FUNDAMENTELE PROIECTĂRII COMPILATOARELOR

CURS 3

Gianina Georgescu

CUPRINSUL CURSULUI 3

ANALIZA LEXICALĂ

- Algoritm de transformarea ExprReg – AFD (funcțiile *nullable*, *firstpos*, *lastpos*, *followpos*, exemple)
- Translator finit – definiție, exemple, proprietăți
- Analiza sintactică - introducere
- Gramatici independente de context

CONSTRUCȚIA DIRECTĂ A UNUI $\boldsymbol{AFD}$ PORNIND DE LA O EXPRESIE REGULATĂ

Fie r expresie regulată peste Σ și # un simbol nou, $\# \notin \Sigma$.

Numim **extensia lui $\boldsymbol{r}$** expresia $(r)\#$.

Vom construi un AFD care recunoaște r. Pentru aceasta construim arborele sintactic T asociat lui $(r)\#$ apoi, printr-o parcurgere în adâncime a lui T, calculăm 4 funcții: $\boldsymbol{nullable}$, $\boldsymbol{firstpos}$, $\boldsymbol{lastpos}$, $\boldsymbol{followpos}$**.**

- Numerotăm frunzele lui T de la stânga la dreapta cu numere (numite **poziții**) de la 1 la n (unde n este asociat lui #). Se vor numerota doar frunzele etichetate cu o literă din $\Sigma \cup \{\#\}$. Nodurile etichetate cu λ nu se etichetează.

- Nodurile interne ale lui T sunt etichetate cu unul dintre cei 3 operatori de bază ai expresiilor regulate: $|, \cdot, *$

- Nodurile externe ale lui T sunt etichetate cu simboluri din $\Sigma \cup \{\lambda, \#\}$.

- Pentru un nod n al lui T care este rădăcina unui subarbore T_1 al lui T, ce corespunde subexpresiei r_1, calculăm:

➢ $nullable(n) = true$ dacă și numai dacă $\lambda \in L(r_1)$

➢ $firstpos(n)$ reprezintă pozițiile care corespund primelor simboluri ale șirurilor din $L(r_1)$

➢ $lastpos(n)$ reprezintă pozițiile care corespund ultimelor simboluri ale șirurilor din $L(r_1)$

➢ $followpos(p)$ (unde p este o poziție, $1 \leq p \leq n$) cuprinde toate pozițiile $q, 1 \leq q \leq n$, astfel încât în $L((r)\#)$ există un șir w de forma $w = w' a_1 a_2 w''$, $a_1, a_2 \in \Sigma$, iar poziția corespunzătoare lui a_1 este p, iar a lui a_2 este q.

Calculul pentru *nullable*, *firstpos*, *lastpos*

Nodul n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
n este frunză etichetată cu λ	true	$\emptyset$	$\emptyset$
$n = a \in \Sigma$, frunza având asociată poziția i	false	$\{i\}$	$\{i\}$
$n \mid$ $\swarrow\searrow$ $c_1 \quad c_2$	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$

$n \cdot$
 $\swarrow\searrow$
 $c_1 \quad c_2$
	$nullable(c_1)$ and $nullable(c_2)$	if ($nullable(c_1)$)
		 $firstpos(c_1) \cup firstpos(c_2)$
		 else
		 $firstpos(c_1)$
			if ($nullable(c_2)$)
			 $lastpos(c_1) \cup lastpos(c_2)$
			 else
			 $lastpos(c_2)$

$n \star$ $\downarrow$ c	true	$firstpos(c)$	$lastpos(c)$

Calculul lui *followpos*

- Pentru fiecare nod concatenare cu descendenții c1 și c2, atunci:

  $\forall i \in lastpos(c1),\ followpos(i) \supseteq firstpos(c2)$
- Pentru fiecare nod * având descendentul direct c, atunci:

  $\forall i \in lastpos(c),\ followpos(i) \supseteq firstpos(c)$

ALGORITM PENTRU OBȚINEREA ARBORELUI SINTACTIC CORESPONDENT UNEI EXPRESII REGULATE

Fie r expresie regulată peste alfabetul Σ. Se aplică algoritmul Shunting Yard pentru transformarea expresiei (r)# în formă postfixată.

Se scanează elementele (token-ii) expresiei în ordinea postfixată. Folosim o stivă, inițial vidă.

Se repetă cât timp există token-i:

- dacă token-ul curent este un simbol $a \in \Sigma \cup \{\lambda, \#\}$, se construiește arborele T_a care constă dintr-un singur nod etichetat cu a. Se introduce T_a pe stivă.

- dacă token-ul curent este $op \in \{\cdot, |\}$, se scot din stivă arborii T_1, T_2 și se introduce pe stivă arborele T_{op} având ca rădăcină nodul etichetat cu op, arborele stâng T_2 și arborele drept T_1.

- dacă token-ul curent este $*$, se scoate de pe stivă arborele T și se introduce pe stivă arborele T_* având ca rădăcină nodul etichetat cu $*$ și ca unic descendent pe T. Putem seta pe T ca subarbore stâng al lui T_*, subarborele drept fiind setat cu $null$.

