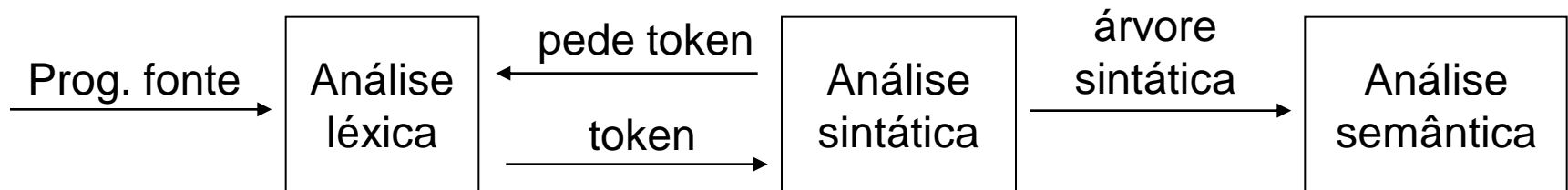


Análise Sintática I: Analisadores Descendentes com Retrocesso

Definição

- A análise sintática é o processo de determinar
 - se uma cadeia de átomos (tokens), isto é, o programa já analisado pelo analisador léxico, pode ser gerado por uma gramática
- Seu objetivo é a **construção da árvore sintática** ou apenas **a decisão** se a cadeia fornecida é ou não uma sentença da gramática que define a linguagem

Papel da Análise Sintática na Estrutura de um Compilador



Formas de análise sintática

- A análise sintática pode ser:
 - Top-down (A.S. Descendente), na qual a construção da árvore de derivação/sintática (explicitamente ou não) é da raiz para as folhas
 - Bottom-up (A.S. Ascendente), que começa das folhas para o símbolo inicial

Métodos de análise

- Existem métodos universais de análise sintática, como o algoritmo de **Cocke-Younger-Kasami** e o de **Earley**,
 - que trabalham com qualquer tipo de gramática livre de contexto,
- Mas são ineficientes para se usar na produção de compiladores,
 - pois são de ordem de $O(n^3)$, com n sendo o tamanho da cadeia de tokens.

Métodos de análise

- Earley, em 1970, apresentou um método de A.S. Top-Down sem Backtracking que reconhece todas as GLCs com $O(n^3)$;
 - mas se elas forem não ambíguas, o tempo é $O(n^2)$ e chega a ser $O(n)$ para muitas linguagens.

Mais detalhes em:

www.cs.uiowa.edu/~RVS/Courses/Compiler/Notes/earley.pdf

Métodos de análise

Os métodos Top-Down e Bottom-Up mais eficientes e interessantes são **determinísticos ($O(n)$)**.

Eles trabalham somente com **subclasses de GLCs**, mas muitas dessas subclasses, por exemplo, as gramáticas **LL(k)** e **LR(k)** são bastante expressivas para descrever a maioria das linguagens de programação.

■ Duas delas são de interesse imediato:

- LL(1): Left to right scan, Left-most derivation, 1 token look-ahead
- LR(1): Left to right scan, Right-most derivation, 1 token look-ahead

Métodos de análise

Veremos 3 métodos de A.S.D. e suas vantagens e desvantagens:

- A.S.D. Recursiva com Retrocesso (tentativa e erro na escolha de produção)
- Parsers Preditivos:
 - Com **procedimentos recursivos**: são mais adequados para serem escritos **manualmente**; um símbolo look-ahead determina a produção a ser escolhida
 - **Dirigidos por tabela**: fazem uso de uma pilha explícita para guardar o lado direito das produções;
 - mais adequado para serem implementados automaticamente pela pré-computação da Tabela de Análise



A.S.D. com Retrocesso

- Ineficiente
- Foi usada em implementações pioneiras
- Ainda é utilizada em aplicações específicas
- Usada em muitos compilers-compilers com backtracking;

Funcionamento

$\alpha \leftarrow$ cadeia a ser analisada

$D \leftarrow$ árvore formada apenas pelo símbolo reservado S

folha corrente $\leftarrow S$

Repita:

{Seja X a folha corrente}

Se X é não terminal então

[escolha uma produção $X ::= X_1 X_2 \dots X_r$; substitua na árvore D a folha corrente por uma árvore com raiz de rótulo X e descendentes diretos com rótulos

X_1, X_2, \dots, X_r ;

folha corrente $\leftarrow X_1$]

senão {X é terminal}

[Se $\alpha = X\beta$ então

$\alpha \leftarrow \beta$;

folha corrente \leftarrow próxima folha em D no percurso das folhas da esquerda para a direita]

senão { $\alpha = \lambda$ ou 1º símbolo de α não é X}

[retroceder, ou seja, restaurar os valores de D e de folha corrente, antes da última aplicação de produção;

Se existe outra produção

então aplicação da produção

senão retroceder novamente {repete-se até que uma produção seja encontrada}]

até que ($\alpha = \lambda$ e folha corrente aponta para fora da árvore \rightarrow ACEITAR) ou (D=S e não há alternativa \rightarrow ERRO)

