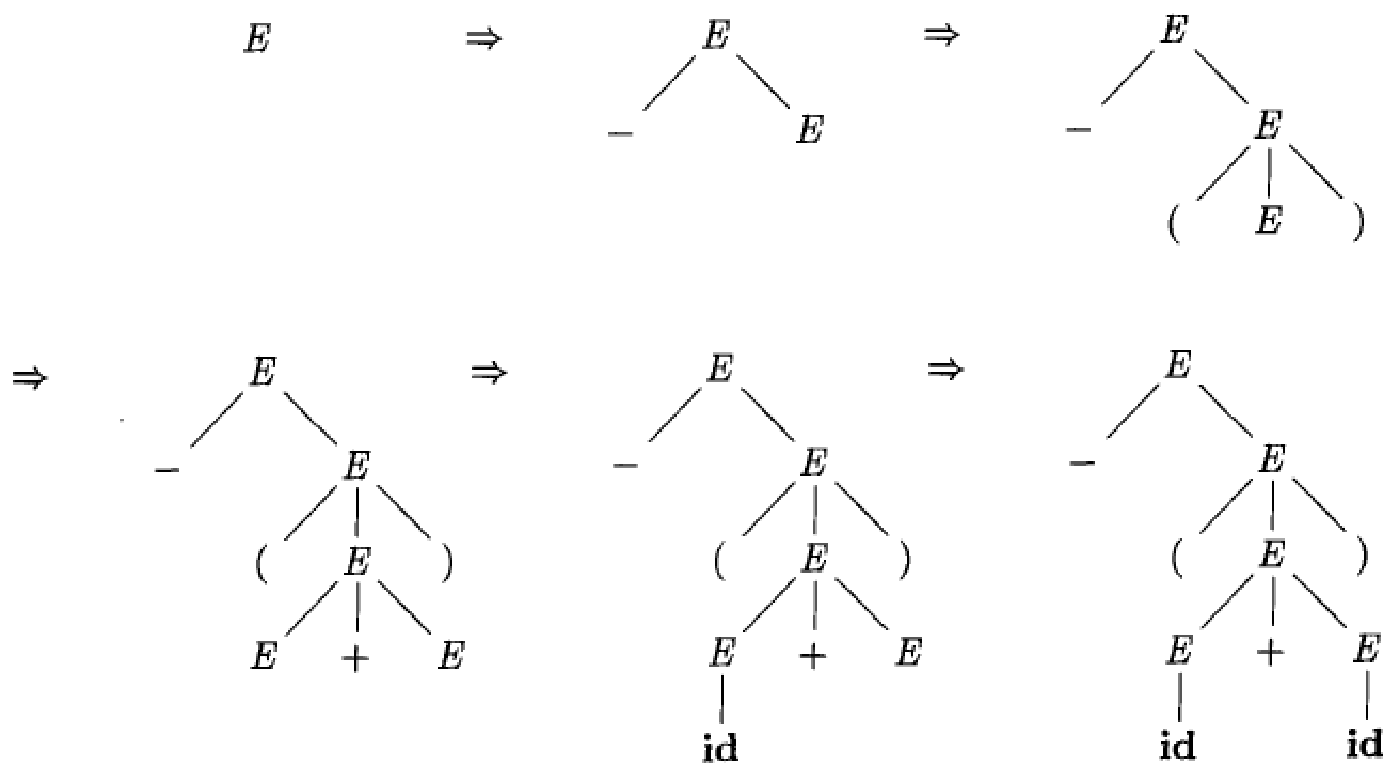
  sau

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$$



Construirea arborelui de derivare

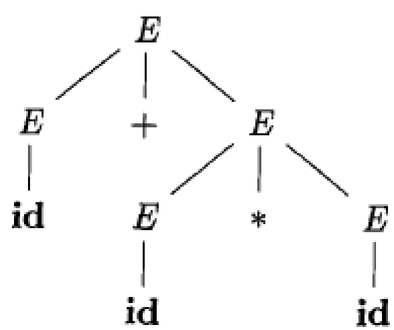
- Gramatica: $E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$
- Expresia: **- (id + id)**



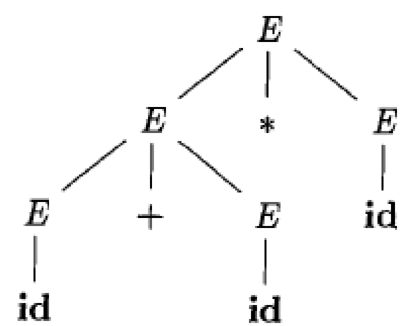
Ambiguități

- Gramatica: $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$
- Permite două derivări distincte pentru expresia: $\mathbf{id} + \mathbf{id} * \mathbf{id}$

$$
\begin{array}{lll}
E & \Rightarrow & E + E \\
  & \Rightarrow & \mathbf{id} + E \\
  & \Rightarrow & \mathbf{id} + E * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{lll}
E & \Rightarrow & E * E \\
  & \Rightarrow & E + E * E \\
  & \Rightarrow & \mathbf{id} + E * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
  & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
$$