În final, arborele (unic, dacă expresia în formă postfixată este corectă) rămas în stivă reprezintă arborele sintactic al expresiei regulate inițiale.

Algoritmul ExpReg ⇒ AFD

Intrare: $(r)\#$, unde r expresie regulata peste Σ

Ieșire: AFD $A = (Q, \Sigma, \delta, s, F), L(A) = L(r)$

Metoda:

1. Se construieste arborele sintactic T pentru $(r)\#$;
2. Se calculează funcțiile *nullable*, *firstpos*, *lastpos*, *followpos* ca în tabelul de mai sus, prin parcurgerea în adâncime a lui T.
3. $Q \subseteq 2^{\{1,2,\ldots,n\}}$, unde $1,2,\ldots,n$ reprezinta pozițiile asociate frunzelor lui T. Stările lui A sunt mulțimi de poziții.

Algoritmul ExpReg ⇒ AFD

{

$s \leftarrow firstpos(rad);$ // rad este radacina lui T

$Q \leftarrow \{s\}$, s stare nemarcata;

while (exista $M \in Q$ nemarcata)

{

marcheaza M;

for ($a \in \Sigma$){

$V \leftarrow \{q | \exists p \in M, p$ pozitie ce corespunde lui $a, q \in followpos(p)\}$;

if($V \notin Q$) $Q \leftarrow Q \cup \{V\}$ ***V*** **stare nemarcată;**

$\delta(M, a) \leftarrow V$;

} \\end_for

} \\end_while

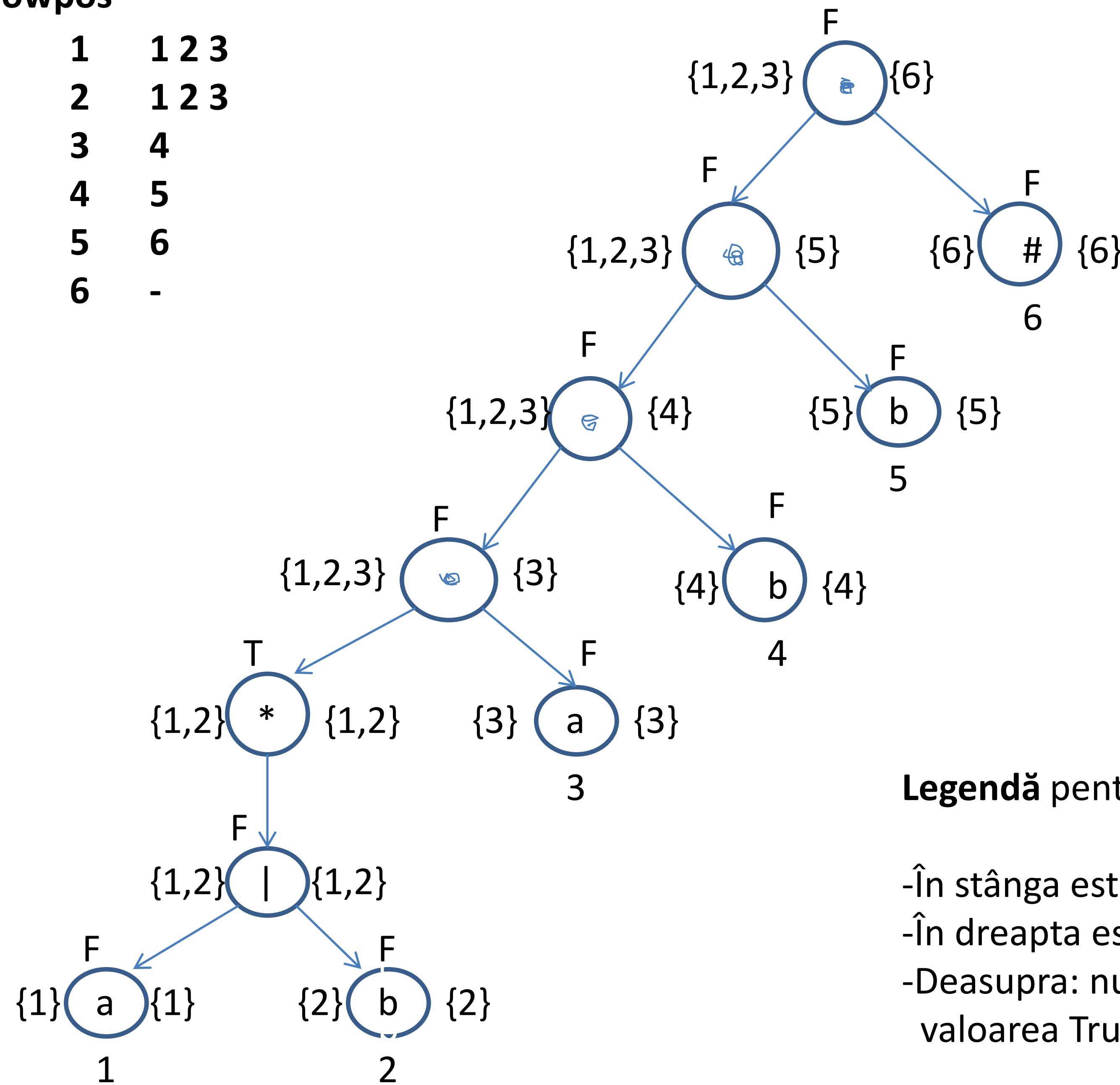
$F \leftarrow \{M \in Q | M$ contine pozitia lui #$\}$;