Exemplo

Seja a gramática

$$E ::= E + T \mid T$$
$$T ::= T * F \mid F$$
$$F ::= a \mid b \mid (E)$$

→ elimina-se a recursão a esquerda, para evitar que o algoritmo entre numa repetição infinita

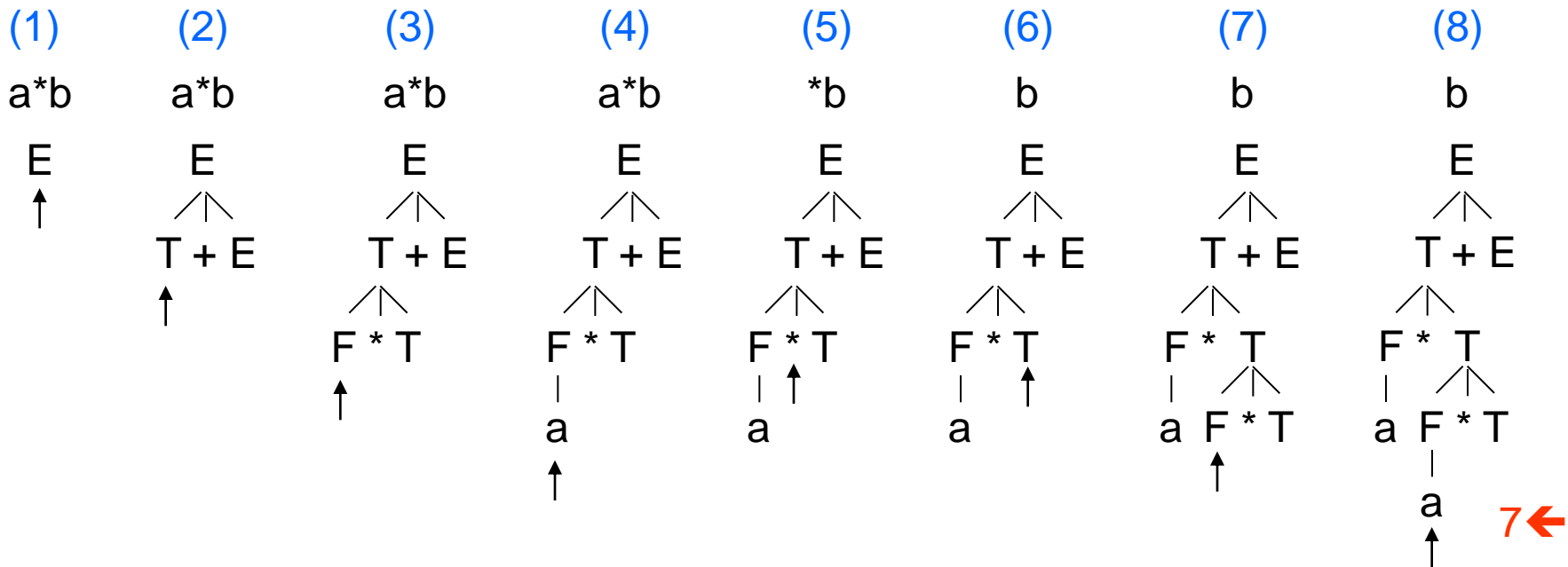
$$E ::= T + E \mid T$$
$$T ::= F * T \mid F$$
$$F ::= a \mid b \mid (E)$$

