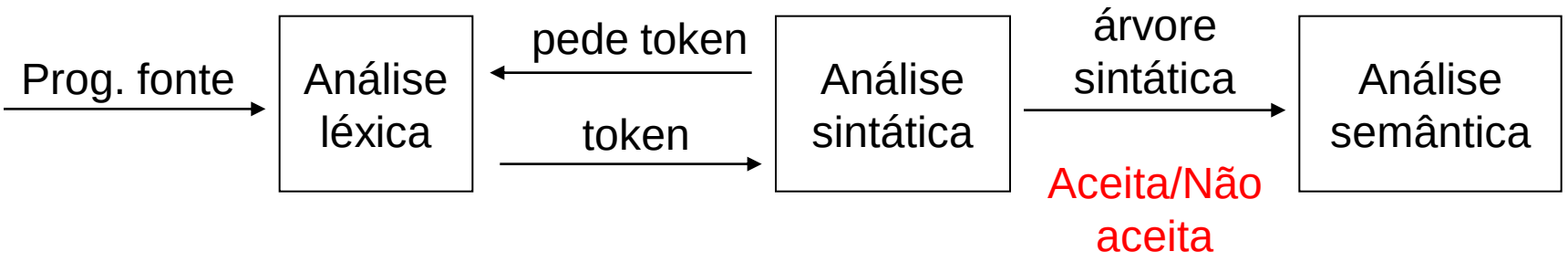
Análise Sintática I: Analisadores Descendentes com Retrocesso

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 $O(n^3)$;

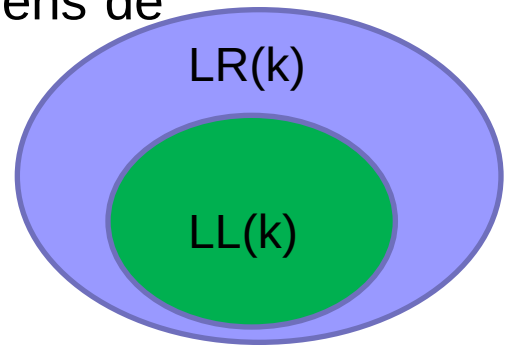
  - mas se elas forem não ambíguas, o tempo é $O(n^2)$ e chega a ser O(n) para muitas linguagens.

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, por exemplo, as gramáticas LL(k) e LR(k), que 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 tipos de A.S.D. e suas vantagens e desvantagens:

- A.S.D. Recursiva com Retrocesso/Backtracking (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
    - JavaCC, no entanto, cai neste tipo, mesmo sendo um compiler compiler. Ele também permite que certas regras possam ser analisadas com um K maior que 1
  - Dirigidos por tabela/Tabular: 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

α ← cadeia a ser analisada

D ← árvore formada apenas pelo símbolo reservado S

folha corrente ← S

Repita:

{Seja X a folha corrente}

Se X é não terminal então

[escolha uma produção $X::=X_1X_2...X_r$; substitua na árvore D a folha corrente por uma árvore com raiz de rótulo X e descendentes diretos com rótulos $X_1, X_2, ... X_r$;

folha corrente ← X_1]

senão {X é terminal}

[Se α=Xβ então

α←β;

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

senão {α=λ 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 (α=λ e folha corrente aponta para fora da árvore -> ACEITAR) ou (D=S e não há alternativa -> ERRO)

Código pode refletir a gramática

- Em Price & Toscani (2001), pgs. 40 e 41, vemos uma implementação de um ASD recursivo com retrocesso:
  - Cada não terminal é uma função para tratar o trecho de código da gramática

- Price, A.M.A. e Toscani, S.S. (2001). Implementação de Linguagens de Programação: Compilador. Editora Sagra Luzzatto.

Exemplo de ASD com retrocesso em código genérico

Seja a gramática

E::=E+T | T

T::=T*F | F

F::= a | b | (E)

