



COMPILADORES

Aula 9 – Análise Sintática

Curso de Ciência da Computação

Prof. Dr. Rodrigo Xavier de Almeida Leão

É importante lembrar:

A **sintaxe** de uma linguagem define as estruturas básicas da linguagem.

As **estruturas básicas**, por sua vez, são compostas pelas regras léxicas e sintáticas, que são definidas pela gramática.

As **regras léxicas** são definidas pelas gramáticas regulares (GR).

As **regras sintáticas** são definidas pelas gramáticas livres de contexto (GLC).

Segundo a hierarquia de Chomsky, $LR \subset LLC$, isto significa que toda linguagem regular (LR) é livre de contexto (LLC), mas nem toda LLC é LR. Logo, GR define LR, e GLC define LLC.

1. Definição Formal:

Linguagens Regulares: São definidas por expressões regulares ou autômatos finitos (como autômatos finitos determinísticos ou não-determinísticos).

Linguagens Livres de Contexto: São definidas por gramáticas livres de contexto.

2. Expressividade:

Linguagens Regulares: São menos expressivas que as linguagens livres de contexto. Elas podem representar padrões simples, como sequências repetidas de caracteres, mas não podem lidar com construções mais complexas que envolvam aninhamento ou contagem de elementos.

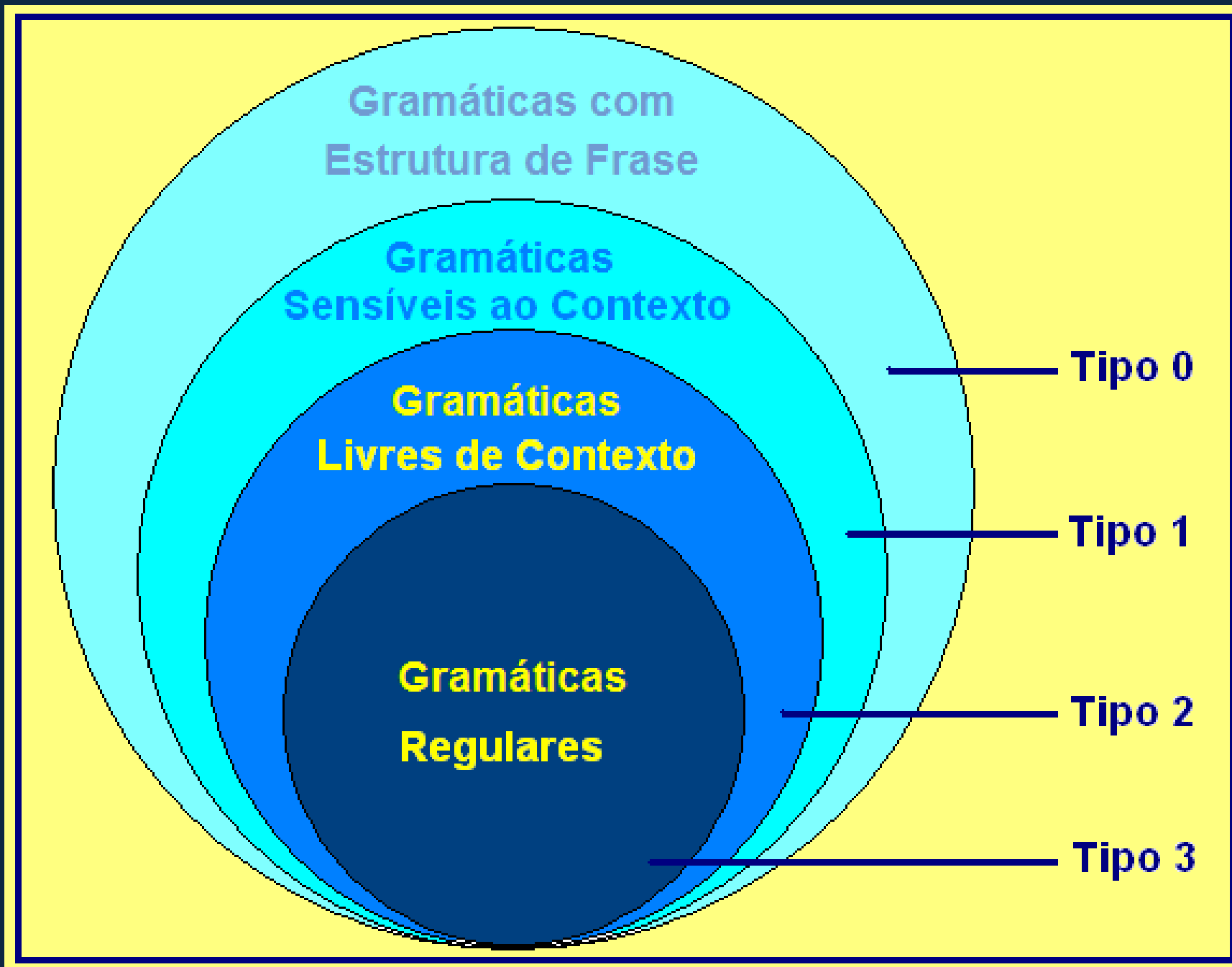
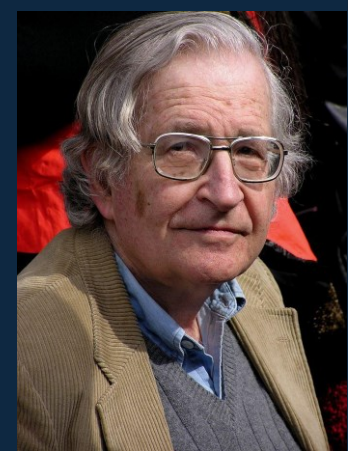
Linguagens Livres de Contexto: São mais expressivas e podem representar uma ampla variedade de estruturas, incluindo aninhamento e recursão.

