



Judson Santos Santiago

# Gramáticas

Compiladores

# Introdução

- As linguagens de programação possuem regras precisas para descrever a estrutura sintática de programas bem formados
  - Em linguagem C:
    - Um programa é composto por funções
    - Uma função é composta de declarações e instruções
    - Uma instrução é formada a partir de expressões
    - Etc.
- A estrutura sintática de uma linguagem pode ser especificada por uma gramática livre de contexto usando a notação BNF

# Introdução

## Gramática da Linguagem JSON

object : '{' pairs '}'

pairs : pair pairs\_tail |  $\epsilon$

pair : **STRING** ':' value

pairs\_tail : ',' pairs |  $\epsilon$

value : **STRING** | **NUMBER** | 'true' | 'false' | 'null' | object | array

array : '[' elements ']'

elements : value elements\_tail |  $\epsilon$

elements\_tail : ',' elements |  $\epsilon$

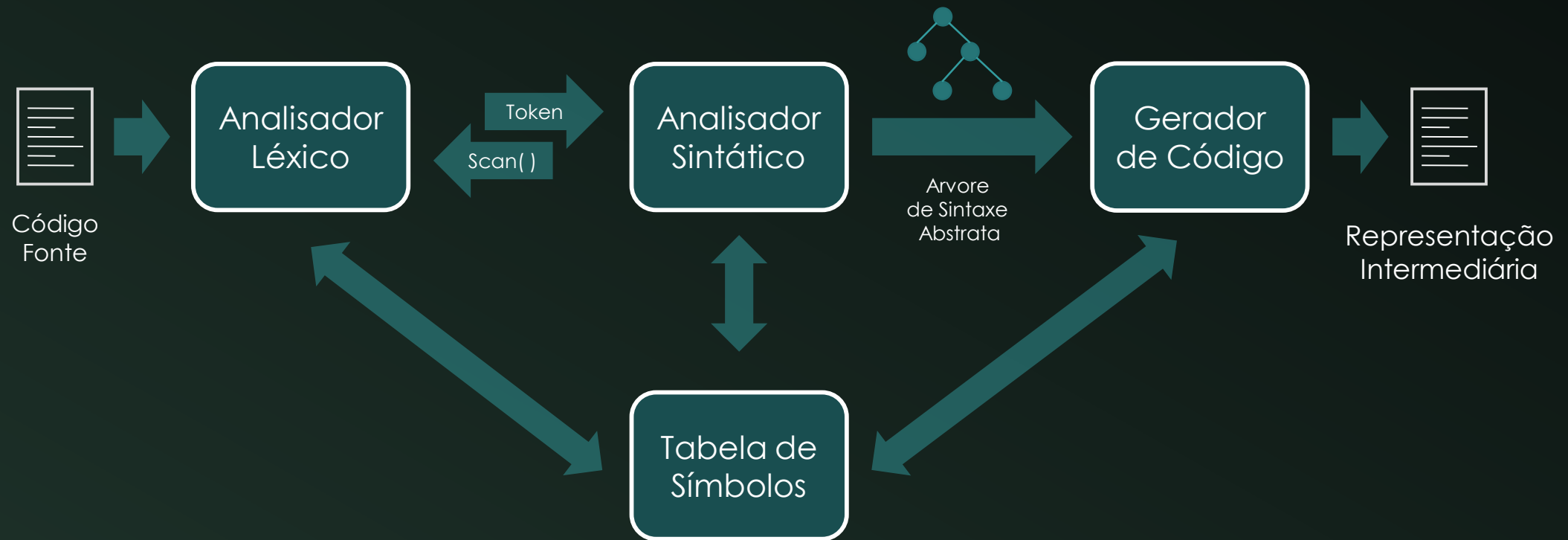
# Introdução

- As **gramáticas** oferecem diversos benefícios para o projetista de uma linguagem de programação
  - Especificação sintática precisa e fácil de entender
  - Tradução de programas fonte em código objeto
  - Exibição e detecção de erros
  - Detecção de ambiguidades
  - Construção da linguagem de forma gradual e iterativa
  - Construção automática de um analisador sintático eficiente