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

  E::=T+E | T

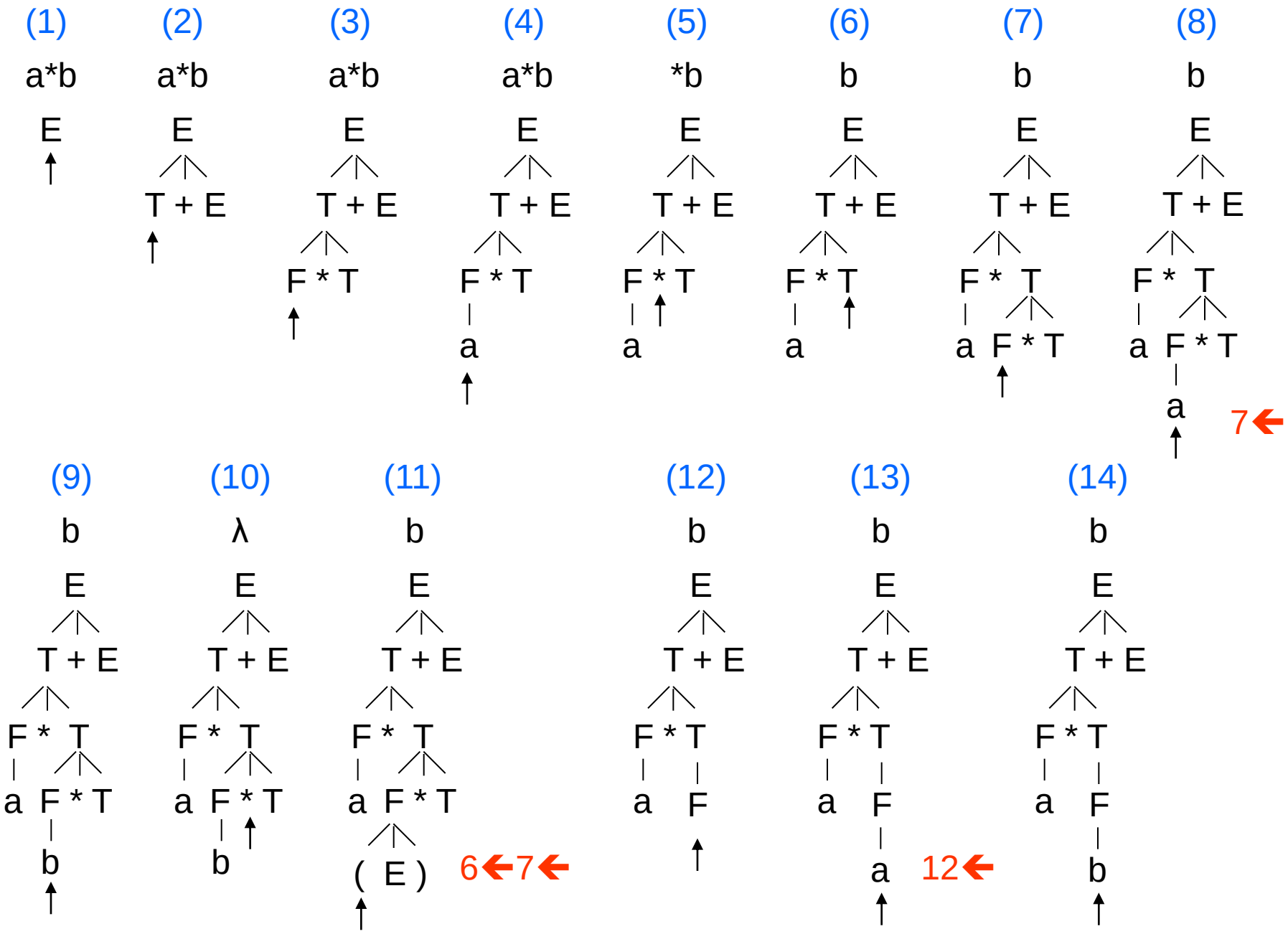