Linguagens Regulares: São frequentemente utilizadas em linguagens de programação para análise léxica (tokenização), em expressões regulares para busca e manipulação de texto, em processamento de linguagem natural para identificação de padrões simples, entre outros.

Linguagens Livres de Contexto: São utilizadas em linguagens de programação para análise sintática (parsing) e na descrição de estruturas mais complexas, como a sintaxe de uma linguagem de programação.

Hierarquia de Chomsky:

As linguagens regulares são o nível mais baixo na hierarquia de Chomsky, enquanto as linguagens livres de contexto estão em um nível acima delas.



Seja uma estrutura simples, que represente uma expressão matemática, do tipo $\mathbf{a + b * c}$. O nosso alfabeto será $\Sigma = \{a, b, c, +, *\}$ e a gramática no padrão EBNF, a qual iremos denominar de **L**, será:

$$\langle \text{expr} \rangle ::= \underline{\langle \text{expr} \rangle} \langle \text{op} \rangle \langle \text{expr} \rangle \mid \langle \text{id} \rangle$$
$$\langle \text{id} \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$$
$$\langle \text{op} \rangle ::= \mathbf{+} \mid \mathbf{*}$$

Seja ω uma palavra (sentença) que pertença à linguagem **L** definida, então $\omega \in \mathbf{L}$.

Seja $\omega_1 = \mathbf{a + b}$ e desejamos verificar se $\omega_1 \in \mathbf{L}$.

Uma forma de verificar isso é por meio do processo de derivação, que consiste em substituir a sentença, ou parte dela, no lado direito da produção da gramática repetidas vezes, até alcançar os símbolos terminais, e, segundo Aho (2007), podemos aplicar repetidamente as produções em qualquer ordem, a fim de obtermos uma sequência de substituições. Esse é o processo de derivação.

Para ajudar a compreensão do processo de derivação, vamos substituir por letra **MAIÚSCULA** cada **símbolo não-terminal** da gramática, e adotaremos para os **símbolos terminais**, aqueles que pertencem ao alfabeto, neste caso " Σ ", a grafia em **negrito** para diferenciá-los dos demais. A seta " \rightarrow " indicará a definição da produção, equivalente à notação " $::=$ " no padrão EBNF, e " \Rightarrow " indicará **uma** derivação. Assim, de acordo com essas convenções, antes de aplicarmos o processo de derivação à gramática no padrão EBNF, a qual estamos analisando, vamos reescrever a gramática equivalente de acordo com os padrões convencionados aqui, assim teremos:

Gramática padrão EBNF

$\langle expr \rangle ::= \langle \underline{expr} \rangle \langle op \rangle \langle expr \rangle$