→ atenção: mudar a recursividade muda o significado atribuído às expressões

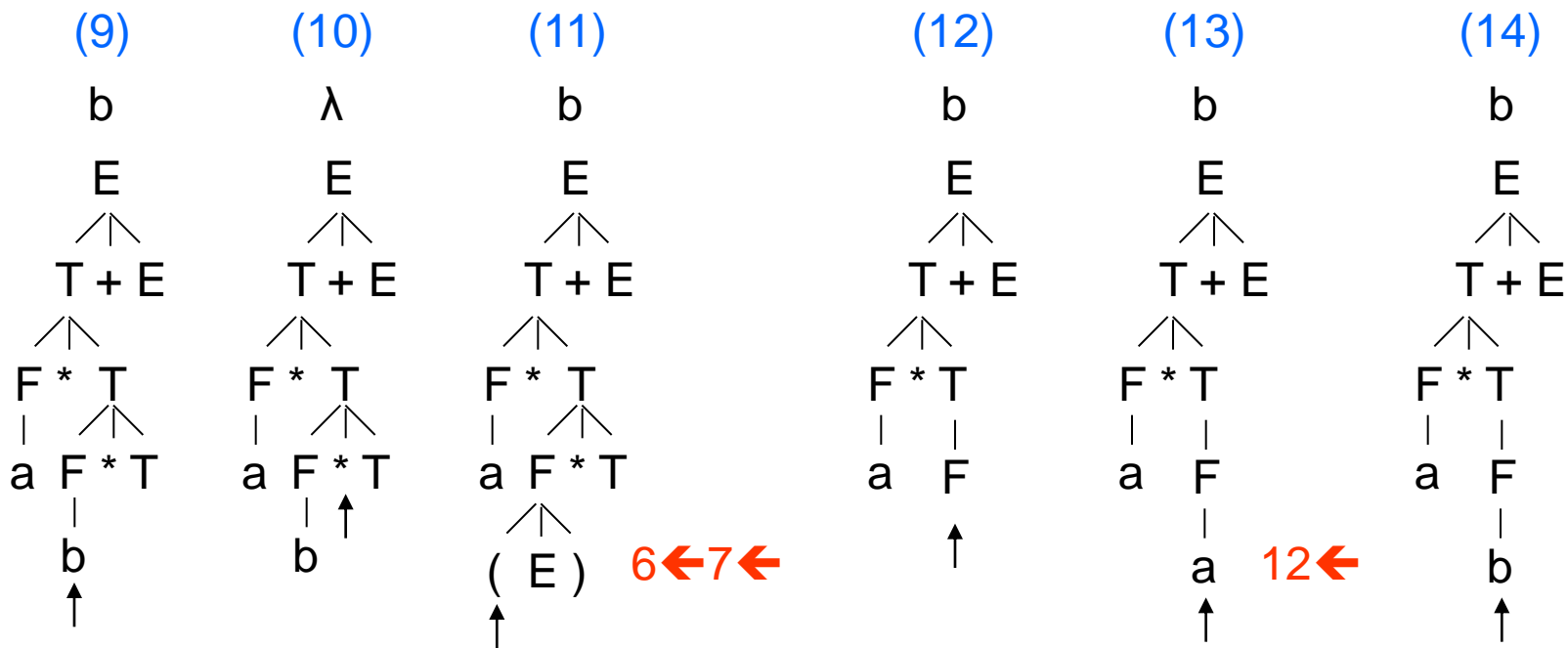
→ a eliminação da recursividade feita não resolveu o problema, pois mudou a associação dos operadores

→ **Análise de $a*b$**

Seja a sentença a^*b



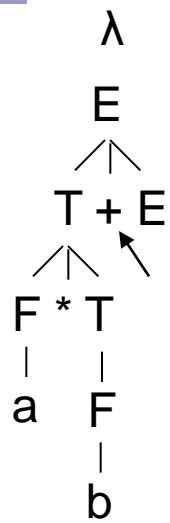
7 ←



6 ← 7 ←

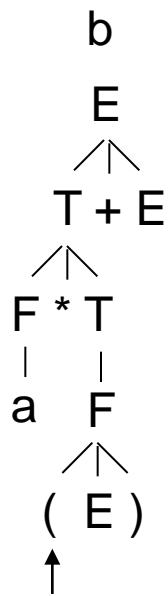
12 ←

(15)



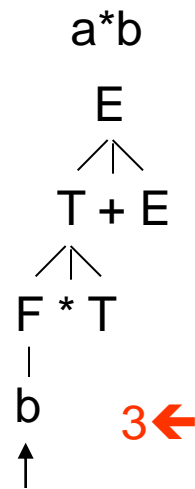
12 ←

(16)



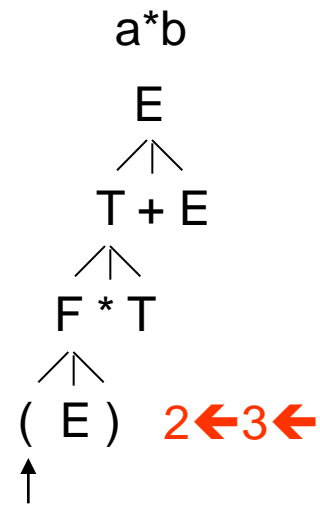
3 ← 6 ← 12 ←

(17)



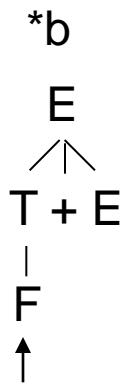
3 ←

(18)

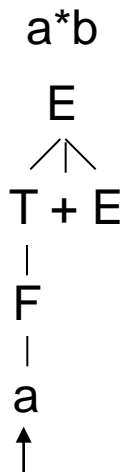


2 ← 3 ←

(19)

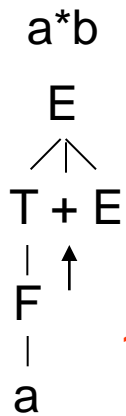


(20)

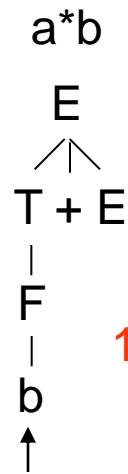


19 ←

(21)

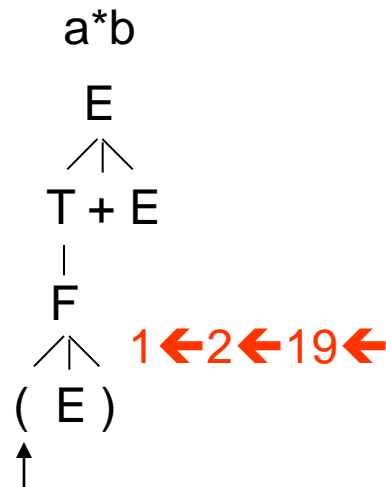


(22)