  T::=F*T | F

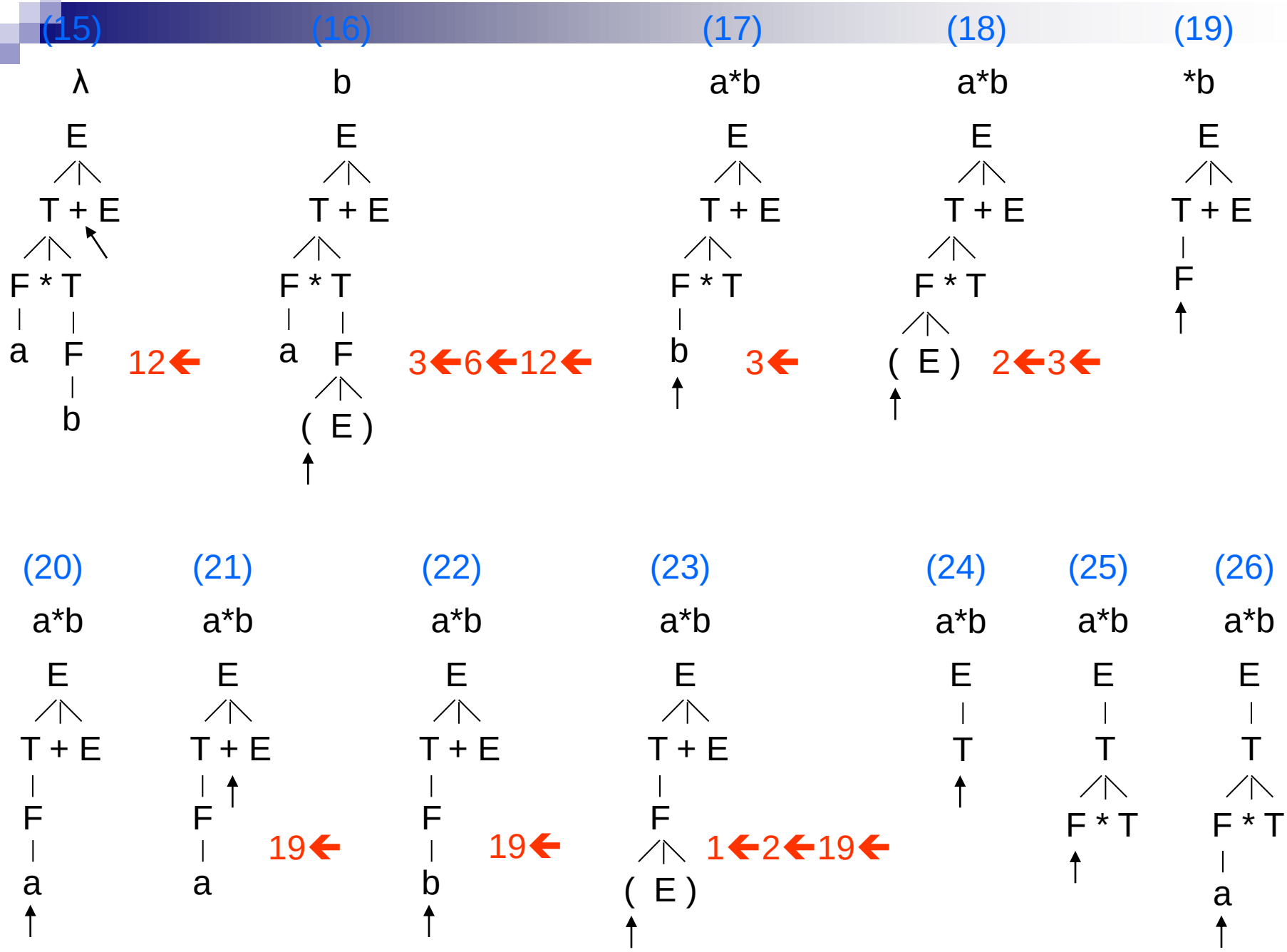
  F::=a | b | (E)