$\langle id \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

$\langle op \rangle ::= + \mid *$

Gramática equivalente

$E \rightarrow E \text{ OP } E \mid \text{ID}$

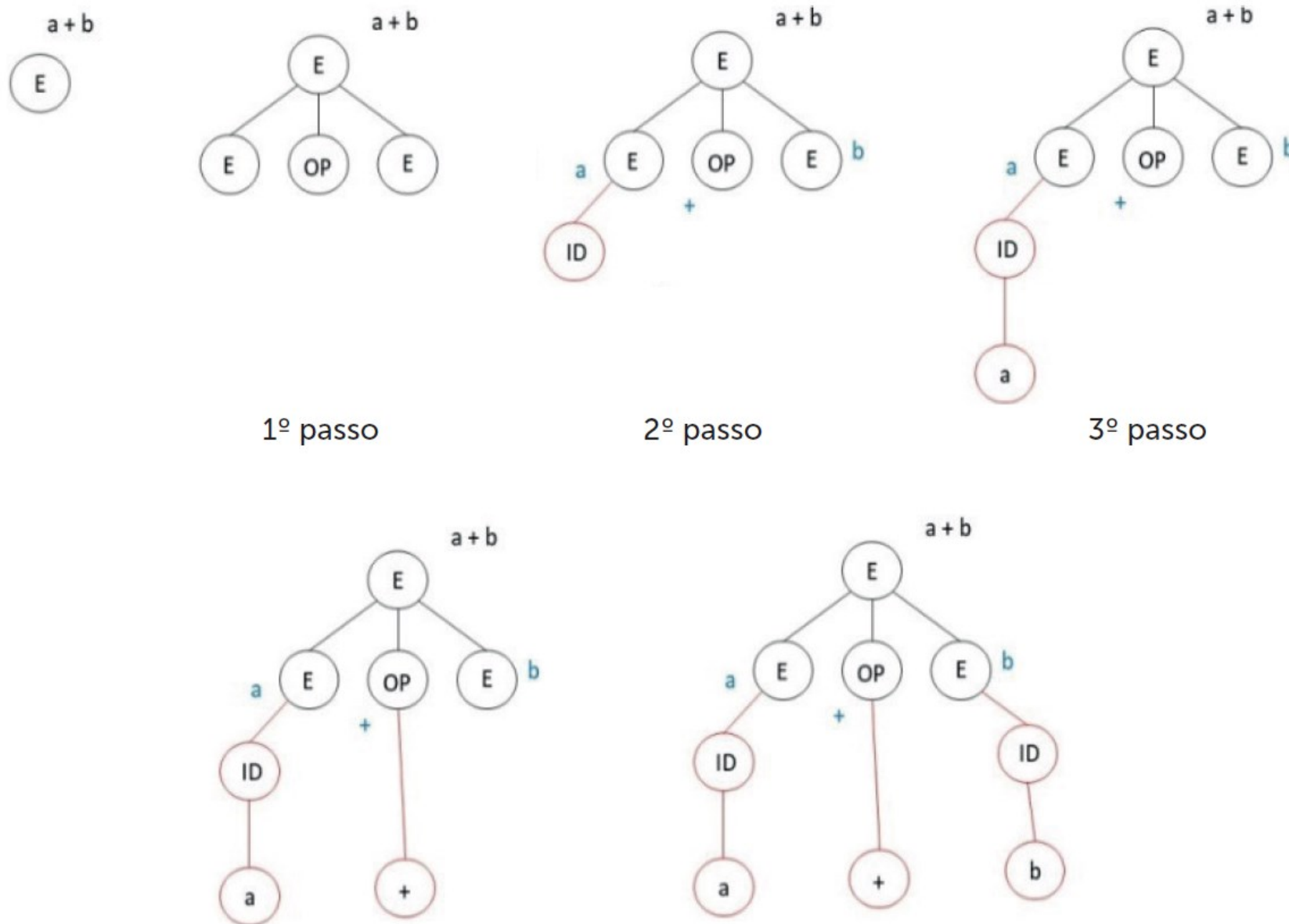
$\text{ID} \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

$\text{OP} \rightarrow + \mid *$

$\Rightarrow E \rightarrow E \text{ OP } E$	1º passo deriva a regra mais à esquerda, pois a
$\Rightarrow E \rightarrow \text{ID} \text{ OP } E$	cadeia ω_1 tinha um OP
$\Rightarrow E \rightarrow \text{a} + E$	2º passo derivou o E <u>mais à esquerda</u>
	3º passo derivou o ID . Chegamos ao símbolo terminal a
$\Rightarrow E \rightarrow \text{a} \text{ OP } E$	4º passo derivou OP .
$\Rightarrow E \rightarrow \text{a} + E$	5º passo derivou o ID . Chegamos ao símbolo terminal +
$\Rightarrow E \rightarrow \text{a} + \text{ID}$	6º passo derivou o E
$\Rightarrow E \rightarrow \text{a} + \text{b}$	7º passo derivou ID . Chegamos ao símbolo terminal b .

Portanto, podemos afirmar que a sentença "a+b" é uma cadeia gerada pela linguagem L, pois, ao fazermos as derivações sucessivas, segundo a gramática **L**, chegamos a uma cadeia de símbolos terminais, logo se $\omega_1 = \text{a} + \text{b}$ então $\omega_1 \in \text{L}$.