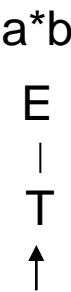
19 ←

(23)

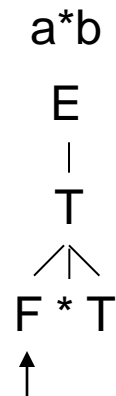


1 ← 2 ← 19 ←

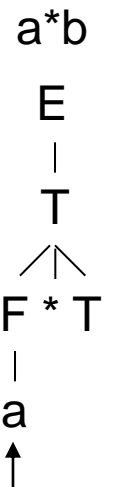
(24)



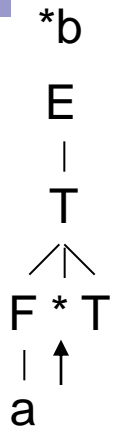
(25)



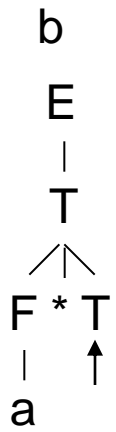
(26)



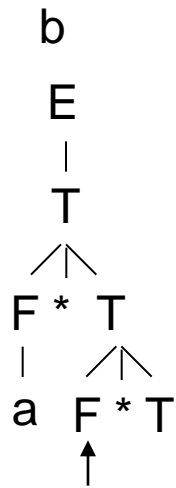
(27)



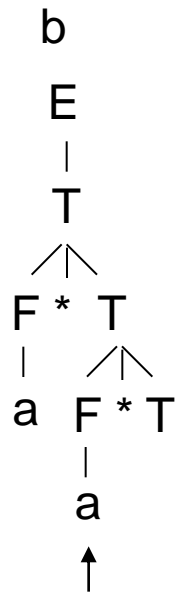
(28)



(29)

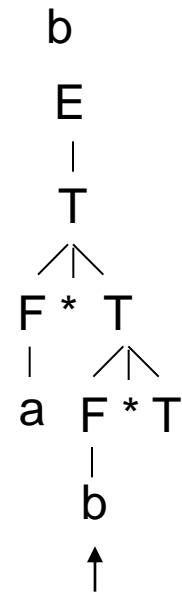


(30)

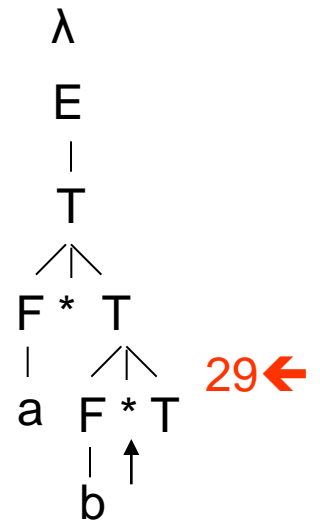


29 ←

(31)

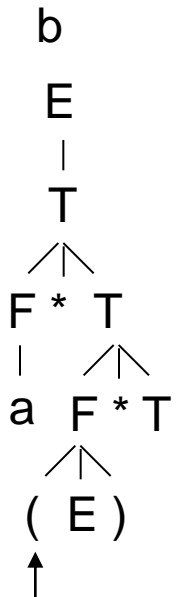


(32)



29 ←

(33)

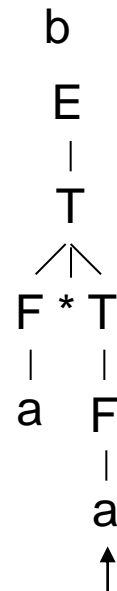


28 ← 29 ←

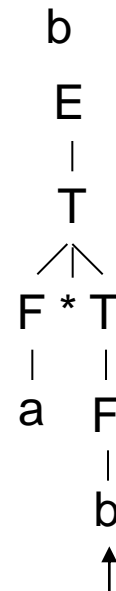
(34)



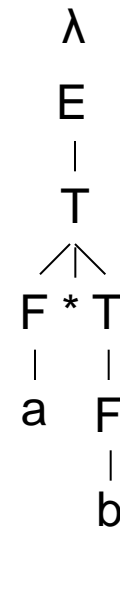
(35)



(36)



(37)



A.S.D. com retrocesso

- A utilização de retrocesso fará com que as ações de modificações da tabela e possível geração de código em compiladores de 1 passo sejam anulados. Geralmente esta não é uma tarefa simples.
- O número de derivações pode ser uma **função exponencial** do tamanho da cadeia
- Este algoritmo caracteriza derivações esquerdas de cadeias (substitui o terminal mais a esquerda)
- A recursividade a esquerda **não** é permitida nos métodos de A.S.D.
- Pela ineficiência e problemas, a análise descendente só é usada quando se **elimina retrocesso**