}

Exemplu pentru expresia (a|b)*abb, cu extensia ((a|b)*abb)#

followpos

1	**1 2 3**
2	**1 2 3**
3	**4**
4	**5**
5	**6**
6	**-**

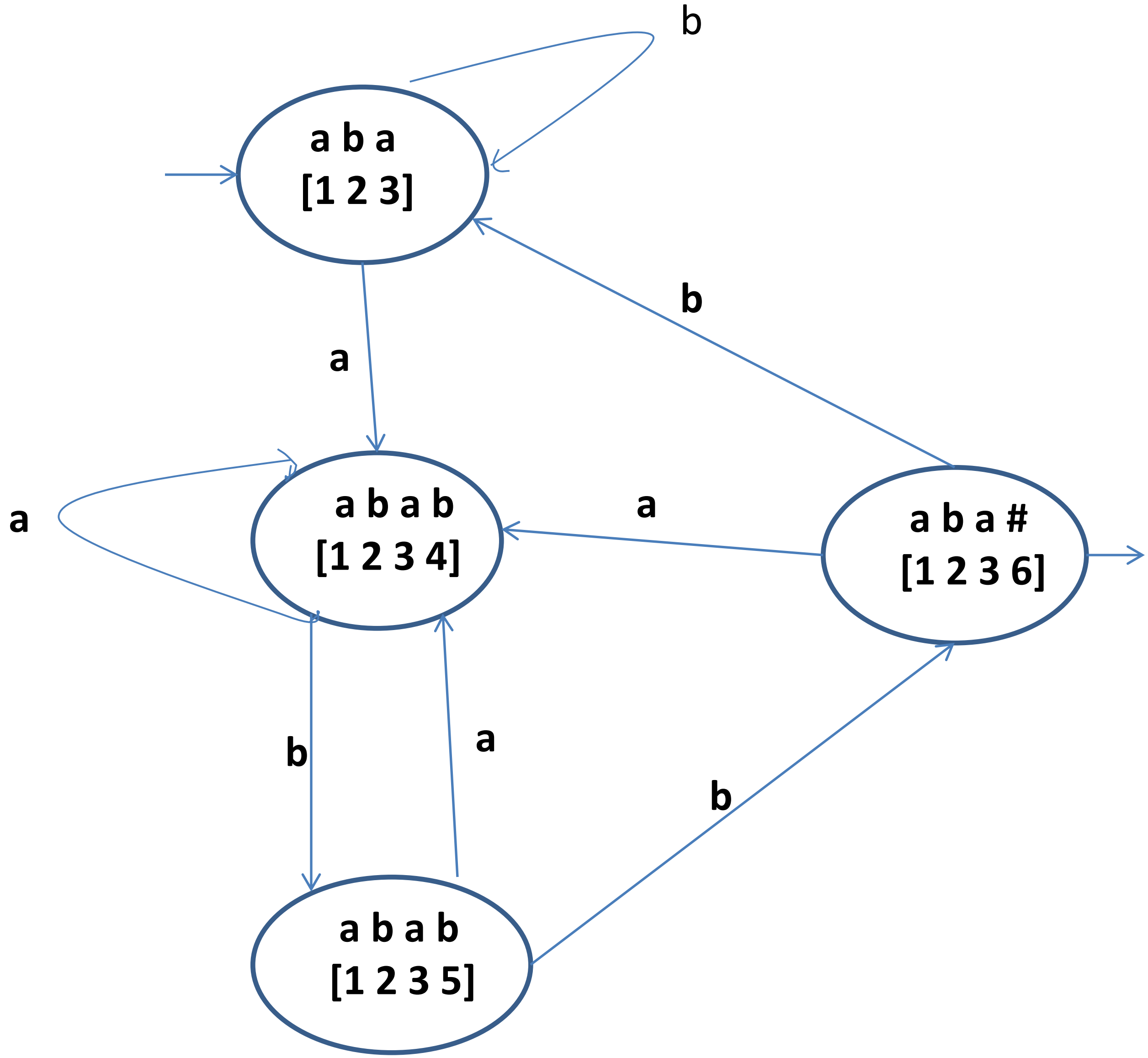


Legendă pentru un nod n:

-În stânga este figurat firstpos(n)
-În dreapta este figurat lastpos(n)
-Deasupra: nullable(n) cu
valoarea True sau False

followpos

1	1 2 3
2	1 2 3
3	4
4	5
5	6
6	-



AFD echivalent cu (a|b)*abb
(nu este neaparat minimal)

TRANSLATOARE FINITE

Sunt ca și automatele finite, cu deosebirea că la fiecare tranziție produc o ieșire. Formal:

Definiție. Un translator finit nedeterminist cu λ-tranziții este o structură de forma:

$T = (Q, V_i, V_e, \delta, s, F)$, unde:

- Q este mulțimea stărilor
- V_i este alfabetul de intrare
- V_e este alfabetul de ieșire
- $\delta: Q \times (V_i \cup \{\lambda\}) \to \mathcal{P}_{fin}(Q \times V_e^*)$

  ↑ mulțimea părților finite ale lui $(Q \times V_e^*)$
- $s \in Q$ este starea initiala a lui T
- $F \subseteq Q$ mulțimea stărilor finale

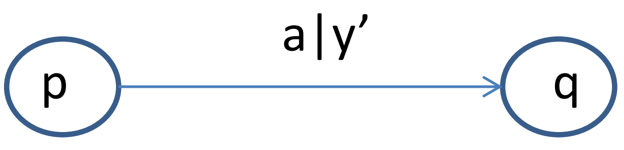
TRANSLATOARE FINITE

Descriere instantanee (configurație, instanță) a lui T: triplet de forma (q, x, y), unde:

- $q \in Q$ starea curentă a translatorului
- $x \in V_i^*$ șirul curent scanat din intrare
- $y \in V_e^*$ șirul de ieșire curent

Mișcare a lui T:

- $(p, ax, y) \vdash (q, x, yy')$ ddacă $(q, y') \in \delta(p, a)$, $p, q \in Q,\ a \in V_i \cup \{\lambda\},\ x \in V_i^*,\ y, y' \in V_e^*$



Închiderea reflexivă și tranzitivă a relației $\vdash$ este notată cu $\vdash^*$ (reprezintă 0 sau mai multe mișcări)

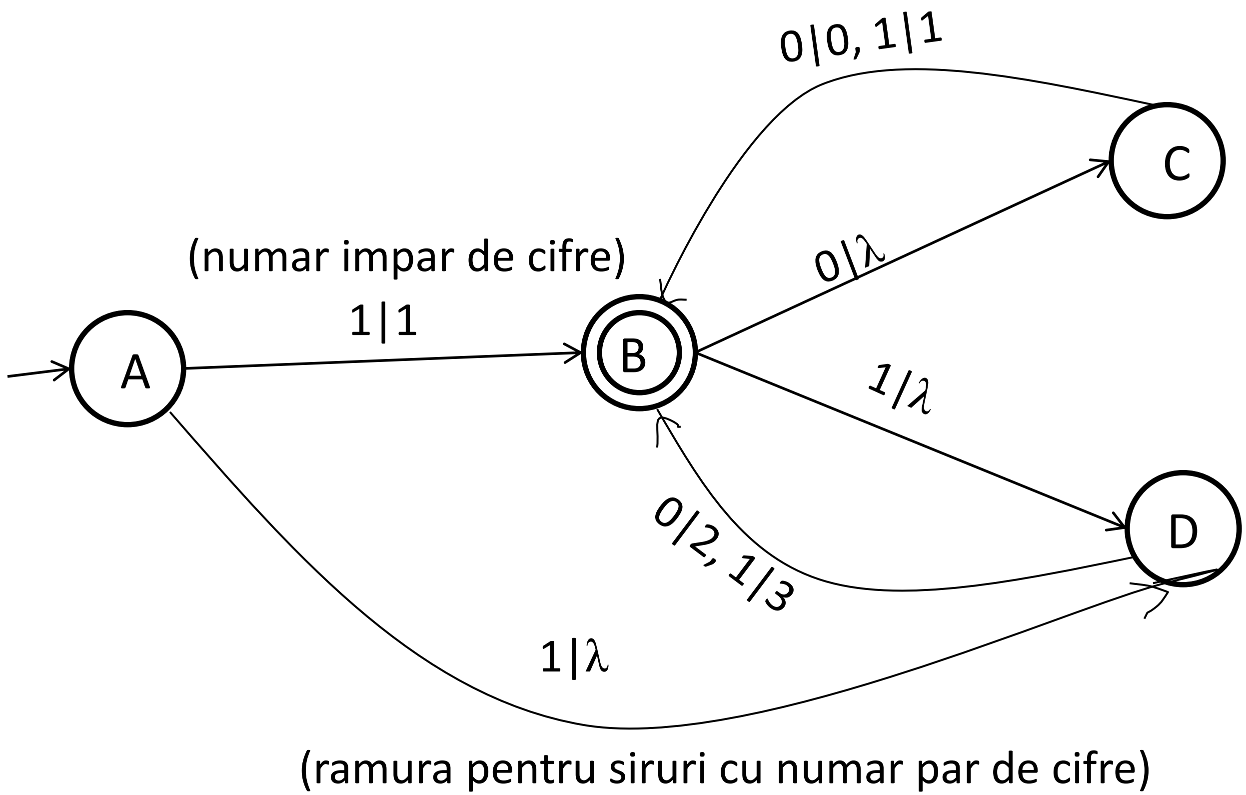
TRANSLATAREA DEFINITĂ DE TRANSLATORUL *T*