Figura 3.2 | Passo a passo da árvore de derivação para a sentença "a+b"



A linguagem L é definida pela **Gramática**

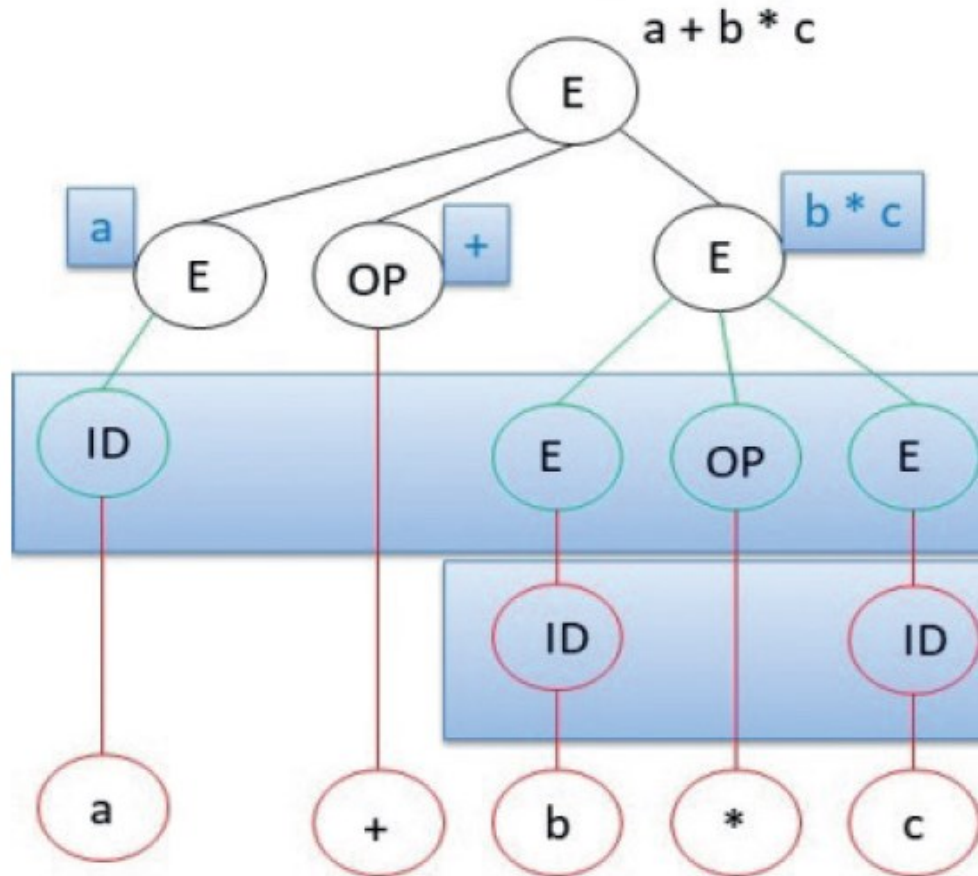
$E \rightarrow E \text{ OP } E \mid \text{ID}$