(a)



(b)

Tipuri de analiză sintactică

- Descendenta (top-down, de sus in jos)
  - Inlocuieste cate un neterminal cu partea dreapta a unei productii, pana ramane doar cu terminali
- Ascendenta (bottom-up, de jos in sus)
  - Porneste de la sirul de atomi lexicali, abstractizeaza din sir simbolul de start prin reduceri succesive
- Analiza descendenta – derivare stanga
  - Tot timpul inlocuim cel mai din stanga neterminal
  - LL (**L**eft to right, **L**eftmost derivation)
- Analiza ascendenta - derivarea dreapta
  - primul neterminal înlocuit este cel mai din dreapta din forma propozi ională curentă
  - LR (**L**eft to right, **R**ightmost derivation)

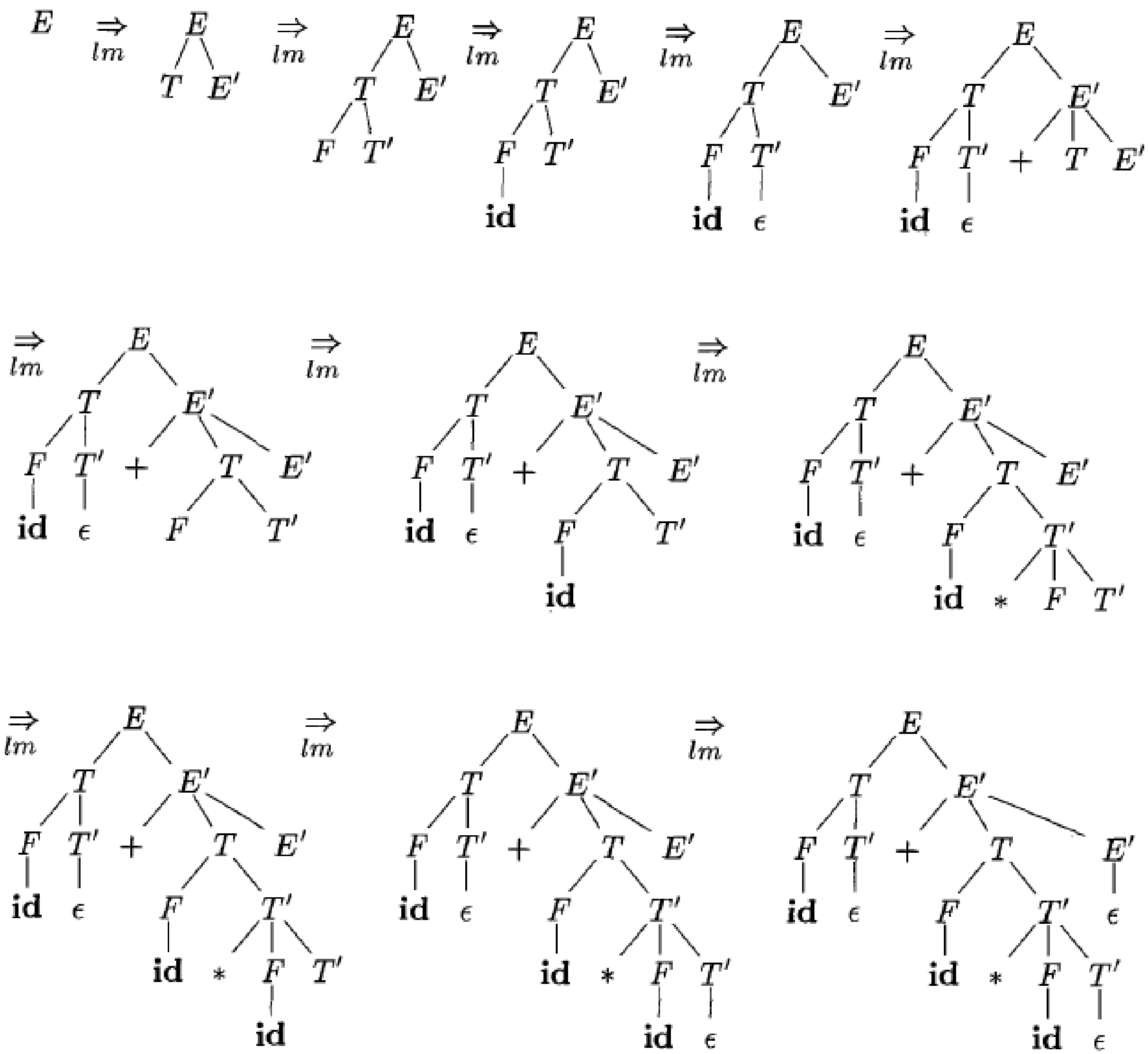
Top-down parsing

- Construirea arborelui de derivare pornind de la expresia de intrare, începând cu nodul rădăcină și creând nodurile în pre-ordine (în adâncime).
- Cu alte cuvinte, parsarea descendentă este echivalentă cu găsirea celei mai la stânga derivări (leftmost derivation) pentru o expresie de intrare