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

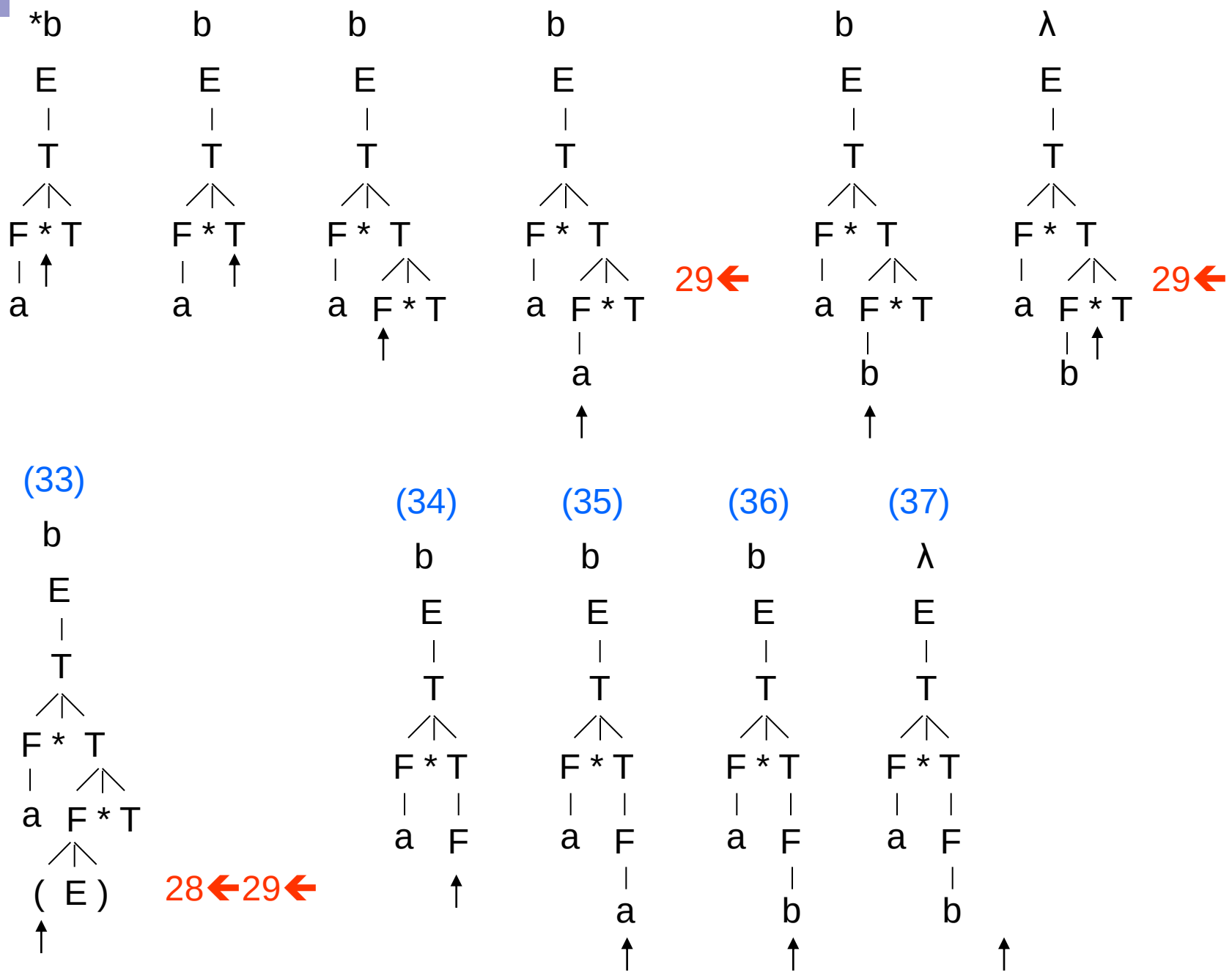
  - a eliminação da recursividade feita não resolveu o problema, pois mudou a associação dos operadores
  - Análise de a+b

Seja a sentença a*b




(15)
λ
E
T + E
F * T
a
F
b
12
(16)
b
E
T + E
F * T
a
F
(E)
3
6
12
(17)
a*b
E
T + E
F * T
b
3
(18)
a*b
E
T + E
F * T
(E)
2
3
(19)
*b
E
T + E
F
(20)
a*b
E
T + E
F
a
(21)
a*b
E
T + E
F
a
19
(22)
a*b
E
T + E
F
b
19
(23)
a*b
E
T + E
F
(E)
1
2
19
(24)
a*b
E
T
(25)
a*b
E
T
F * T
(26)
a*b
E
T
F * T
a

(27) (28) (29) (30) (31) (32)