- Pentru un șir $x \in V_i^*$:
  $T(x) = \{\, y \in V_e^* | (s, x, \lambda) \vdash^* (q, \lambda, y), q \in F\}$
- Pentru un limbaj $L \subseteq V_i^*$:
  $$T(L) = \bigcup_{x \in L} T(x)$$
- Translatarea definita de T la modul global:
  $\tau(T) = \{(x, y) | x \in V_i^*, y \in V_e^*, y \in T(x)\}$

EXEMPLU

Translator care translatează orice șir $w \in \{0,1\}^*$ care începe cu '1' în șirul echivalent din baza 4.
Astfel: $100110010 \rightarrow 10302$, $101100 \rightarrow 230$

T=({A,B,C,D}, {0,1}, {0,1,2,3}, δ, A, {B})



(A,101100,λ)→
(B,01100, 1)→
(C,1100,1) →
(B,100,11) →
(D,00,11) →
(B,0,112) →
(C, λ,112),
C nefinala

(A,101100,λ)→
(D,01100, λ)→
(B,1100,2) →
(D,100,2) →
(B,00,23) →
(C,0,23) →
(B, λ,230)
B finala

T(101100)={230}

PROPRIETĂȚI ALE TRANSLATOARELOR FINITE

Propoziția 1. Familia limbajelor regulate, $\mathcal{L}_{REG}$, este închisă la translatări finite.
Aceasta înseamnă că dacă $L \in \mathcal{L}_{REG}$ și T este un translator finit, atunci $T(L) \in \mathcal{L}_{REG}$, unde $L \subseteq \Sigma^*$ iar alfabetul de intrare al lui T este Σ.

Propoziția 2. Familia limbajelor independente de context, $\mathcal{L}_{CF}$, este închisă la translatări finite.
Aceasta înseamnă că dacă $L \in \mathcal{L}_{CF}$ și T este un translator finit, atunci $T(L) \in \mathcal{L}_{CF}$, unde $L \subseteq \Sigma^*$ iar alfabetul de intrare al lui T este Σ.

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
- $S \epsilon N$ este simbolul de start
- P este mulțimea producțiilor de forma:

$$A \rightarrow x, A \epsilon N, x \epsilon (N \; U \; \Sigma)^*$$

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:

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

EXEMPLE DE GRAMATICI INDEPENDENTE DE CONTEXT ȘI LIMBAJELE GENERATE

$G_1 = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to \lambda\})$

O derivare in G_1: $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow \cdots \Rightarrow a^nSb^n \Rightarrow a^nb^n$;

$L(G_1) = \{a^nb^n | n \geq 0\}$

$G_2 = (\{S\}, \{a, b\}, S, \{S \to aSa, S \to bSb,$
$S \to a, S \to b, S \to \lambda\})$

$L(G_2) = \{w \in \{a, b\}^* | w \text{ palindrom}\}$

EXEMPLE DE GRAMATICI INDEPENDENTE DE CONTEXT ȘI LIMBAJELE GENERATE

$$G_3 = (\{S, A, B\}, \{a, b\}, S,$$
$$\{S \to AB, A \to aAb, B \to aBb, A \to \lambda, B \to \lambda\})$$
$$L(G_3) = \{a^m b^m a^n b^n | m, n \geq 0)$$

$$G_4 = (\{S\}, \{a, b\}, S, \{S \to aSb, S \to aS, S \to \lambda \})$$
$$L(G_4) = \{a^m \, b^n | m \geq n \geq 0\}$$

GRAMATICI INDEPENDENTE DE CONTEXT

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

- rădăcina este etichetată cu un neterminal
- 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

Definiție. Spunem că o gramatică independentă de context $G = (N, \Sigma, S, P)$ este **ambiguă** dacă există $w \in L(G)$ pentru care este adevărată una dintre afirmațiile (echivalente) următoare:

a) w are cel puțin 2 derivări stângi distincte

b) w are cel puțin 2 derivări drepte distincte

c) w are cel puțin 2 arbori de derivare distincți

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

La modul general, problema dacă o gramatică independentă de context este ambiguă sau nu este nedecidabilă.

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

EXEMPLU DE GRAMATICĂ INDEPENDENTĂ DE CONTEXT PENTRU EXPRESII

$E \rightarrow \texttt{int}$

$E \rightarrow E \; Op \; E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

Simbolurile care apar în stânga regulilor, **E, Op** sunt neterminali. Primul neterminal ce apare în lista de producții este simbol de start (în acest caz **E**)
Celelalte simboluri, care apar în șirurile din dreapa regulilor și nu sunt neterminali, respectiv **int, (,), +, -, *, /** sunt terminali.