Top-down parsing

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

$$\mathbf{id} + \mathbf{id} * \mathbf{id}$$

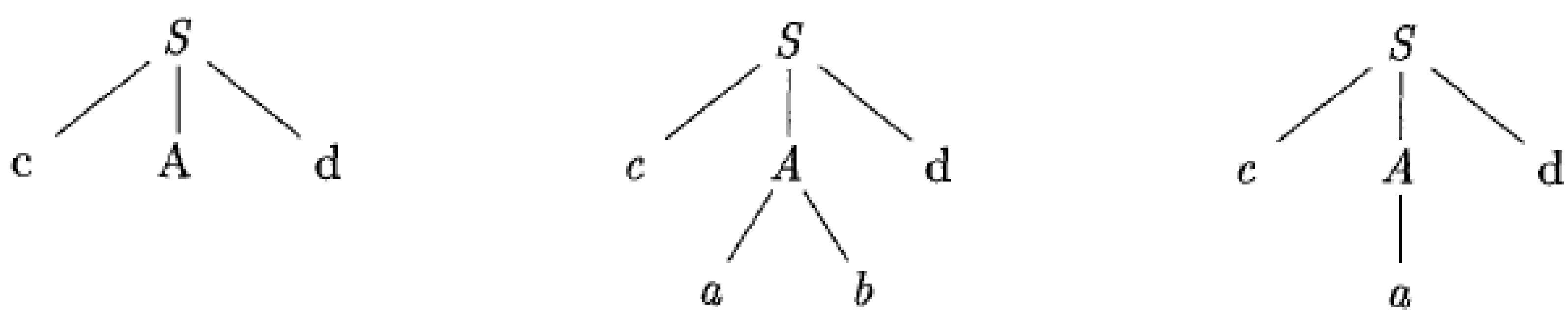


Algoritmul de parsare descendent recursiv

```
void A() {
    Choose an A-production, A → X1X2 ··· Xk;
    for ( i = 1 to k ) {
        if ( Xi is a nonterminal )
            call procedure Xi();
        else if ( Xi equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

Expresia de intrare: cad

Etapele parsării top-down:



Derivare stânga, top down (LL)

Instr	id = (id + id) ;
id = **Expr** ;	id = (id + id) ;
id = (**Expr**) ;	id = (id + id) ;
id = (**Expr** + Expr) ;	id = (id + id) ;
id = (id + **Expr**) ;	id = (id + id) ;
id = (id + id) ;	id = (id + id) ;

- LL: Șirul de tokeni se parcurge din stânga (L)
- Se deriveaza non-terminalul cel mai din stânga (L)
- Cum alegem producția folosită pentru derivare?
- Backtracking dacă alegem producția greșită

Derivare stânga, top down (LL)

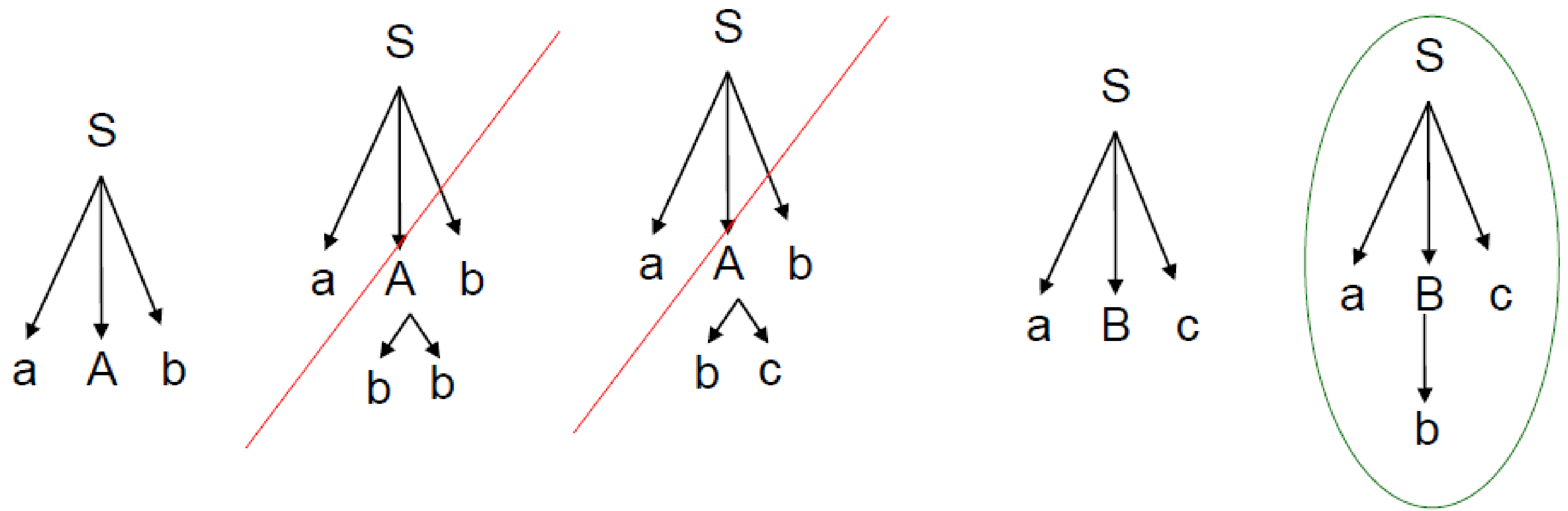
Instr	id = (id) + (id) ;
id = **Expr** ;	id = (id) + (id) ;
id = **Expr** + Expr ;	id = (id) + (id) ;
id = (**Expr**) + Expr ;	id = (id) + (id) ;
id = (id) + **Expr** ;	id = (id) + (id) ;
id = (id) + (**Expr**) ;	id = (id) + (id) ;
id = (id + id) ;	id = (id) + (id) ;

- Exemplu similar, dar trebuie alese alte producții pentru o derivare corectă fără backtracking
- Este necesară o metodă de predicție

Top down cu backtracking

- $S \rightarrow$ a A b | a B c
- $A \rightarrow$ bb | bc
- $B \rightarrow$ b

Gramatica generează de fapt (abbb, abcb şi abc).
Arborele de derivare pentru şirul abc (abordare descendenta):



Derivare dreapta, bottom up (LR)

	id = (id + id) ;
id	= (id + id) ;
id =	(id + id) ;
id = (	id + id) ;
id = (id	+ id) ;
id = (**Expr**	+ id) ;
id = (Expr +	id) ;
id = (Expr + id	) ;
id = (Expr + **Expr**	) ;
id = (Expr + Expr)	;
id = (**Expr**)	;
id = **Expr**	;
id = Expr ;	
Instr	

- LR: șirul de tokeni se parcurge din stânga (L)