# Análise Sintática

- O **analisador sintático**:
  - Recebe uma **sequência de tokens**
  - Verifica se ela **obedece a gramática** da linguagem
  - Constrói uma representação da **árvore de derivação**
    - Uma **árvore sintática** é uma representação comum
- Nem sempre é necessário construir a árvore sintática
  - Tradução e verificação podem ser feitas juntas com a análise
  - Todo o *front-end* pode ser implementado em **um único módulo**

# Análise Sintática



Posicionamento do analisador  
sintático no compilador

# Análise Sintática

- Existem três estratégias de análise sintática
  - Análise universal
  - Análise descendente
  - Análise ascendente
- Características do método universal:
  - Algoritmo de Cocke-Younger-Kasami
  - Algoritmo de Earley
  - Podem analisar qualquer gramática
  - Ineficientes para compiladores comerciais

# Análise Sintática

- As **estratégias mais empregadas** em compiladores:
  - Análise descendente
  - Análise ascendente
- Os métodos mais eficientes usam **subclasses de gramáticas**
  - **Gramáticas LL**: utilizadas com analisadores descendentes (tipicamente implementados manualmente)
  - **Gramáticas LR**: utilizadas com analisadores ascendentes (tipicamente implementados por ferramentas automáticas)
  - Ambas são gramáticas livres de contexto



# Análise Sintática

- As **gramáticas LL** são aquelas tratadas pelos **analísadores preditivos**<sup>†</sup> e possuem as **restrições** vistas anteriormente:
  - Não podem ter recursividade à esquerda
  - Duas produções não podem iniciar com o mesmo terminal
  - Não pode existir **ambiguidade**
- Significado de **LL** e **LR**
  - O primeiro L diz respeito a varredura da entrada  
(da esquerda para direita)
  - A segunda letra diz respeito ao casamento com as produções  
(da esquerda para direita ou da direita para esquerda)

# Análise Sintática

- As **gramáticas LR** são tratadas por **analísadores shift-reduce**, que são métodos de análise ascendente
  - Empilham os tokens de entrada
  - Processam a gramática da direita para a esquerda
  - Esperam até casar todo o corpo de uma produção

```
object → '{' pairs '}'  
pairs  → pair pairs_tail | ε  
pair   → STRING ':' value  
value  → NUMBER | 'true' | 'false'
```

# Gramáticas

- Uma gramática livre de contexto é composta por:
  - Terminais: são os símbolos básicos que formam a linguagem  
Ex.: `if`, `else`, `{`, `}`, `(`, `)`
  - Não-terminais: são variáveis sintáticas que impõem uma hierarquia  
Ex.: `expr`, `term`, `factor`
  - Produções: especificam a relação entre terminais e não-terminais  
Ex.: `factor`  $\rightarrow$  `(expr)`
  - Símbolo inicial: é um não-terminal que define a linguagem  
Ex.: `program`  $\rightarrow$  `expression block`

# Gramáticas

- A gramática abaixo **não pertence** a classe de **gramáticas LL**
  - Inadequada para o método descendente
  - Possui **recursividade à esquerda**

```
expr    → expr + term  
         | term  
term    → term * factor  
         | factor  
factor  → ( expr )  
         | id
```

# Gramáticas

- A gramática pode ser adaptada para a análise descendente

*expr*     → *expr* + *term*  
          | *term*  
  
*term*     → *term* \* *factor*  
          | *factor*  
  
*factor*   → ( *expr* )  
          | *id*

Remoção  
da recursão  
à esquerda

*expr*     → *term* *expr*'  
*expr*'    → + *term* *expr*'  
          | ε  
  
*term*     → *factor* *term*'  
*term*'    → \* *factor* *term*'  
          | ε  
  
*factor*   → ( *expr* )  
          | *id*

# Gramáticas

- Uma **versão ambígua** é mostrada abaixo:

```
expr → expr + expr  
      | expr * expr  
      | -expr  
      | (expr)  
      | id
```

- A gramática ambígua permite **mais de uma árvore de derivação** para algumas entradas

Ex.:  $a + b * c$

# Derivações

- O **processo de derivação** é obtido pelos seguintes passos:
  1. A partir do símbolo inicial
  2. Substitui-se um não-terminal pelo corpo de uma de suas produções
  3. Repete-se o processo até que reste apenas símbolos terminais

$expr \rightarrow expr + expr$   
|  $expr * expr$   
|  $-expr$   
|  $(expr)$   
|  $id$

A cadeia  $-(id + id)$  é uma sentença da gramática porque existe uma derivação:

$expr \Rightarrow -expr$   
 $\Rightarrow -(expr)$   
 $\Rightarrow -(expr + expr)$   
 $\Rightarrow -(id + expr)$   
 $\Rightarrow -(id + id)$

# Derivações

- Em cada passo da derivação, existem **duas escolhas**:
  - Qual não-terminal substituir
  - Qual produção daquele não-terminal usar

$expr \rightarrow$   
|  $expr + expr$   
|  $expr * expr$   
|  $-expr$   
|  $(expr)$   
|  $id$

A cadeia  **$-(id + id)$**  poderia ser derivada também da seguinte forma:

$expr \Rightarrow$   $-expr$   
 $\Rightarrow$   $-(expr)$   
 $\Rightarrow$   $-(expr + expr)$   
 $\Rightarrow$   $-(expr + id)$   
 $\Rightarrow$   **$-(id + id)$**



# Derivações

- No exemplo abaixo, cada não-terminal é substituído pelo mesmo corpo em ambas derivações, mas a **ordem é diferente**
  - O não-terminal a ser substituído pode ser escolhido:
    - Por **derivações mais à esquerda**
    - Por **derivações mais à direita**

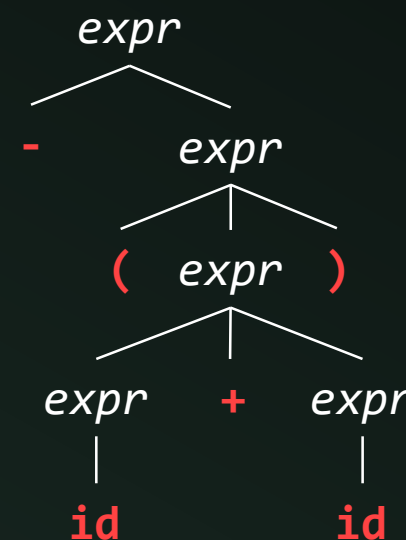
$expr \rightarrow expr + expr$   
|  $expr * expr$   
|  $-expr$   
|  $(expr)$   
|  $id$

$expr \Rightarrow -expr$   
 $\Rightarrow -(expr)$   
 $\Rightarrow -(expr + expr)$   
 $\Rightarrow -(id + expr)$   
 $\Rightarrow -(id + id)$

$expr \Rightarrow -expr$   
 $\Rightarrow -(expr)$   
 $\Rightarrow -(expr + expr)$   
 $\Rightarrow -(expr + id)$   
 $\Rightarrow -(id + id)$

# Árvore de Derivação

- A árvore de derivação é uma representação gráfica
  - Cada nó interior representa a aplicação de uma produção
  - As folhas, quando lidas da esquerda para a direita, constituem uma sentença da gramática
  - A árvore ao lado é o resultado das derivações mais à esquerda e mais à direita

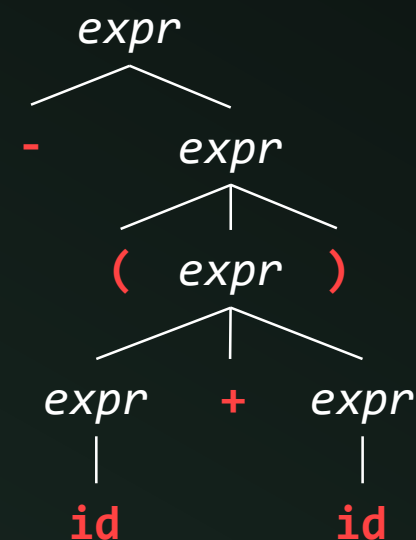


# Árvore de Derivação

- Para *cada árvore* existe:
  - Uma única derivação à esquerda
  - Uma única derivação à direita

$\text{expr} \Rightarrow -\text{expr}$   
 $\Rightarrow -( \text{expr} )$   
 $\Rightarrow -( \text{expr} + \text{expr} )$   
 $\Rightarrow -( \text{id} + \text{expr} )$   
 $\Rightarrow -( \text{id} + \text{id} )$

$\text{expr} \Rightarrow -\text{expr}$   
 $\Rightarrow -( \text{expr} )$   
 $\Rightarrow -( \text{expr} + \text{expr} )$   
 $\Rightarrow -( \text{expr} + \text{id} )$   
 $\Rightarrow -( \text{id} + \text{id} )$