$\text{ID} \rightarrow a \mid b \mid c$

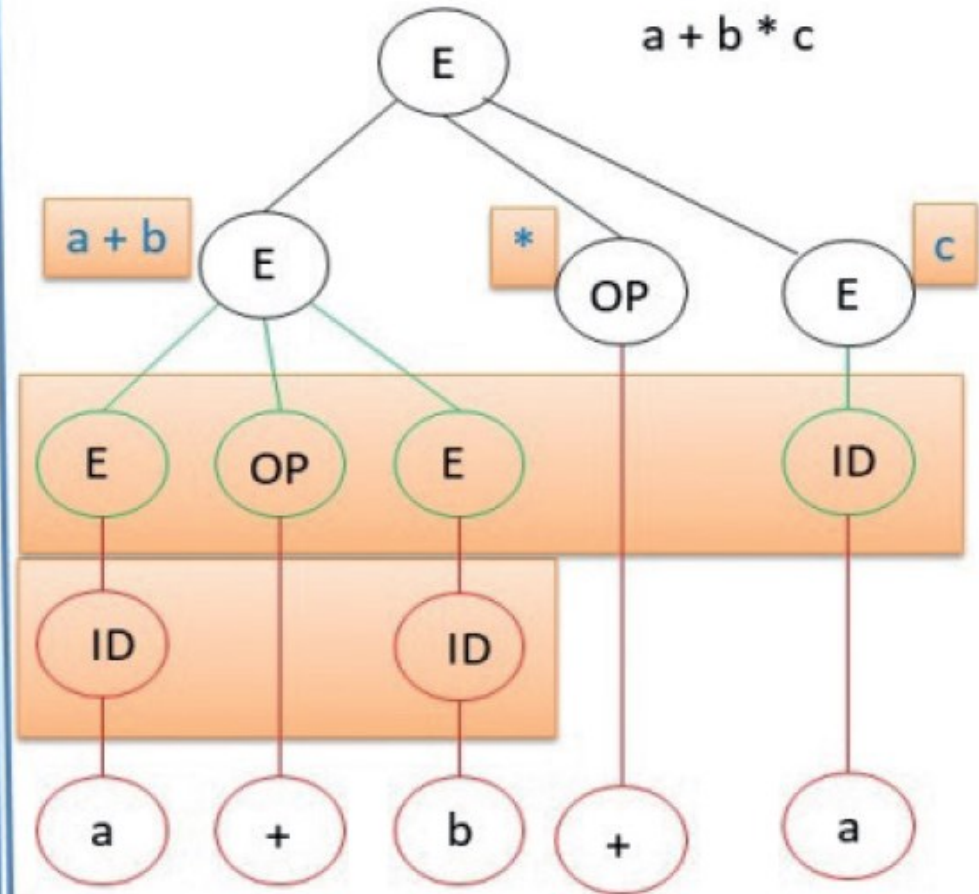
$\text{OP} \rightarrow + \mid *$

A gramática é ambígua, pois para sentença "a+b*c" temos duas árvores distintas!

DERIVAÇÃO MAIS A ESQUERDA



DERIVAÇÃO MAIS A DIREITA



eliminar a ambiguidade, devemos reescrever a gramática, tratando o problema de ambiguidade dos operadores, e, neste caso, construir mais produções para tratar o operador de adição e o de multiplicação distintamente, assim, a nova gramática sem ambiguidade será:

$$E \rightarrow E + F \mid F$$
$$F \rightarrow F * ID \mid ID$$
$$ID \rightarrow a \mid b \mid c$$

Como citado na Seção 2.3, há dois tipos de analisadores sintáticos: os **ascendentes**, que constroem a árvore gramatical das folhas para a raiz, chamados também de ***bottom-up***, ou seja, de baixo para cima. E os **descendentes**, que analisam a árvore gramatical da raiz para as folhas, isto é, de cima para baixo, também chamados de ***top-down***. As árvores gramaticais desenvolvidas nos exemplos anteriores foram todas descendentes, pois começamos pela produção inicial (nó inicial) e derivamos cada nó até chegarmos aos símbolos terminais (as folhas). Esse é o método mais intuitivo e usual e consiste em construir a árvore gramatical em pré-ordem (raiz-esquerda-direita), como você pôde acompanhar no processo de construção das árvores apresentadas.

Os analisadores descendentes utilizam um modelo de algoritmo preditivo recursivo, que pode ser com retrocesso ou sem retrocesso. Por que preditivo? É fácil entender. Para isso, vamos considerar a produção:

$$E \rightarrow E + E \mid E * E \mid a$$

O algoritmo que irá analisar a sentença de entrada precisará avançar na leitura, além do primeiro *token*, para saber se escolhe a primeira opção, " $E + E$ ", a segunda, " $E * E$ ", ou, ainda, a terceira, caso haja apenas o *token* ' a ', apesar de estar analisando apenas o primeiro *token*. Ou seja, um algoritmo preditivo é aquele que avança para ser capaz de tomar uma decisão agora, com base em uma informação que será inserida no futuro, mas que se conhece previamente.

E o que significa com retrocesso ou sem? Um algoritmo de análise sintática **com retrocesso** faz a escolha da primeira opção

análise sintática **com retrocesso** faz a escolha da primeira opção

e avança na construção da árvore gramatical, armazenando os dados em uma pilha, até alcançar o final, mas se não reconhecer a sentença, retrocede, desempilhando os dados até o nó inicial e reinicia o processo. Esse algoritmo é ineficiente do ponto de vista do tempo gasto na análise (o vai e volta), do tratamento da recuperação de erros, e, também, para da análise semântica. Mas, e o algoritmo de análise sintática sem retrocesso, como funciona afinal?

Como o próprio nome diz, o objetivo desse algoritmo é não retroceder, isto é, não voltar. Para que seja possível utilizar um algoritmo de **análise sintática recursivo preditivo sem retrocesso** há uma condição: a gramática não pode ter recursão a esquerda. Assim, basta conhecer apenas o primeiro o *token* de entrada e a produção à frente para expandir a árvore. Esse tipo de analisador é capaz de derivar a maioria das linguagens de programação, por serem do tipo livre de contexto.