- Tokenii sunt adăugați pe o stivă
- Se compara partea dreaptă a stivei (R) cu partea dreaptă a unei producții
- De ce primul **id** nu a fost transformat în **Expr**?
- Decizie : SHIFT sau REDUCE

Analiza LL, LR

- Vrem sa evitam backtrackingul
- O clasă de gramatici independente de context care permit o analiza deterministă.
  - Alg. LL(k) analizeaza left-to-right, derivare stanga
  - Alg. LR(k) analizeaza left-to-right, derivare dreapta
  - K –lookahead (cati tokeni sunt cititi)
- LL(k) <LR(k)
- Algoritmul folosit nu depinde de limbaj, gramatica da.

Alternativa la backtracking: analiza predictivă (descendent recursivă)

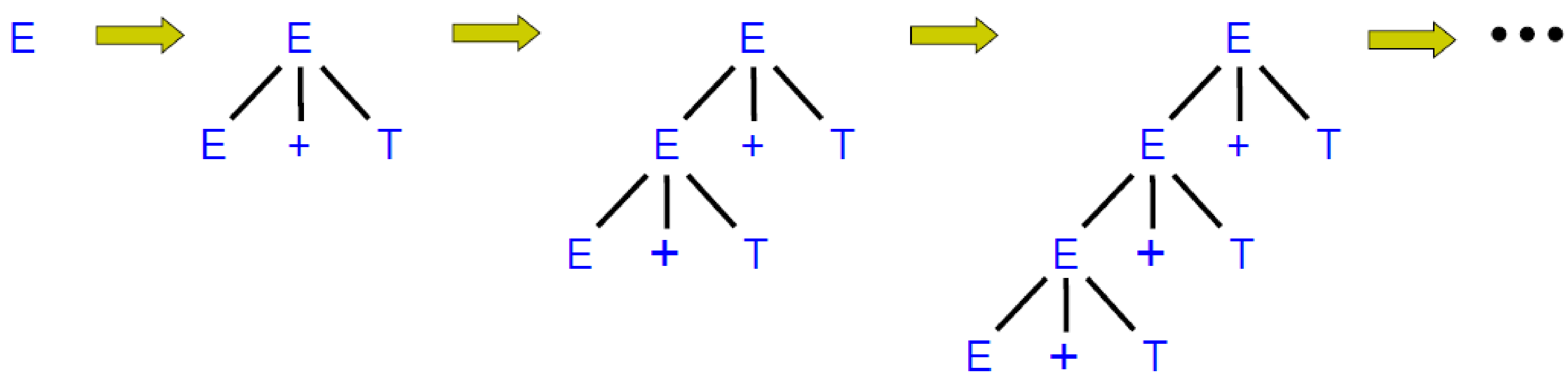
- Fiecare neterminal are o functie care il parseaza
- Daca simbolul apare in partea dreapta a productiei $\rightarrow$ functia se va apela recursiv
- Daca un neterminal apare in partea stanga a mai multor productii $\rightarrow$ se alege una din ele in functie de urmatorii atomi lexicali (lookahead)

Recursivitate stânga

Sa luam gramatica:

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Un parser descendent intra in bucla infinita cand incearca sa parseze aceasta gramatica



(Aho,Sethi,Ullman, pp. 176)

Recursivitate stânga

Cand o gramatica are cel putin o productie de forma

$$A \rightarrow A\alpha$$

spunem ca este o gramatica **recursiva stanga**.

Analizoarele descendente nu functioneaza (fara backtracking) pe gramatici recursive stanga.

Recursivitatea poate sa nu fie imediata

$$A \rightarrow B\alpha$$

$$B \rightarrow A\beta$$

Gramatici de tip 2 – recursivitate stângă

- Un neterminal A este stâng recursiv dacă există măcar o derivare $A \Rightarrow^{+} A\beta$. Dacă gramatica G conţine cel puţin un neterminal stâng recursiv, G este stâng recursivă.
- Un neterminal A este stâng recursiv imediat dacă există o regulă $A \to A\alpha \in P$.

Eliminarea recursivitatii stângi imediate:

- Fie $A \to A\alpha_1 | A\alpha_2 \ldots | A\alpha_k | \beta_1 | \ldots \beta_n$ toate regulile care încep cu A ($\beta_1, \ldots, \beta_n$ nu încep cu A). Fie P_A mulţimea acestor reguli.
- Gramatica G' în care A nu este stâng recursiv imediat:
  - $G' = (N \cup \{A'\}, T, S, P')$
    $P' = P \setminus P_A \cup \{A' \to \alpha_1 A' | \ldots \alpha_k A' | \epsilon, A \to \beta_1 A' | \ldots \beta_n A'\}$

Exemplu

$G = (\{S, A\}, \{a, b, c\}, S, P)$ unde P este:

- $S \rightarrow Ac|c$
- $A \rightarrow Aa|Ab|a|b|Sc$

$G' = (S, A, A', a, b, c, S, P')$ unde P' este:

- $S \rightarrow Ac|c$
- $A \rightarrow aA'|bA'|ScA'$
- $A' \rightarrow aA'|bA'|\epsilon$

Observaţie: A, S stâng recursive

Eliminarea recursivității stânga

- Intrare: $G = (N, T, S, P)$ în formă redusă
- Ieşire: $G' = (N', T, S', P')$, $L(G') = L(G)$, fără recursie stângă

Se ordonează N; fie $N' = N = \{A_1, A_2, \ldots, A_n\}$
for(i = 1; i <=n; i++){
    while($\exists\ A_i \to A_j\alpha \in P : j <= i - 1$) {
        $P = P - \{A_i \to A_j\alpha\}$;
        for($A_j \to \beta \in P$) $P = P \cup \{A_i \to \beta\alpha\}$;
    }
    Se elimină recursia stângă imediată pentru A_i
}
N' este obţinută din N prin adăugarea tuturor neterminalilor nou introduşi iar P' este noua mulţime de reguli

Exemplu

$G = (\{A_1, A_2, A_3\}, \{a, b, c\}, A_1, P)$, unde P:

- $A_1 \rightarrow A_2a|b$
- $A_2 \rightarrow A_3b$
- $A_3 \rightarrow A_1c|c$

Gramatica echivalentă care nu este stâng recursivă:

$G' = (\{A_1, A_2, A_3, A_3'\}, \{a, b, c\}, A_1, P')$, unde P':

- $A_1 \rightarrow A_2a|b$
- $A_2 \rightarrow A_3b$
- $A_3 \rightarrow bcA_3'|cA_3'$
- $A_3' \rightarrow bacA_3'|\epsilon$

Eliminarea recursivității stânga

Se face prin rescrierea gramaticii

List → List Item | Item

$\Rightarrow$

List → Item List'
List' → Item List' | ε

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (\ E\ ) \mid \textbf{id}
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \textbf{id}
\end{aligned}
$$

Exemplu de parser recursiv