# Ambiguidade

- Podemos definir ambiguidade agora de duas formas:
  - Uma gramática é ambígua se permitir a construção de mais de uma derivação mais à esquerda ou mais de uma derivação mais à direita para a mesma sentença

```
expr → expr + expr  
      | expr * expr  
      | -expr  
      | (expr)  
      | id
```

Duas formas de gerar `id + id * id`

```
expr ⇒ expr + expr  
      ⇒ id + expr  
      ⇒ id + expr * expr  
      ⇒ id + id * expr  
      ⇒ id + id * id
```

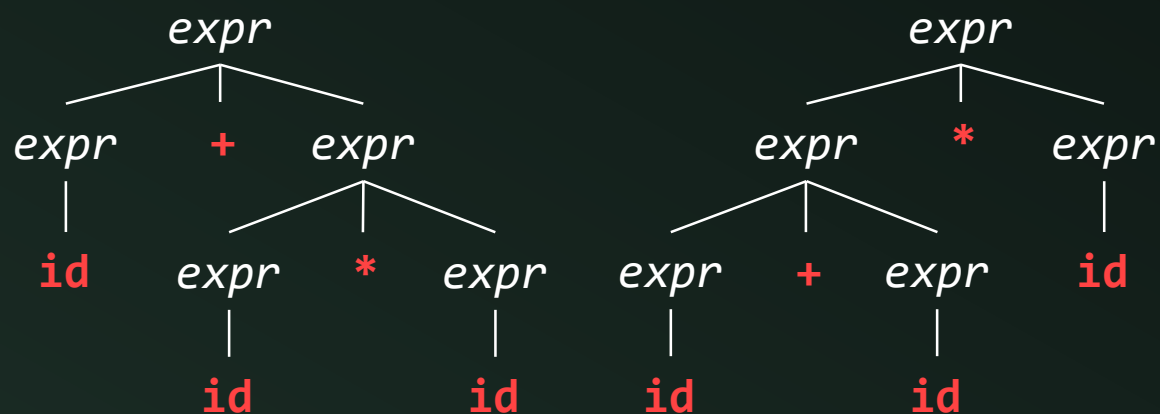
```
expr ⇒ expr * expr  
      ⇒ expr + expr * expr  
      ⇒ id + expr * expr  
      ⇒ id + id * expr  
      ⇒ id + id * id
```

# Ambiguidade

- Podemos definir ambiguidade agora de duas formas:
  - Uma gramática que produz mais de uma árvore de derivação é ambígua

$expr \rightarrow$   
 $expr + expr$   
 $expr * expr$   
 $-expr$   
 $(expr)$   
 $id$

Duas árvores de derivação para:  $id + id * id$



# Gramáticas e Expressões Regulares

- As gramáticas livres de contexto são consideradas construções mais poderosas que as expressões regulares
  - Toda construção que pode ser descrita por uma expressão regular pode ser descrita por uma gramática, mas não vice-versa

$(a|b)^*abb$

$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \epsilon$

Expressão regular e gramática descrevem a mesma linguagem

# Gramáticas e Expressões Regulares

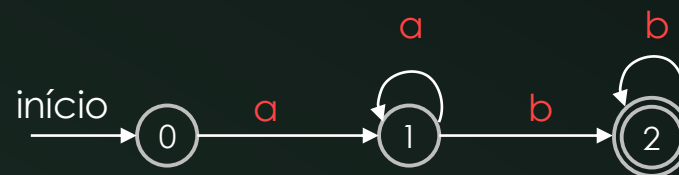
- A linguagem abaixo pode ser descrita por uma gramática mas não por uma expressão regular:

$$L = \{ a^n b^n \mid n \geq 1 \}$$

$$\begin{array}{l} S \rightarrow aSb \\ \mid \epsilon \end{array}$$

Gramática descreve  
linguagem L

Autômatos finitos não  
sabem contar



O diagrama descreve  
a expressão regular  
 $a^+b^+$

# Gramáticas e Expressões Regulares

- Então **por que usar expressões regulares** na análise léxica se as gramáticas são mais poderosas?
  - Elas fornecem uma notação mais **simples** e **compacta**
  - Permitem construir analisadores léxicos mais **eficientes**
- **Não existem regras** rígidas mas normalmente:
  - As **expressões regulares** são usadas no **léxico**
    - Descrevem identificadores, constantes, palavras reservadas, etc.
  - As **gramáticas** são usadas no **sintático**
    - Descrevem estruturas **begin-end**, **if-then-else**, **do-while**, etc.