SCRIERE SIMPLIFICATĂ PENTRU GRAMATICA EXPRESIILOR CU AJUTORUL META-CARACTERULUI '|'

$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

EXEMPLU DE GRAMATICĂ INDEPENDENTĂ DE CONTEXT CE REDĂ SINTAXA PENTRU EXPRESII

Cu notația Backus-Naur

$$< expr >::=< term > " + " < expr > | < term >$$

$$< term >::=< factor > " * " < term > | < factor >$$

$$< factor >::=< const > | < var > |"(" < expr > ")"$$

$$< const >::=< digit > | < digit >< const >$$

$$< digit >::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"$$

Neterminalii apar în paranteze unghiulare.
| delimitează membrii drepți ai producțiilor cu același neterminal în membrul stâng.
Ceea ce apare între ghilimele trebuie să apară ca atare într-un șir corect (ce se potrivește cu regula respectivă)

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

; marchează sfârșitul producțiilor unui neterminal

| delimitează producțiile unui același neterminal

Primul neterminal ce apare în secțiunea de reguli este simbol de start dacă nu este folosită directiva specială pentru desemnarea acestui simbol

EXEMPLU DE GRAMATICĂ CF PENTRU UN LIMBAJ DE PROGRAMARE

```
BLOCK  →  STMT
       |  { STMTS }

STMTS  →  ε
       |  STMT STMTS

STMT   →  EXPR;
       |  if (EXPR) BLOCK
       |  while (EXPR) BLOCK
       |  do BLOCK while (EXPR);
       |  BLOCK
       |  ...

EXPR   →  identifier
       |  constant
       |  EXPR + EXPR
       |  EXPR – EXPR
       |  EXPR * EXPR
       |  ...
```