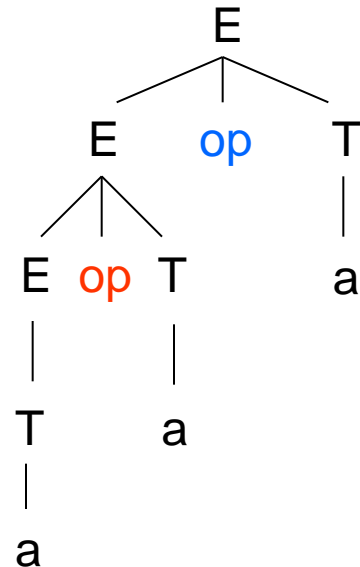
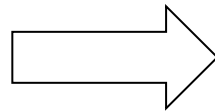
Exemplo

$E ::= E \text{ op } T \mid T$

$T ::= a$

- recursiva a **esquerda**
- tem precedência da esquerda para a direita de seus operadores

a op a op a



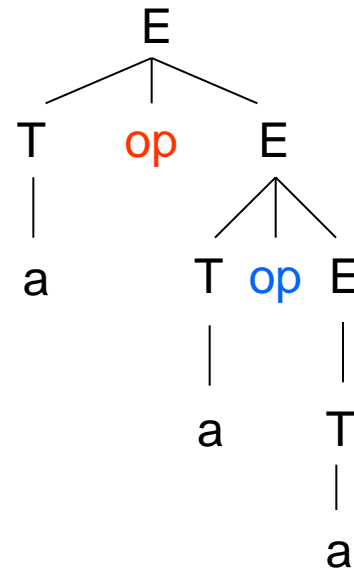
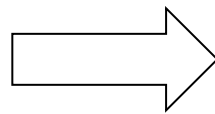
Exemplo

$E ::= T \text{ op } E \mid T$

$T ::= a$

- recursiva a **direita**
- tem precedência da direita para a esquerda de seus operadores

a op a op a



Eliminação de retrocessos e de recursão a esquerda

Para **eliminar retrocessos**:

fazer com que o algoritmo tome a **decisão correta** quanto à produção a ser aplicada.

Uma classe de gramática para as quais isso pode ser feito pode ser descrita por:

1. Toda produção é da forma $A \rightarrow X\alpha$, **X terminal**
2. Se $A \rightarrow X_1\alpha_1 \mid X_2\alpha_2 \mid \dots \mid X_m\alpha_m$, então $X_1 \neq X_2$
 $\neq \dots \neq X_m$

- No algoritmo de análise, a escolha de expansões é resolvida consultando-se o 1º símbolo de entrada

Ex.: Seja a gramática $E \rightarrow a \mid b \mid * EE \mid + EE$ que satisfaz 1 e 2.

Na análise da sentença + a * b a

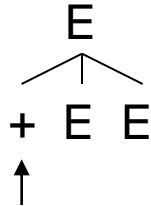
(1)

+ a * b a



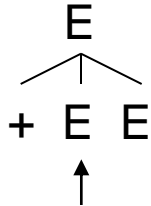
(2)

+ a * b a



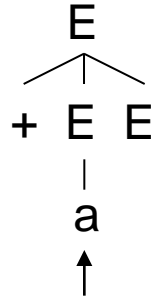
(3)

a * b a



(4)

a * b a



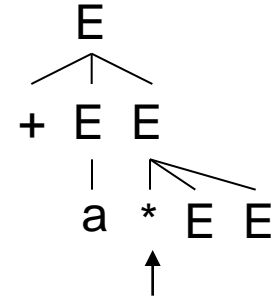
(5)

* b a



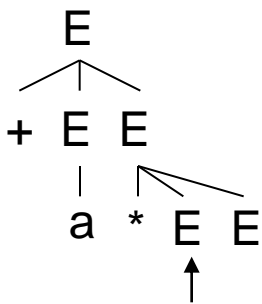
(6)

* b a



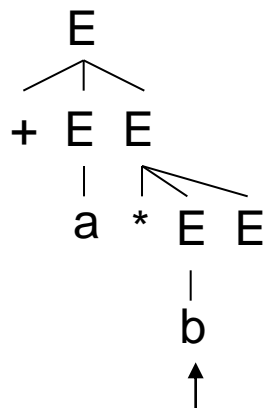
(7)

b a



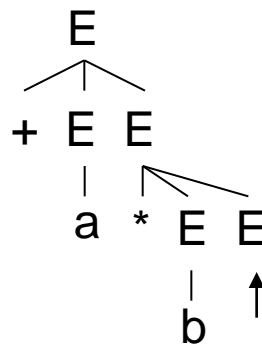
(8)

b a



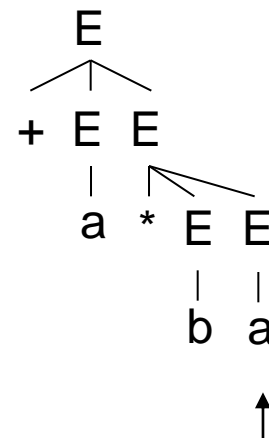
(9)

a



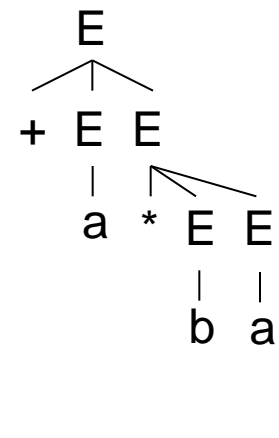
(10)

a



(11)

λ



■ Essas restrições são muito **severas** (é muito difícil encontrar uma gramática que satisfaça essas condições para uma determinada linguagem).