```
ParseE() {
ParseT(); ParseE1();
}

ParseE1() {
if (lookahead==T_PLUS)
{
     MatchToken(T_PLUS);
    ParseT();
    ParseE1();
    }
}
```

```
ParseT() {
ParseF(); ParseT1();
}

ParseT1() {
if (lookahead==T_STAR)
{
     MatchToken(T_STAR);
     ParseF();
     ParseT1();
      }
}
```

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (\, E\,) \mid \mathbf{id}$$

```
ParseF() {
if (lookahead == T_LPAREN) {
   MatchToken(T_LPAREN); ParseE(); MatchToken(T_RPAREN);
   }
else
   MatchToken(T_ID);
```

Analiza descendent recursivă

Cum alegem intre doua productii?

Cum stim ce conditii punem la if?

Cand emitem erori?

```
F → ( E )
F → id
T' → *FT'
T' → ϵ
```

```
ParseT1() {
if (lookahead==T_STAR) {
   MatchToken(T_STAR);
    ParseF();
    ParseT1();
    }
else if (lookahead == T_PLUS) { }
else if (lookahead == T_RPAREN) { }
else if (lookahead == T_EOF) { }
else throw error();
}
```

```
ParseF() {
if (lookahead == T_LPAREN) {
   MatchToken(T_LPAREN);
   ParseE();
   MatchToken(T_RPAREN);
   }
else if (lookahead == T_ID) {
   MatchToken(T_ID);
    }
else throw error();
}
```

Cum punem condițiile?

- Folosim doua seturi de terminali –'First' si 'Follow'
  - Plus 'Nullable' –multime de neterminali ce pot deriva in ε.
- Setul de terminali-prefix ai neterminalului u -notat First(u)
  - Setul de terminali care apar pe prima pozitie intr-o derivare legala a lui u
  - Daca u=>* ε, atunci ε e in First(u)
- Setul de terminali care pot urma dupa u –notat Follow(u)

Cum construim FIRST

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (\, E\,) \mid \mathbf{id}$

SETS:

FIRST(**id**) = {**id**}
FIRST(∗) = {∗}
FIRST(**+**) = {**+**}
FIRST(**(**) = {**(**}
FIRST(**)**) = {**)**}
FIRST(E') = $\cancel{\{\varepsilon\}}$ {**+**, ε}
FIRST(T') = $\cancel{\{\varepsilon\}}$ {**∗**, ε}
FIRST(F) = {(, id}
FIRST(T) = FIRST(F) = {(, id}
FIRST(E) = FIRST(T) = {(, id}

FIRST (pseudocod):

1. If X is a <u>terminal</u>, FIRST(X) = {X}
2. If $X \rightarrow \varepsilon$, then $\varepsilon \in$ FIRST(X)
3. If $X \rightarrow Y_1\, Y_2 \cdots Y_k$
   and $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$
   and $a \in$ FIRST(Y_i)
   then $a \in$ FIRST(X)
4. If $X \rightarrow Y_1 Y_2 \cdots Y_k$
   and $a \in$ FIRST(Y_1)
   then $a \in$ FIRST(X)

(Aho,Sethi,Ullman, pp. 189)

Cum construim FOLLOW

FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid \textbf{id}$

SETS:

FOLLOW(E) = ~~{$}~~ {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = {), $}

FOLLOW – pseudocod:

1. If S is the start symbol, then $\$ \in$ FOLLOW(S)
2. If $A \rightarrow \alpha B\beta$,
   and $a \in$ FIRST(β)
   and $a \neq \varepsilon$
   then $a \in$ FOLLOW(B)
3. If $A \rightarrow \alpha B$
   and $a \in$ FOLLOW(A)
   then $a \in$ FOLLOW(B)

3a. If $A \rightarrow \alpha B\beta$
   and $\beta \overset{*}{\Rightarrow} \varepsilon$
   and $a \in$ FOLLOW(A)
   then $a \in$ FOLLOW(B)

A si B sunt neterminali,
α si β siruri de terminali si neterminali

(Aho,Sethi,Ullman, pp. 189)

FIRST(E') = {+, ε}
FIRST(T') = {* , ε}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

Cum construim FOLLOW

GRAMMAR:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \mathbf{id}$

FOLLOW rules:

1. If S is the start symbol, then $\$ \in$ FOLLOW(S)
2. If $A \rightarrow \alpha B \beta$,
   and $a \in$ FIRST(β)