A.S.D. com retrocesso

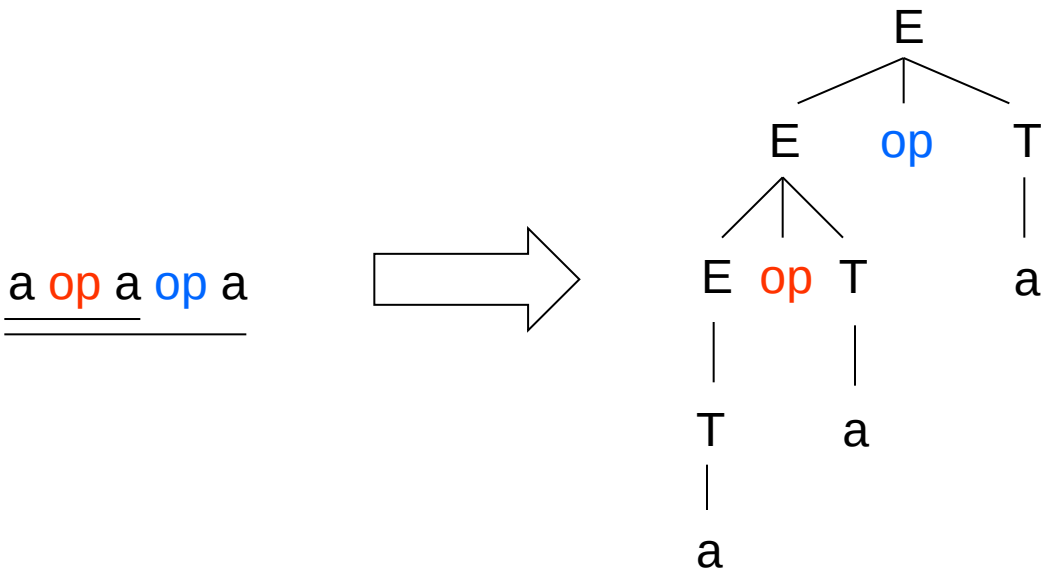
- A utilização de retrocesso fará com que as ações de modificações da tabela de símbolos e possível geração de código em compiladores de 1 passo sejam anulados. Geralmente esta não é uma tarefa simples.
- O número de derivações pode ser uma função exponencial do tamanho da cadeia
- Este algoritmo caracteriza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

Exemplo

E ::= E op T | T

T ::= a

- recursiva a esquerda
- Associa /resolve os operadores da mesma classe da esquerda para a direita

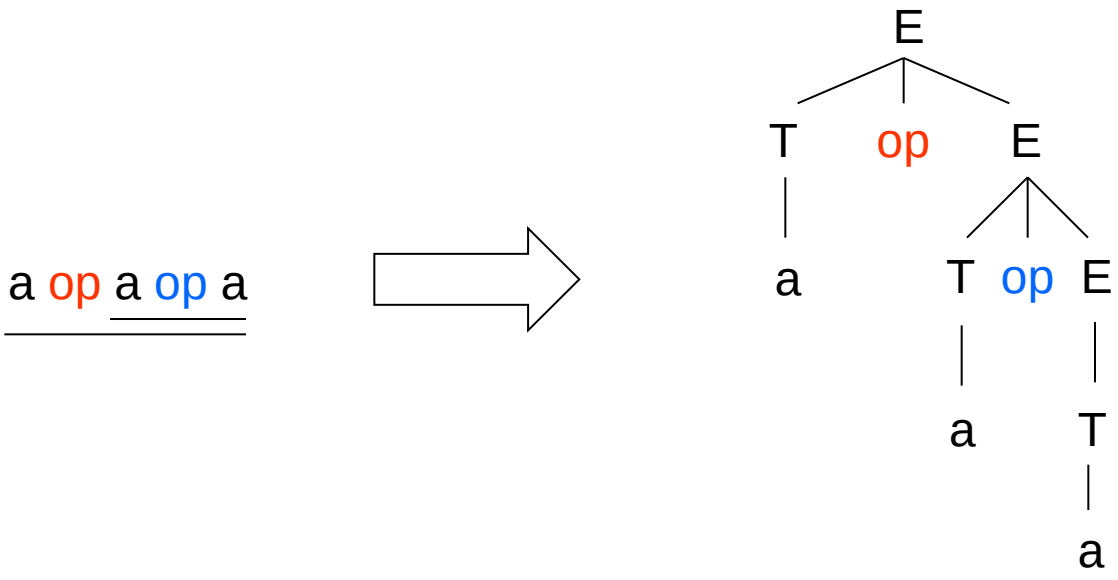


Exemplo

E ::= T op E | T

T ::= a

- recursiva a direita
- Associa /resolve os operadores da mesma classe da direita para a esquerda



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 $A \rightarrow X_1\alpha_1 \mid X_2\alpha_2 \mid \ldots \mid X_m\alpha_m$, então $X_1 \neq X_2 \neq \ldots \neq X_m$

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

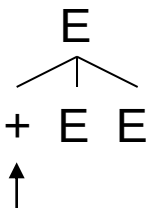
Ex.: Seja a gramática E → a | b | * EE | + EE

que satisfaz 1 e 2.