A recursividade em gramáticas ocorre quando uma regra de produção de uma gramática se refere direta ou indiretamente ao próprio não-terminal que está sendo definido. Existem dois tipos de recursividade em gramáticas: recursão à esquerda e recursão à direita.

1. Recursão à Esquerda:

Na recursão à esquerda, um não-terminal é definido em termos dele mesmo e de outros símbolos à sua direita.

Por exemplo, considere a seguinte gramática:

go

 Copiar código

```
Expr ::= Expr '+' Numero | Numero
```

Vamos ver na prática como eliminar a recursão à esquerda da gramática?

Seja gramática: $E \rightarrow E + F \mid F$

$F \rightarrow F * ID \mid ID$

$ID \rightarrow \mathbf{a} \mid \mathbf{b}$

Vamos fatorar essa gramática para eliminar a recursividade à esquerda, assim:

$E \rightarrow F E'$

Deslocamos a recursão para a direita

$E' \rightarrow + F E' \mid \varepsilon$

Criamos uma nova produção (fatoração), assim o símbolo terminal fica à esquerda, seguido da produção, e a recursão à esquerda. O processo recursivo termina em vazio.

$F \rightarrow ID F'$

O mesmo conceito aplicado para E é reproduzido

$F' \rightarrow * ID F' \mid \varepsilon$

para eliminar a recursividade em F

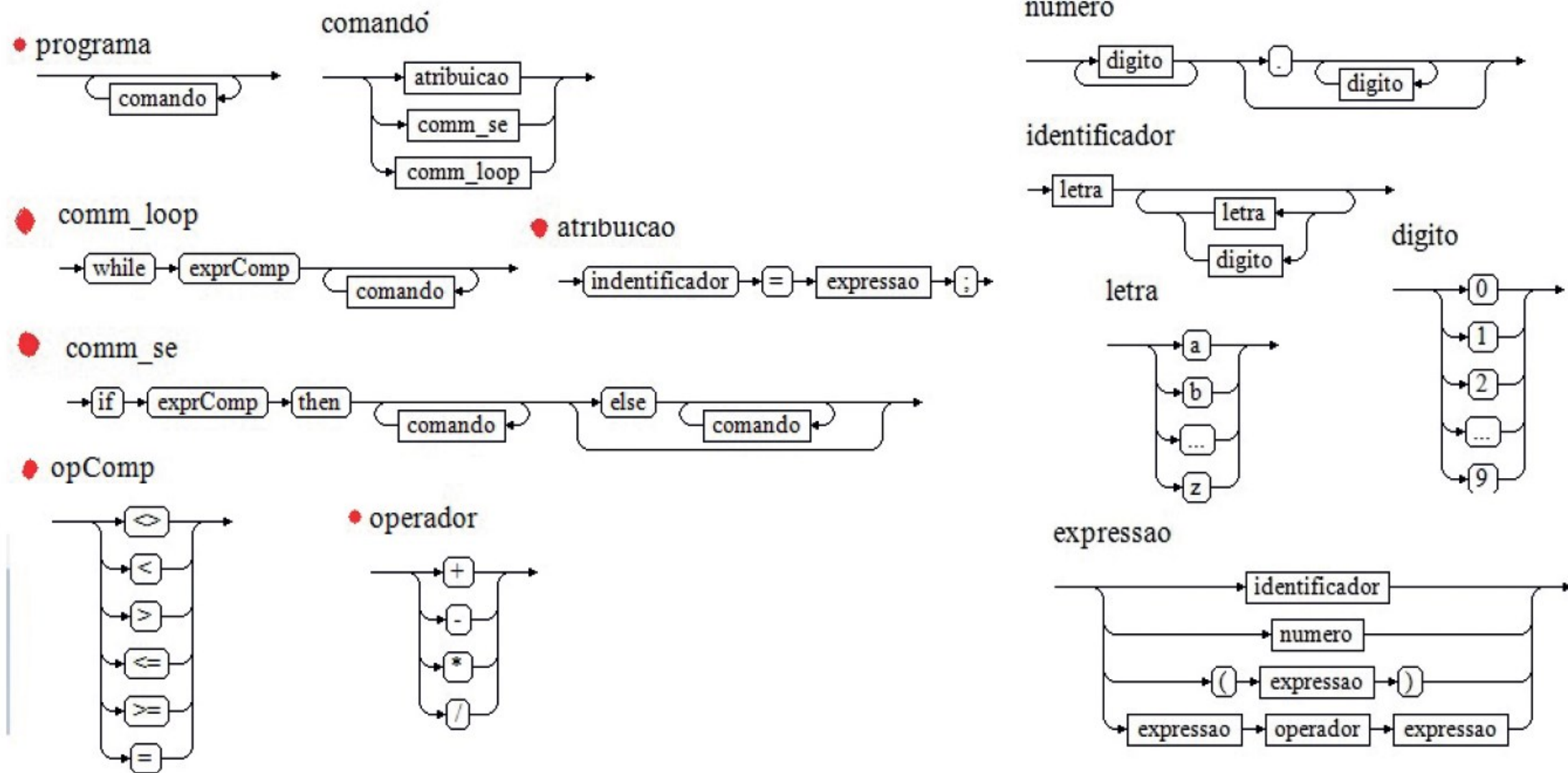
$ID \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