   and $a \neq \varepsilon$
   then $a \in$ FOLLOW(B)
3. If $A \rightarrow \alpha B$
   and $a \in$ FOLLOW(A)
   then $a \in$ FOLLOW(B)

3a. If $A \rightarrow \alpha B \beta$
   and $\beta \stackrel{*}{\Rightarrow} \varepsilon$
   and $a \in$ FOLLOW(A)
   then $a \in$ FOLLOW(B)

SETS:

FOLLOW(E) = {), $}
FOLLOW(E') = {), $}
FOLLOW(T) = ~~{), $}~~ {+,), $}

(Aho,Sethi,Ullman, pp. 189)

Cum construim FOLLOW

FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

GRAMMAR:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | **id**

SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $\$ \in FOLLOW(S)$
2. If $A \rightarrow \alpha B \beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \varepsilon$
   then $a \in FOLLOW(B)$
3. If $A \rightarrow \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \rightarrow \alpha B \beta$
   and $\beta \stackrel{*}{\Rightarrow} \varepsilon$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

(Aho,Sethi,Ullman, pp. 189)

FIRST(E') = {+, ε}
FIRST(T') = {*, ε}
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

Cum construim FOLLOW

GRAMMAR:

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid$ **id**

SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = {**+**, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $ \in FOLLOW(S)
2. If $A \to \alpha B \beta$,
   and $a \in FIRST(\beta)$
   and $a \neq \varepsilon$
   then $a \in FOLLOW(B)$
3. If $A \to \alpha B$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

3a. If $A \to \alpha B \beta$
   and $\beta \overset{*}{\Rightarrow} \varepsilon$
   and $a \in FOLLOW(A)$
   then $a \in FOLLOW(B)$

(Aho,Sethi,Ullman, pp. 189)

Cum construim FOLLOW

FIRST(E') = {+, ε}
$\boxed{\text{FIRST(T') = \{*, ε\}}}$
FIRST(F) = {(, id}
FIRST(T) = {(, id}
FIRST(E) = {(, id}

GRAMMAR:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | **id**

SETS:

FOLLOW(E) = {**)**, **$**}
FOLLOW(E') = { **)**, **$**}
FOLLOW(T) = {**+**, **)**, **$**}
FOLLOW(T') = {**+**, **)**, **$**}
FOLLOW(F) = ~~{+,), $}~~ {**+**, *****, **)**, **$**}

FOLLOW rules:

1. If S is the start symbol, then $ ∈ FOLLOW(S)
2. If A → αBβ,
   and a ∈ FIRST(β)
   and a ≠ ε
   then a ∈ FOLLOW(B)
3. If A → αB
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

3a. If A → αBβ
   and $\beta \overset{*}{\Rightarrow} \varepsilon$
   and a ∈ FOLLOW(A)
   then a ∈ FOLLOW(B)

(Aho,Sethi,Ullman, pp. 189)

Algoritmul generic recursiv LL(1)

- Pentru fiecare non-terminal A se creaza o functie de parsare.
- Pentru fiecare regula $A \rightarrow \alpha$ se adauga un test
  - if (lookahead in FIRST(αFOLLOW(A)))
- Pentru fiecare nonterminal din α se apeleaza functia de parsare.
- Pentru fiecare terminal din α, se verifica lookahead-ul (match)

```
ParseA() {
if (lookahead in FIRST(a B … x FOLLOW(A)) {
   MatchToken(a); ParseB(); … MatchToken(x);
}
else if (lookahead in FIRST(C D … y FOLLOW(A)) {
   ParseC(); ParseD(); … MatchToken(y);
}
…
else throw error();
}
```

```
A → a B … x
A → C D … y
…
```

Factorizare stânga

Sa analizam o instructiune if:

if_statement -> IF expression THEN statement ENDIF |
IF expression THEN statement ELSE statement ENDIF

Pentru a o putea analiza LL, trebuie factorizata stanga:

if_statement -> IF expression THEN statement close_if
close_if -> ENDIF | ELSE statement ENDIF