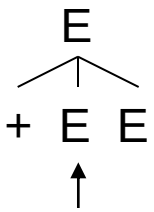
Na análise da sentença + a * b a

(1) + a * b a

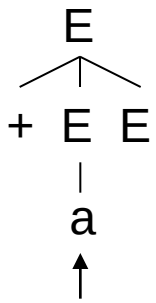
(2) + a * b a



(3) a * b a

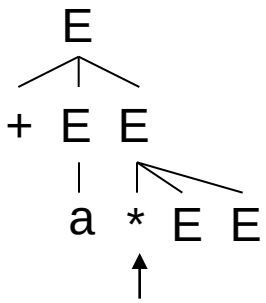


(4) a * b a

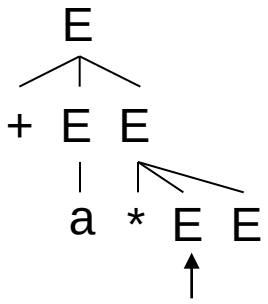


(5) * b a

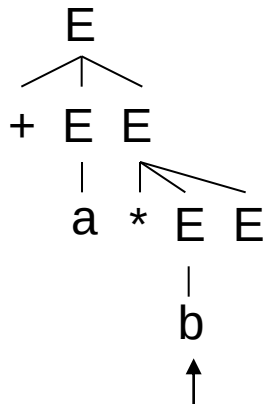
(6) * b a



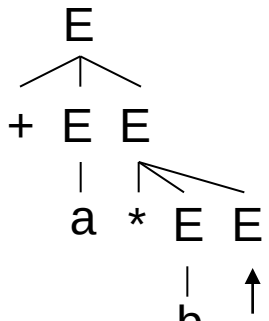
(7) b a



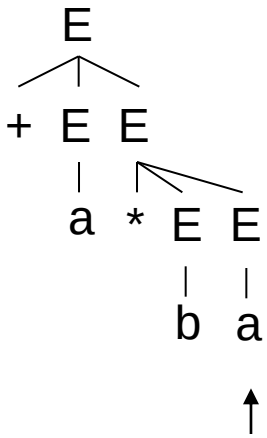
(8) b a



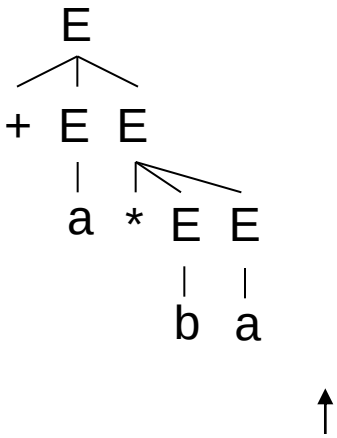
(9) a



(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

$$First(X) = \{Y \in T \mid X \ \psi^*_p \ Y \}$$ {primeiro_símbolo}

isto é, First(X) = {X} se $X \in T$

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

A restrição passa a ser:

- para todo símbolo A ϵ N com as regras:

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

onde X ϵ V, temos que:

- First(X_i) são disjuntos dois a dois:
  - $First(X_k) \cap First(X_j) = \{ \}$ para $\forall$ k,j ϵ {1, 2...n} com k≠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 → AS | BA