■ Vamos obter uma classe mais ampla de gramáticas.

Seja a relação FIRST

$$\text{First}(X) = \{Y \in T \mid X \xrightarrow{\psi_p^*} Y\} \text{ \{primeiro_símbolo\}}$$

isto é, $\text{First}(X) = \{X\}$ se $X \in T$

$$= \{Y \in T \mid X \Rightarrow^* Y\alpha\} \text{ se } X \in N$$

A restrição passa a ser:

- para todo símbolo $A \in N$ com as regras:

$$A \rightarrow X_1\alpha_1 \mid X_2\alpha_2 \mid \dots \mid X_n\alpha_n$$

onde $X \in V$, temos que:

- $\text{First}(X_i)$ são disjuntos dois a dois:
 - $\text{First}(X_k) \cap \text{First}(X_j) = \{ \}$ para $\forall k, j \in \{1, 2, \dots, n\}$ com $k \neq j$

Agora, o algoritmo de análise deve verificar a qual dos conjuntos First pertence o 1º símbolo da cadeia de entrada α .

Como os conjuntos devem ser disjuntos, haverá no máximo 1 alternativa possível

Seja a gramática:

$S \rightarrow AS \mid BA$

$A \rightarrow aB \mid C$

$B \rightarrow bA \mid d$

$C \rightarrow c$

Gerem o conjunto
dos FIRST para
todos os símbolos

$S \rightarrow AS \mid BA$

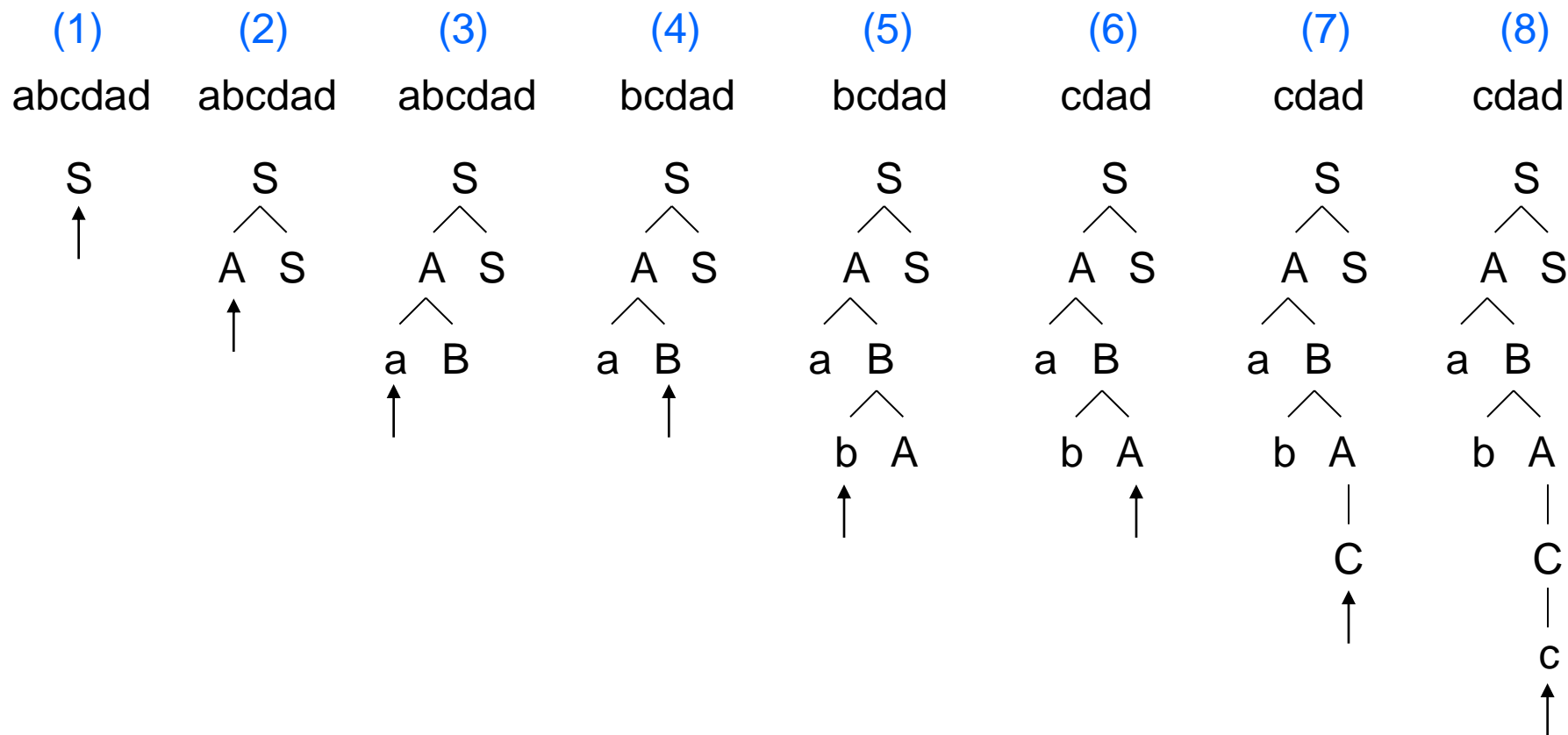
$A \rightarrow aB \mid C$

$B \rightarrow bA \mid d$

$C \rightarrow c$

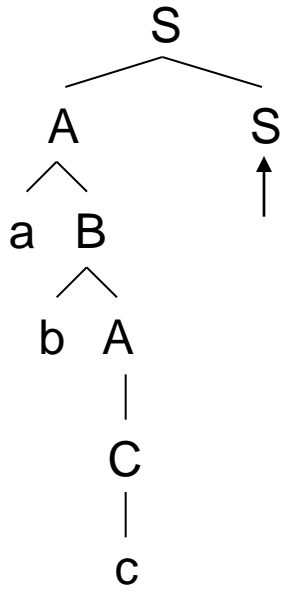
	ψ_p	ψ_p^*	First
S	A, B	A,B,a,C,b,d,c,S	a,b,d,c
A	a, C	a,C,c,A	a,c
B	b, d	b,d,B	b,d
C	c	c,C	c
a	-	-	a
b	-	-	b
c	-	-	c
d	-	-	d

Passos da análise para a sentença abcdad



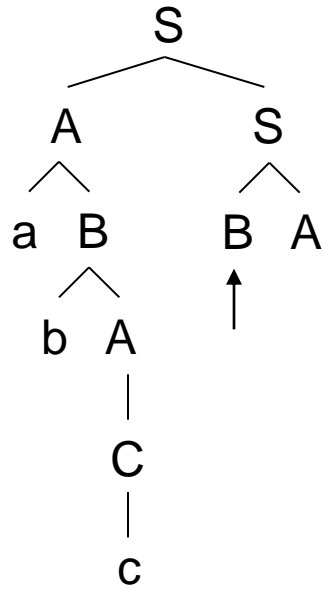
(9)

dad



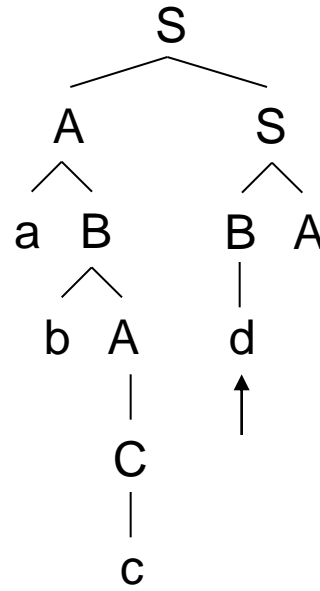
(10)

dad



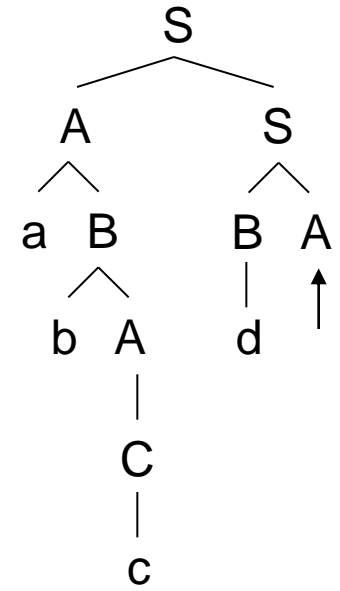
(11)

dad



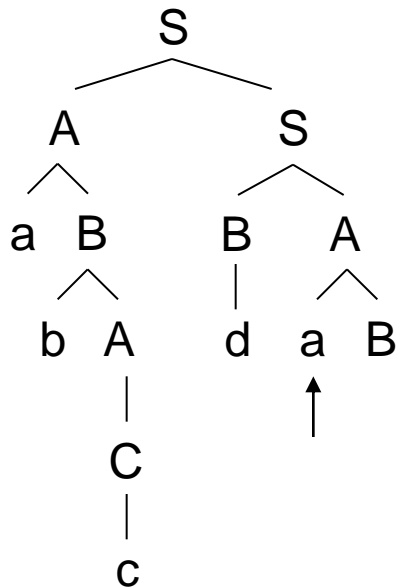
(12)