```
void ParseIfStatement()
{
   MatchToken(T_IF);
   ParseExpression();
   MatchToken(T_THEN);
   ParseStatement();
   ParseCloseIf();
}
```

```
void ParseCloseIf()
{
   if (lookahead == T_ENDIF)
      lookahead = yylex();
   else {
      MatchToken(T_ELSE);
      ParseStatement();
      MatchToken(T_ENDIF);
   }
}
```

Factorizare stânga

- Cazul general:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \delta$$

- Factorizat:

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

Eliminarea ambiguităților

- Ambiguu: $E \to E + E \mid E * E \mid a \mid (E)$

1. $E \to E + T \mid T$
   $T \to T * F \mid F$
   $F \to a \mid (E)$

2. $E \to T + E \mid T$
   $T \to F * T \mid F$
   $F \to a \mid (E)$

- Apare explicita precedenta operatorilor, asociativitatea stanga sau dreapta

Eliminarea ambiguităților

- Productii ce pot produce ambiguitati:
  $X \rightarrow aAbAc$

- Cazul general:
  $A \rightarrow A\ B\ A \mid \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

- Dezambiguizat:
  $A \rightarrow A'\ B\ A \mid A'$
  $A' \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

„Dangling else”

- Ambiguu:

  Statement –> if Expr then Statement
  | if Expr then Statement else Statement
  | Other

"**if** Expr **then if** Expr **then** Other **else** Other"

- Factorizat ramane tot ambiguu:

  Statement –> if Expr then Statement CloseIf
  | Other
  CloseIf -> ε | else Statement

- Algoritmul de parsare poate rezolva implicit unele ambiguitati.

„Dangling else"

- Ambiguu:

  Statement –> if Expr then Statement
  | if Expr then Statement else Statement
  | Other

"if Expr **then if** Expr **then** Other **else** Other"

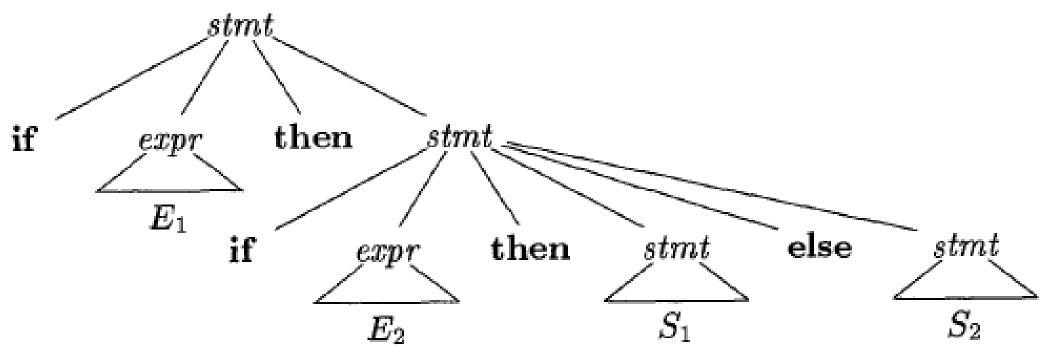
- Dezambiguizat:

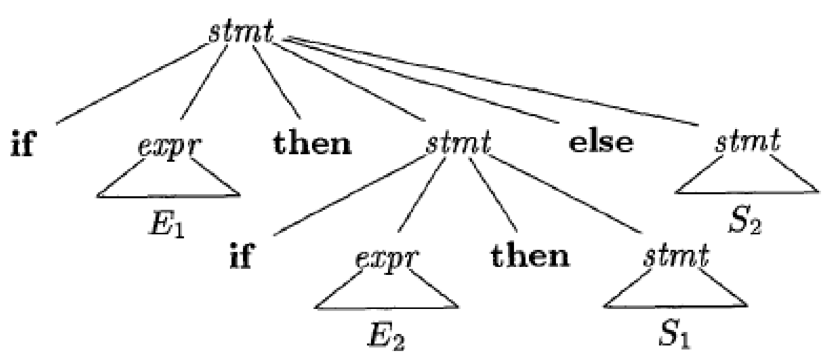
  Statement -> Open | Closed
  Closed -> if Expr then Closed else Closed
  | Other
  Open -> if Expr then Statement
  | if Expr then Closed else Open

- Nu poate fi factorizat - limbajul nu este LL(1), dar este LR(1)

„Dangling else" - ambiguitate

if E_1 **then if** E_2 **then** S_1 **else** S_2





Automatizarea parsării

- Echivalenta cu un automat push-down
- Parsarea se poate face cu un automat si o tabela.

Cursul viitor:

- Automate push-down
- Parsere LL
- Analiza sintactică LR

Întrebări?