A → aB | C

B → bA | d

C → c

Gerem o conjunto dos FIRST para todos os símbolos

S → AS | BA

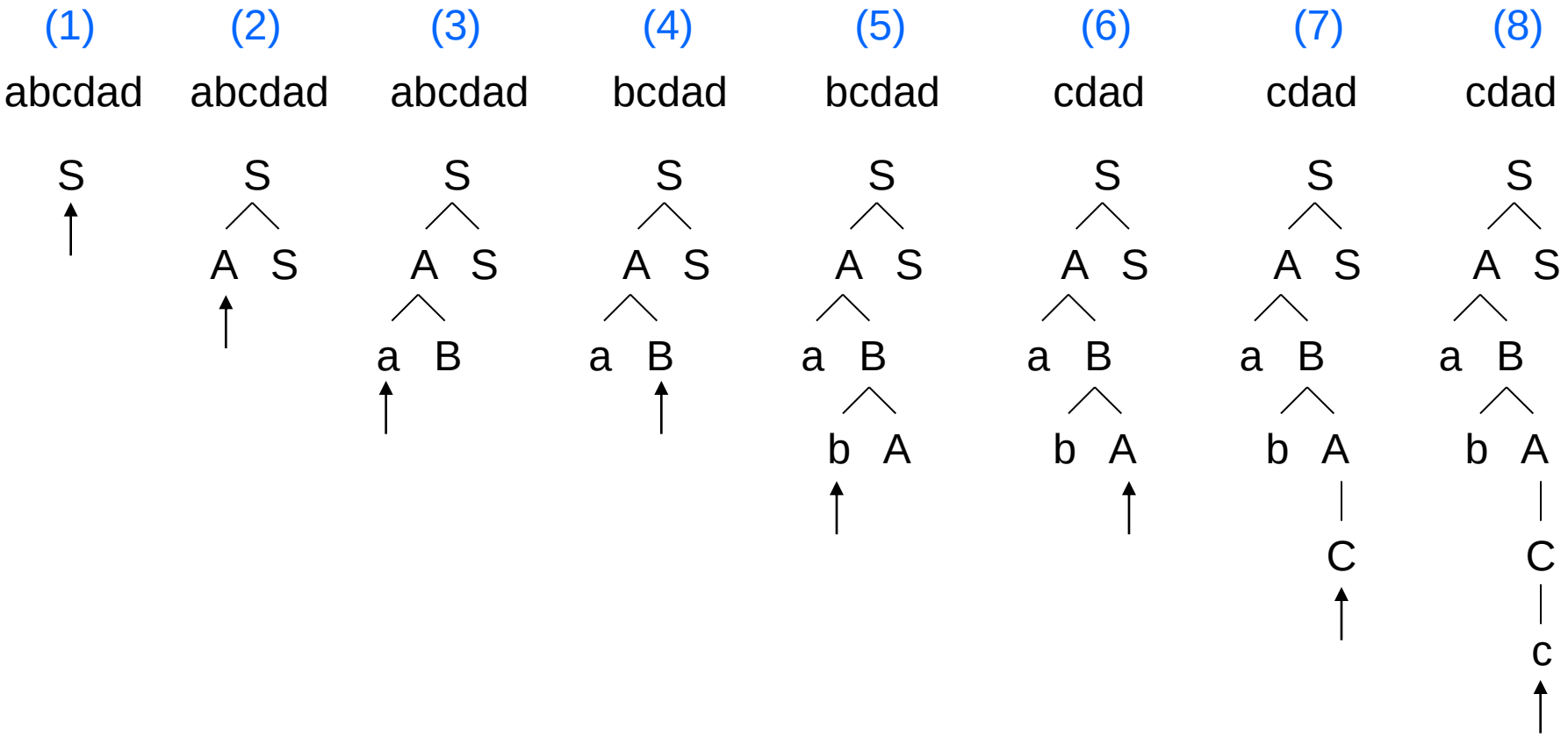
A → aB | C

B → bA | d

C → c

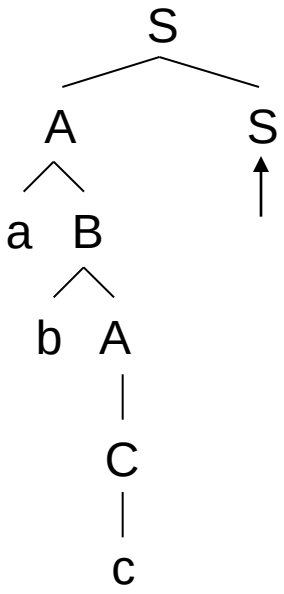
	ψ_p	ψ^*_p	First
S	A, B	A,B,a,C,b,d,c,S	a,b,d,c
A	a, C	a,C,c,A	a,c
B	b, d	b,d,B	b,d
C	c	c,C	c
a	-	-	a
b	-	-	b
c	-	-	c
d	-	-	d

Passos da análise para a sentença abcdad



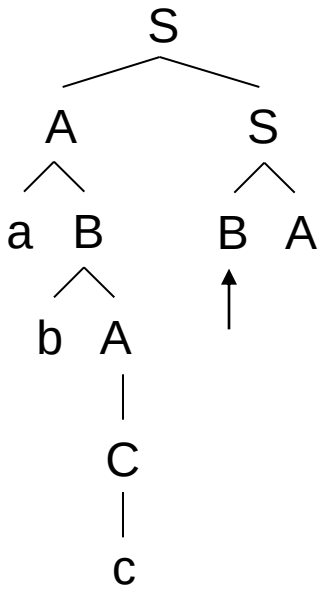
(9)