A nova gramática é equivalente, isto é, gera a mesma linguagem e não apresenta recursividade à esquerda.

é necessário tomar alguns cuidados para construir as gramáticas:

1. Analisar se a gramática definida é ambígua. Caso afirmativo, reescrevê-la, eliminando a ambiguidade;
2. Ao construir a gramática, não utilizar recursão a esquerda para que possa utilizar um analisador *top-down* sem retrocesso. E, caso haja recursão a esquerda, pode-se fatorar para eliminá-la.

Figura 3.4 | Diagrama de sintaxe proposto



Regras Sintáticas

$\langle \text{programas} \rangle ::= \{ \langle \text{comando} \rangle \}$

$\langle \text{comando} \rangle ::= \langle \text{atribuição} \rangle \mid \langle \text{comm_se} \rangle \mid \langle \text{comm_loop} \rangle$

$\langle \text{comm_loop} \rangle ::= \text{while } \langle \text{expr} \rangle \langle \text{comando} \rangle$

$\langle \text{comm_se} \rangle ::= \text{if } \langle \text{exprComp} \rangle \text{ then}$

$\langle \text{comando} \rangle \langle \text{comm} \rangle'$

$\langle \text{comm} \rangle' ::= \text{else } \langle \text{comando} \rangle \mid \epsilon$

$\langle \text{exprComp} \rangle ::= (\langle \text{id} \rangle \mid \langle \text{numero} \rangle) \langle \text{opComp} \rangle$
 $(\langle \text{id} \rangle \mid \langle \text{numero} \rangle)$

$\langle \text{atribuição} \rangle ::= \langle \text{id} \rangle = \langle \text{expr} \rangle$

$\langle \text{expr} \rangle ::= \langle \text{fator} \rangle \langle \text{expr1} \rangle$

$\langle \text{expr1} \rangle ::= + \langle \text{fator} \rangle \langle \text{expr1} \rangle \mid -$

$\langle \text{fator} \rangle \langle \text{expr1} \rangle \mid \epsilon$

$\langle \text{fator} \rangle ::= \langle \text{termo} \rangle \langle \text{fat1} \rangle$

$\langle \text{fat1} \rangle ::= * \langle \text{termo} \rangle \langle \text{fat1} \rangle \mid / \langle \text{termo} \rangle \langle \text{fat1} \rangle$
 $\mid \epsilon$

$\langle \text{termo} \rangle ::= \langle \text{numero} \rangle \mid \langle \text{id} \rangle \mid (\langle \text{expr} \rangle)$

Regras léxicas

$\langle \text{numero} \rangle ::= \langle \text{digito} \rangle \{ \langle \text{digito} \rangle \} [. \langle \text{digito} \rangle$
 $\{ \langle \text{digito} \rangle \}]$

$\langle \text{id} \rangle ::= \langle \text{letra} \rangle \{ \langle \text{letra} \rangle \mid \langle \text{digito} \rangle \}$

$\langle \text{letra} \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots \mid \mathbf{z}$