# Exercício

1. Para cada uma das gramáticas e cadeias forneça:

- Uma derivação mais à esquerda
- Uma derivação mais à direita
- Uma árvore de derivação

a)  $S \rightarrow SS+ \mid SS^* \mid a$ , com a cadeia **aa+a\***

b)  $S \rightarrow 0S1 \mid 01$ , com a cadeia **000111**

c)  $S \rightarrow +SS \mid *SS \mid a$ , com a cadeia **+\*aaa**

# Exercício

a)  $S \rightarrow SS+$   
|  $SS^*$   
|  $a$

Entrada:  $aa+a^*$

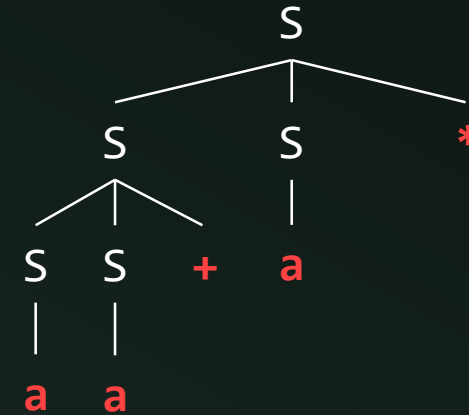
Derivação  
mais à  
Esquerda:

$S$   
 $SS^*$   
 $SS+S^*$   
 $aS+S^*$   
 $aa+S^*$   
 $aa+a^*$

Derivação  
mais à  
Direita:

$S$   
 $SS^*$   
 $Sa^*$   
 $SS+a^*$   
 $Sa+a^*$   
 $aa+a^*$

Árvore de  
Derivação:



# Exercício

b)  $S \rightarrow 0S1$   
|  $01$

Entrada: **000111**

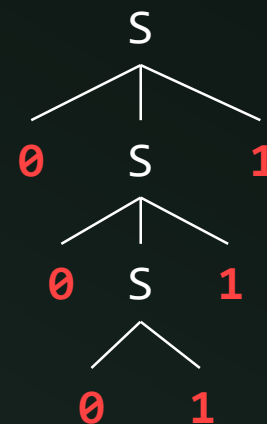
Derivação  
mais à  
Esquerda:

S  
0S1  
00S11  
000111

Derivação  
mais à  
Direita:

S  
0S1  
00S11  
000111

Árvore de  
Derivação:



# Exercício

c)  $S \rightarrow +SS$   
       $\quad | *SS$   
       $\quad | a$

Entrada:  **$+*aaa$**

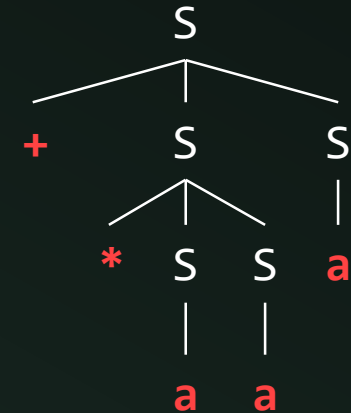
Derivação  
mais à  
Esquerda:

$S$   
 **$+SS$**   
 **$+*SSS$**   
 **$+*aSS$**   
 **$+*aaS$**   
 **$+*aaa$**

Derivação  
mais à  
Direita:

$S$   
 **$+SS$**   
 **$+Sa$**   
 **$+*SSa$**   
 **$+*Saa$**   
 **$+*aaa$**

Árvore de  
Derivação:



# Resumo

- Uma linguagem pode ser especificada por uma gramática
  - As gramáticas mais usadas são do tipo LL ou LR
    - Analisadores LL são obtidos com derivações mais à esquerda
    - Analisadores LR são obtidos com derivações mais à direita
  - Não é desejável trabalhar com gramáticas ambíguas
    - Possuem mais de uma árvore de derivação
    - Ou mais de uma derivação mais à esquerda ou mais à direita
  - Gramáticas são mais expressivas que expressões regulares
    - Toda expressão regular pode ser descrita por uma gramática
    - O contrário não é verdade