dad



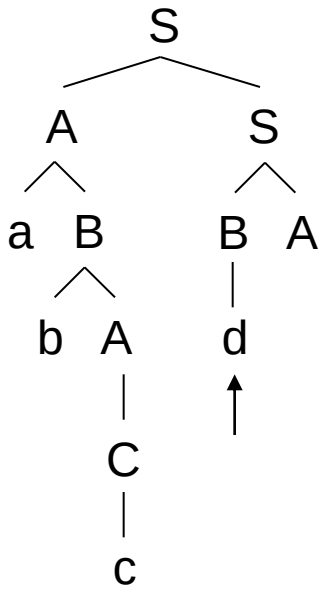
(10)

dad



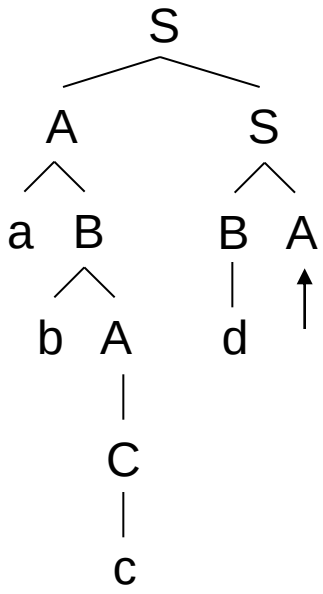
(11)

dad



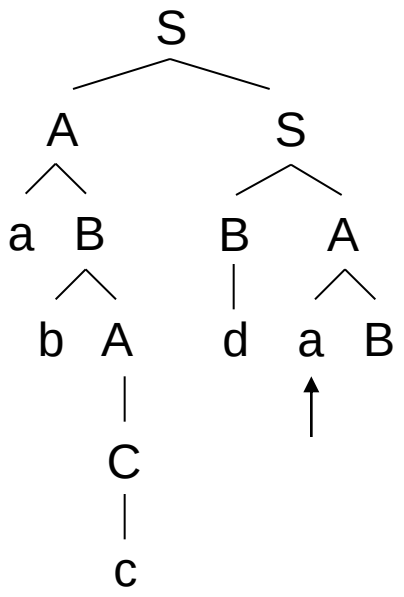
(12)

ad



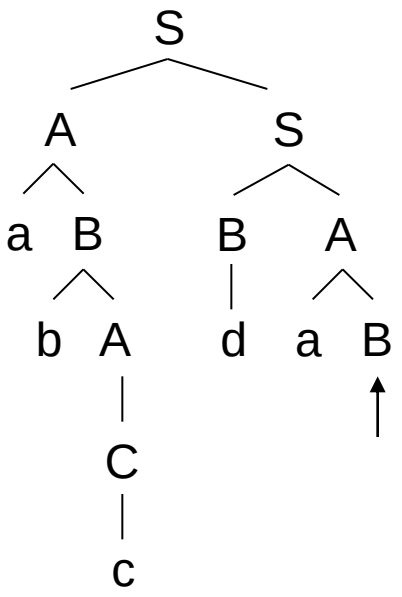
(13)

ad



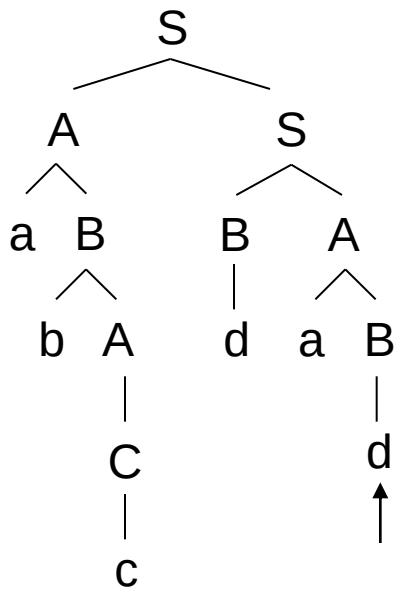
(14)

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 → AS → ABS → abAS → abCS → abcS→ abcBA → abcdA → abcdAB → abcdaB → abcdad

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

Ex.:

S → aAB | aBA

A → b | cS

B → d | 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).

E → T + E | T
T → F * T | F
F → a | b | (E)

A gramática equivalente abaixo pode resolver o problema:

E → T E’
E’ → + E | λ
T → F T’
T’ → * T | λ
F → a | b | (E)

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

Agora, os conjuntos First são disjuntos dois a dois.

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:

$$A \rightarrow \beta y_1 \mid \beta y_2 \mid \ldots \mid \beta y_n, \text{ com } \beta \neq y$$

podemos fatorá-la em:

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

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 $E \rightarrow T \{(+|-) T\}$