EXEMPLU DE DERIVARE STÂNGĂ

```
BLOCK →  STMT
       | { STMTS }

STMTS →  ε
       | STMT STMTS

STMT  →  EXPR;
       | if (EXPR) BLOCK
       | while (EXPR) BLOCK
       | do BLOCK while (EXPR);
       | BLOCK
       | ...

EXPR  →  id
       | constant
       | EXPR + EXPR
       | EXPR – EXPR
       | EXPR * EXPR
       | EXPR = EXPR
       | ...
```

```
  STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

EXEMPLU DE DERIVARE DREAPTĂ ÎN GRAMATICA

$$E \rightarrow \texttt{int} \mid E\ Op\ E \mid (E)$$

$$Op \rightarrow + \mid - \mid * \mid /$$

E	E
$\Rightarrow E\ Op\ E$	$\Rightarrow E\ Op\ E$
$\Rightarrow \texttt{int}\ Op\ E$	$\Rightarrow E\ Op\ (E)$
$\Rightarrow \texttt{int} * E$	$\Rightarrow E\ Op\ (E\ Op\ E)$
$\Rightarrow \texttt{int} * (E)$	$\Rightarrow E\ Op\ (E\ Op\ \texttt{int})$
$\Rightarrow \texttt{int} * (E\ Op\ E)$	$\Rightarrow E\ Op\ (E + \texttt{int})$
$\Rightarrow \texttt{int} * (\texttt{int}\ Op\ E)$	$\Rightarrow E\ Op\ (\texttt{int} + \texttt{int})$
$\Rightarrow \texttt{int} * (\texttt{int} + E)$	$\Rightarrow E * (\texttt{int} + \texttt{int})$
$\Rightarrow \texttt{int} * (\texttt{int} + \texttt{int})$	$\Rightarrow \texttt{int} * (\texttt{int} + \texttt{int})$

ARBORE DE DERIVARE PENTRU O DERIVARE STÂNGĂ

E

⇒ E Op E

⇒ int Op E

⇒ int * E

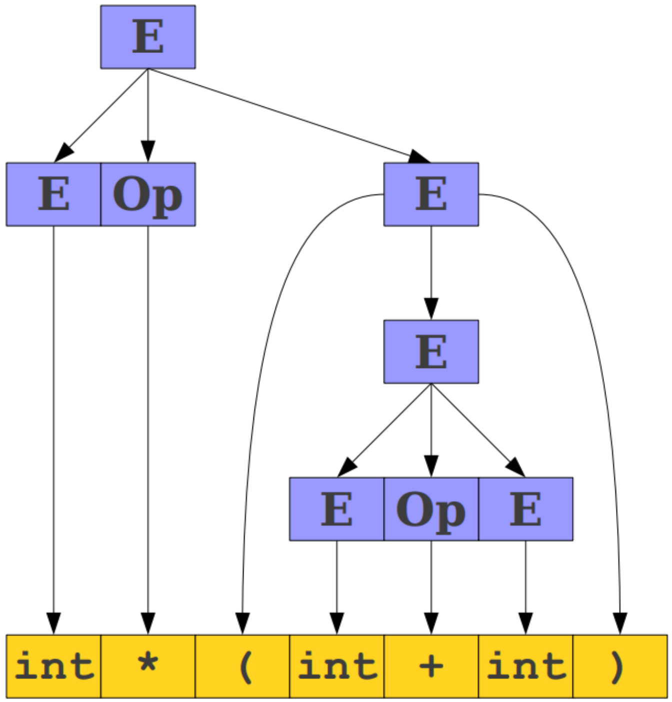