ad



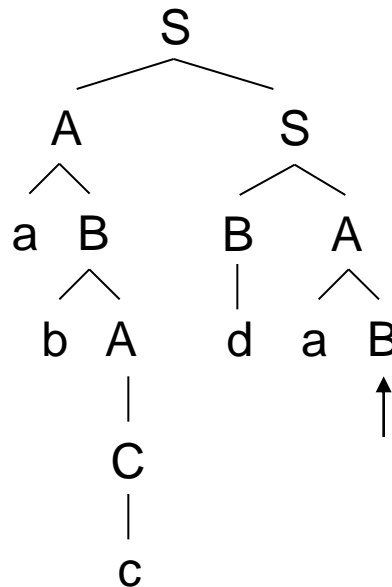
(13)

ad



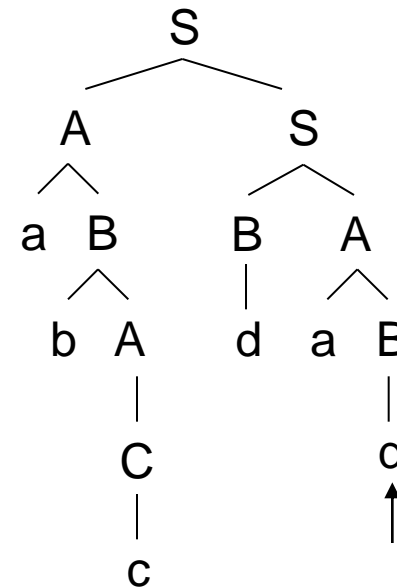
(14)

d



(15)

λ



A classe de gramática que satisfaz esta restrição é chamada LL(1) →

analisa uma cadeia da esquerda para a direita produzindo uma derivação esquerda, verificando apenas **1 símbolo da cadeia de entrada** para decidir qual a produção aplicada.

$S \rightarrow AS \rightarrow ABS \rightarrow abAS \rightarrow abCS \rightarrow abcS \rightarrow abcBA \rightarrow abcdA \rightarrow abcdAB \rightarrow abcdAB \rightarrow abcdad$

Como definido, LL(1) = Left to right, Left-most derivation, 1 símbolo look-ahead

Ex.:

$$S \rightarrow aAB \mid aBA$$
$$A \rightarrow b \mid cS$$
$$B \rightarrow d \mid eS$$

Não é LL(1), mas como A começa com b ou c e B começa com d ou e, a escolha de S pode se basear no segundo caractere – LL(2).

A gramática de expressões abaixo não é LL(1), pois os conjuntos First não são disjuntos dois a dois. Na verdade, não é LL(K).

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

A gramática equivalente abaixo pode resolver o problema:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + E \mid \lambda \\ T &\rightarrow F T' \\ T' &\rightarrow * T \mid \lambda \\ F &\rightarrow a \mid b \mid (E) \end{aligned}$$

Quando uma regra gera λ há uma segunda checagem a ser feita: veremos logo mais

Agora, os conjuntos First são disjuntos dois a dois.

Em $A \rightarrow \alpha \mid \beta$, se $\beta \Rightarrow^* \lambda$,
 First (α) tem que ser diferente do Follow (A)

	FIRST	FOLLOW
A	{ λ , +}	{ $\$, \)$ }
B	{ λ , *}	{ $\$, +, \)$ }
E	{b, a, ($\)$ }	{ $\$, \)$ }
F	{b, a, ($\)$ }	{ $\$, *, +, \)$ }
T	{b, a, ($\)$ }	{ $\$, +, \)$ }

E	\rightarrow	TA
A	\rightarrow	λ
A	\rightarrow	+E
T	\rightarrow	FB
B	\rightarrow	λ
B	\rightarrow	*T
F	\rightarrow	a
F	\rightarrow	b
F	\rightarrow	(E)

Problema: aumento do comprimento das derivações e conseqüente aumento do número de operações para realizar a análise.

Para minimizar esse problema, podemos aplicar outros recursos, isto é, adotar uma **notação estendida para gramática (EBNF)** e modificar o algoritmo de análise.

1) Fatoração

$$E \rightarrow T+E \mid T \quad \rightarrow \quad E \rightarrow T (+E \mid \lambda)$$

De uma forma geral, se:

$$A \rightarrow \beta y_1 \mid \beta y_2 \mid \dots \mid \beta y_n, \text{ com } \beta \neq \epsilon$$

podemos fatorá-la em:

$$A \rightarrow \beta (y_1 \mid y_2 \mid \dots \mid y_n)$$

Dessa forma, **a escolha da alternativa pode ser retardada.**

Ex.:

$$E \rightarrow T (+E \mid \lambda)$$

$$T \rightarrow F (*T \mid \lambda)$$

$$F \rightarrow a \mid b \mid (E)$$

2) Substituição da notação recursiva por uma notação iterativa

$$E \rightarrow E + T \mid T$$

por

$$E \rightarrow T \{+ T\}$$

3) Combinação de Fatoração e Substituição

$$E \rightarrow E + T \mid E - T \mid T$$

fatora: $E \rightarrow E (+ T \mid - T) \mid T$

substitui: $E \rightarrow T \{ +T \mid - T \}$ ou $E \rightarrow T \{ (+|-) T \}$