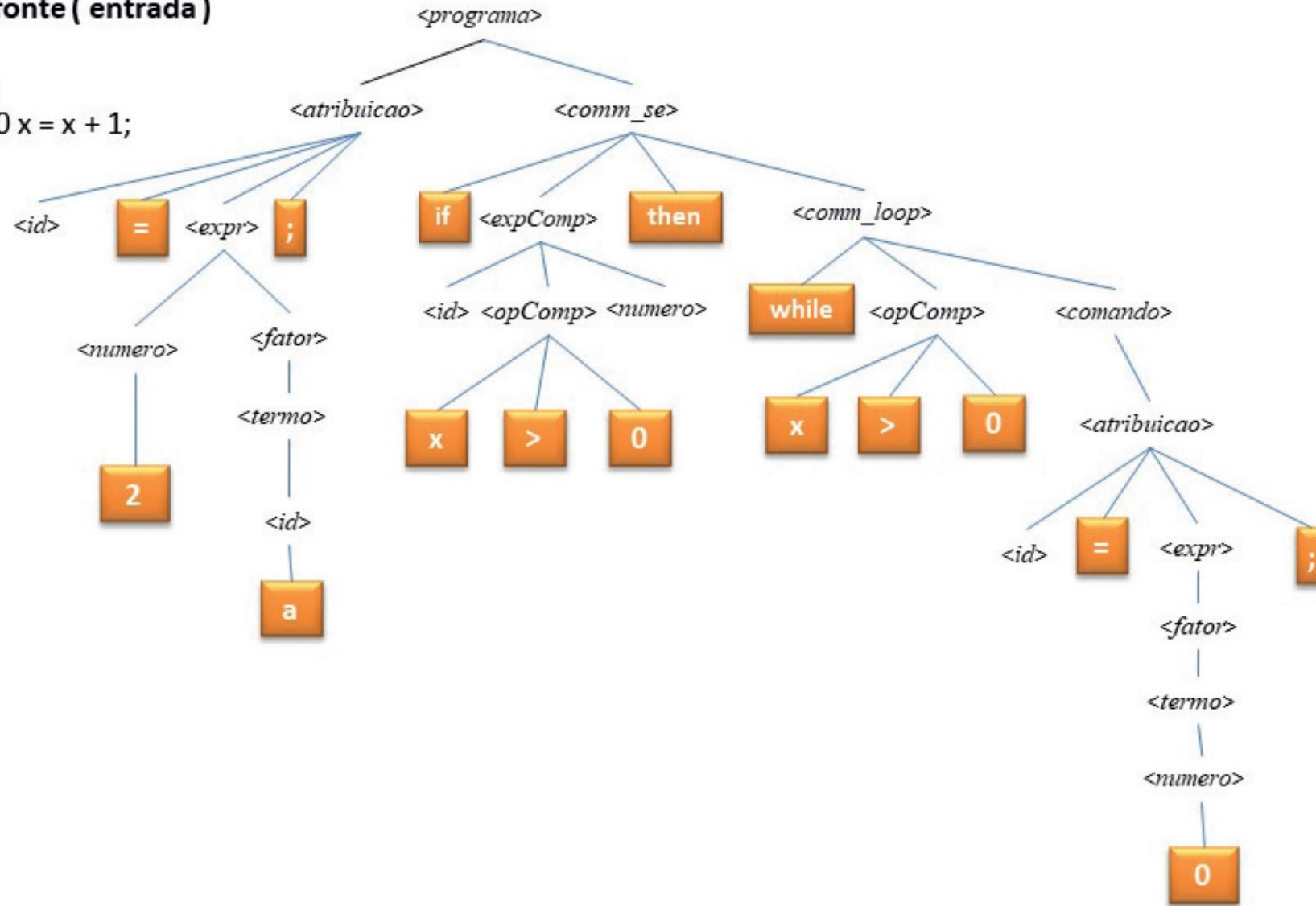
$\langle \text{numero} \rangle ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \dots \mid \mathbf{9}$

$\langle \text{opComp} \rangle ::= \langle \rangle \mid \langle \mid \rangle \mid \langle = \mid \rangle = \mid =$

Figura 3.5 | Árvore gramatical descendente

Programa fonte (entrada)

```
x = 2 * a ;  
if x > 0 then  
  while x > 0 x = x + 1 ;  
else  
  x = 0 ;
```



$\langle comm_se \rangle ::= \text{if } \langle condição \rangle \text{ then } \langle comm_se \rangle \text{ else } \langle comm_se \rangle \mid \text{if } \langle condição \rangle \text{ then } \langle comm_se \rangle$

Neste caso, if <condição> then <comm_se> aparece no início das duas opções, e o analisador, para escolher a opção correta, precisará avançar cinco (5) *tokens* à frente. Para solucionar este caso, pode-se reescrever a gramática:

$\langle comm_se \rangle ::= \text{if } \langle condição \rangle \text{ then } \langle comm_se \rangle \underline{\langle c_else \rangle}$

$\langle c_else \rangle ::= \text{else } \langle comm_se \rangle \mid \epsilon$

Essa solução, além de não precisar usar o retrocesso, também elimina a ambiguidade da gramática.