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



ARBORE DE DERIVARE PENTRU O DERIVARE DREAPTĂ

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

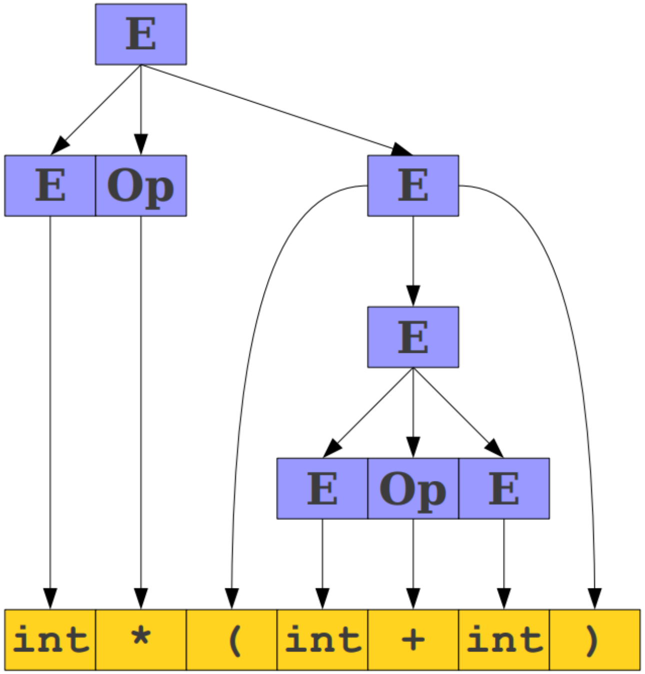
⇒ E Op (E Op int)

⇒ E Op (E + int)

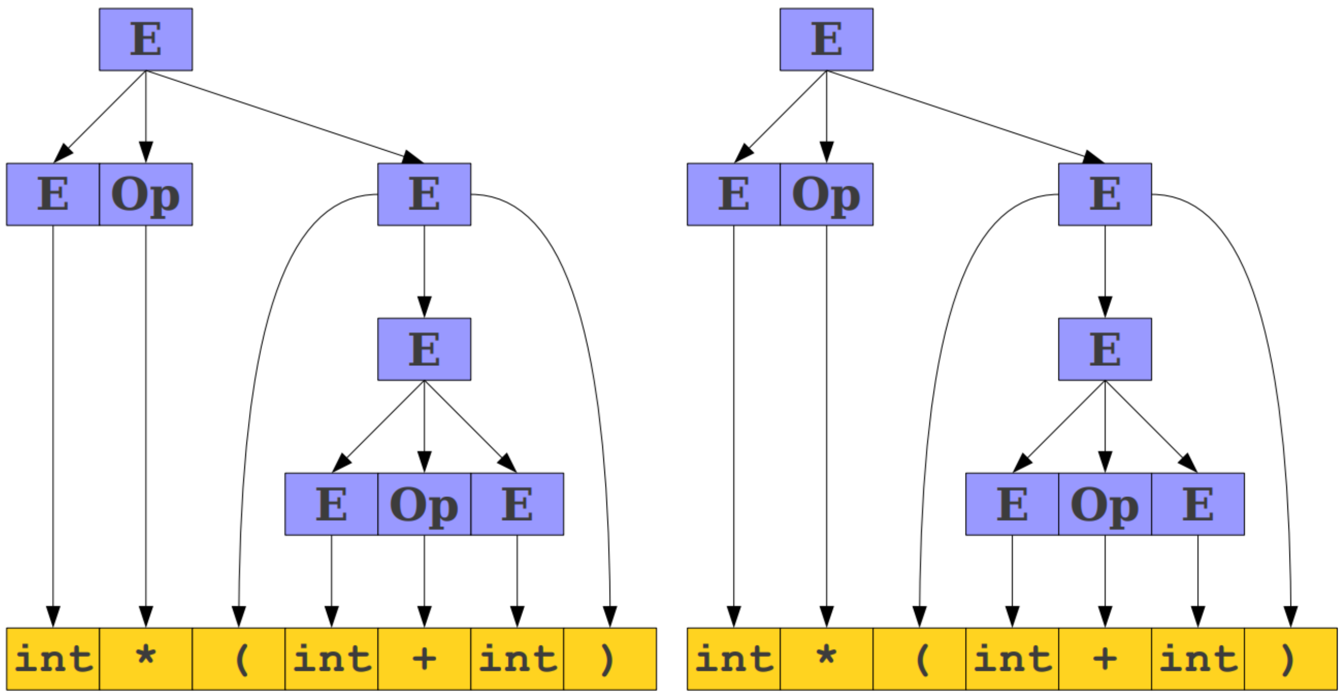
⇒ E Op (int + int)

⇒ E * (int + int)

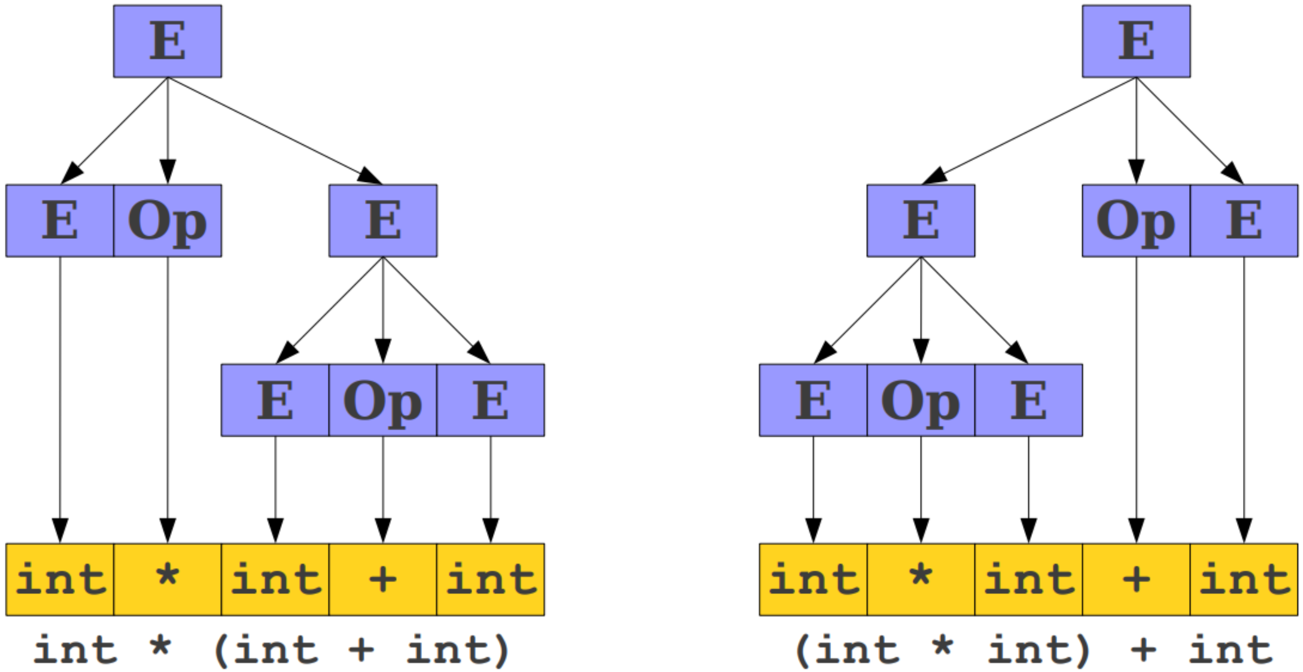
⇒ int * (int + int)



COMPARAȚIE ÎNTRE CEI 2 ARBORI DE DERIVARE



EXEMPLU DE AMBIGUITATE pentru șirul
$int * int + int$, ce are 2 arbori de derivare distincți



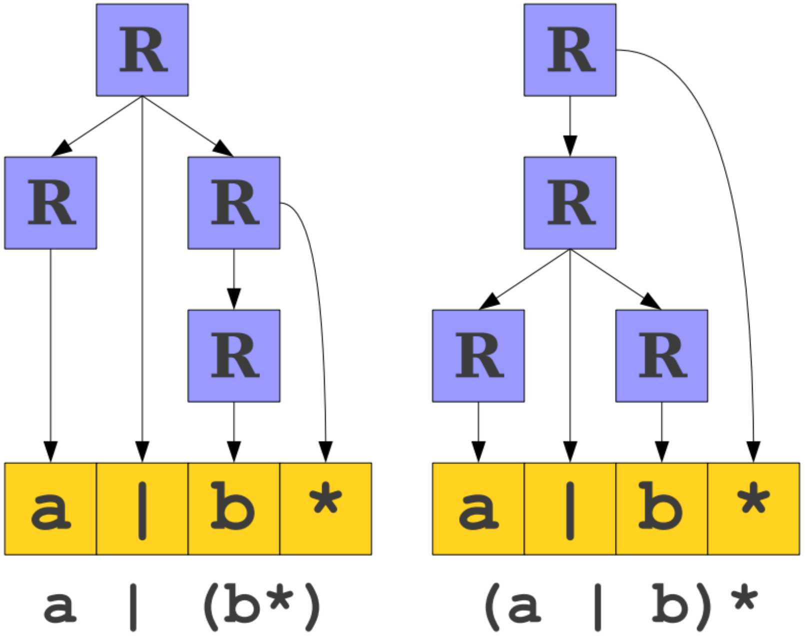
Arborele din stânga sugerează evaluarea expresiei $int * (int + int)$.
Arborele din dreapta sugerează evaluarea expresiei $(int * int) + int$.

EXEMPLU DE GRAMATICĂ INDEPENDENTĂ DE CONTEXT CARE GENEREAZĂ EXPRESIILE REGULATE!

$$R \longrightarrow a|b|c| \ldots$$
$$R \longrightarrow \lambda$$
$$R \longrightarrow RR$$
$$R \longrightarrow R"|"R$$
$$R \longrightarrow R" * "$$

GRAMATICA DE MAI JOS ESTE AMBIGUĂ

$R \longrightarrow a|b|c| \ldots$
$R \longrightarrow \lambda$
$R \longrightarrow RR$
$R \longrightarrow R"|"R$
$R \longrightarrow R" * "$



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:

$$Exp \to Exp + Exp \mid Exp * Exp \mid n$$

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:

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

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