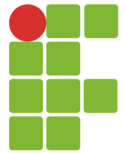
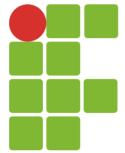
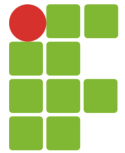


# Sumário

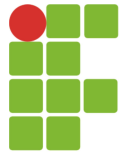


# Sumário



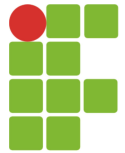
# Introdução

- Estamos acostumados a utilizar regras gramaticais para escrever períodos sintaticamente corretos.
- Essas regras consideram sujeitos, verbos e objetos, por exemplo.
- Gramáticas explicam o que está ausente ou supérfluo em uma construção.



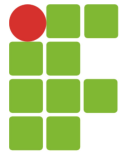
# Introdução

- Em outras palavras, as gramáticas fornecem uma definição concisa de como períodos podem ser construídos corretamente do ponto de vista sintático.
- Também podem ser utilizadas para diagnosticar períodos incorretamente construídos.



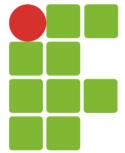
# Introdução

- É claro que podemos escrever períodos sintatica corretos, mas que não fazem sentido algum do ponto de vista semântico, do seu significado.
- Gramáticas lidam com a estrutura textual (sintática) e representam apenas uma parte do que é possível verificar do ponto de vista linguístico.



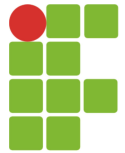
# Introdução

- Linguagens de programação modernas possuem gramáticas em sua especificação.
- Essa gramática é utilizada por quem ensina, estuda ou utiliza a linguagem.
- A gramática também é fundamental para construção de **analisadores sintáticos**.
- Aspectos que não podem ser verificados sintaticamente, são resolvidos em uma outra etapada, pelo **analisador semântico**.

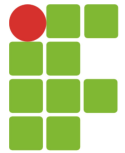


# Introdução

- Estudaremos as gramáticas livre de contexto (em inglês, context-free grammar ou CFG) para que possamos ter um mecanismo para definir linguagens simples.
- Formalizaremos as definições e algoritmos para analisar as gramáticas para, posteriormente, apresentarmos as técnicas de análise sintática.

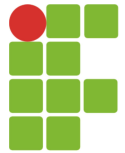


# Sumário

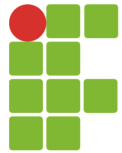


# CFG

- Como vimos anteriomente, linguagens definem conjuntos de palavras sobre um alfabeto finito.
- A maioria das linguagens interessantes são conjuntos infinitos, então não podemos simplesmente enumerar todos os seus elementos.
- Ao definir uma linguagem através de uma CFG, podemos representá-la compactamente.



# Sumário

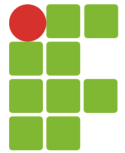


# CFG: definição

## CFG

Uma CFG possui 4 componentes:

1. Um alfabeto finito $\Sigma$ de **terminais**. Esse conjunto é o conjunto de tokens produzido pelo analisador léxico. $\$ \in \Sigma$, em que $\$$ é um símbolo especial que denota fim de entrada.
2. Um alfabeto finito $\mathcal{N}$ de **não-terminais**. Símbolos de $N$ são conhecidos como variáveis.
3. $S \in \mathcal{N}$, o símbolo inicial. Todas as derivações da gramática partem de $S$.
4. Um conjunto de produções $\mathcal{P}$, que são da forma $A \to X_1 \dots X_m$, em que $X_i \in \mathcal{N} \cup \Sigma$, $1 \leq i \leq m$ e $m \geq 0$. A única produção válida com $m = 0$ é $A \to \varepsilon$, em que $\varepsilon$ é a string vazia.



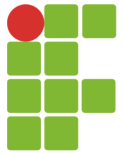
# CFG: definição

## CFG

- Assim, podemos definir uma gramática $G$ como uma quádrupla:

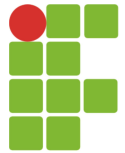
$$G = (\mathcal{N}, \Sigma, \mathcal{P}, S)$$

- Os alfabetos de terminais e não-terminais são disjuntos: $\Sigma \cap \mathcal{N} = \emptyset$.
- O vocabulário $\mathcal{V}$ de uma CFG é o conjunto dos símbolos terminais e não terminais: $\mathcal{V} = \Sigma \cup \mathcal{N}$



# CFG: definição

- Uma CFG é uma receita para derivar strings.
- Começando do símbolo inicial, $S$, aplicamos regras de produção até obter apenas símbolos terminais.
- Uma regra de produção $A \to \alpha$ troca o não-terminal $A$ (lado esquerdo, ou LHS), pela sequência $\alpha$ (lado direito ou RHS). Onde os símbolos de $\alpha$ são terminais ou não-terminais.
- A regra $A \to \varepsilon$ faz com que $A$ efetivamente seja apagado.



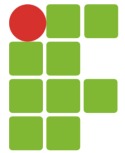
# CFG: definição

## Derivação

Se $A \rightarrow \gamma$ é uma regra de produção, então $\alpha A \beta \Rightarrow \alpha \gamma \beta$ denota um passo de derivação usando esta regra uma única vez.

Podemos estender $\Rightarrow$, para $\Rightarrow^+$, que denota uma derivação obtida a partir de $A$ com um ou mais passos.

Analogamente $\Rightarrow^*$ é definido como a derivação obtida a partir de $A$ com zero ou mais passos.

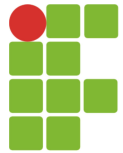


# CFG: definição

## Forma sentencial

Se $S \Rightarrow^* \beta$, dizemos que $\beta$ é uma **forma sentencial** da gramática $G$. Em outras palavras:

$$\mathrm{SF}(G) = \{\beta | S \Rightarrow^* \beta\}$$



# CFG: definição

## Linguagem descrita por $G$

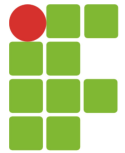
A linguagem descrita por uma gramática, $G$, denotada por $\mathrm{L}(G)$ é o conjunto de strings de $\Sigma^*$ derivadas de $S$:

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

.

Observa-se que $\mathrm{L}(G) = \mathrm{SF}(G) \cap \Sigma^*$.

Efetivamente $\mathrm{L}(G)$ é o conjunto de palavras, compostas apenas por símbolos terminais, que pode ser gerada a partir de $G$ através de derivações.



# CFG: definição

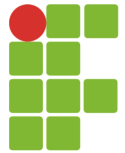
## Abreviação das regras de produção

É comum ter várias regras de produção com o mesmo LHS:

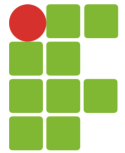
$$\begin{aligned} A &\rightarrow \alpha \\ A &\rightarrow \beta \\ &\dots \\ A &\rightarrow \zeta \end{aligned}$$

Podemos abreviar de acordo com a seguinte notação:

$$A \rightarrow \alpha \mid \beta \mid \dots \mid \zeta$$



# Sumário

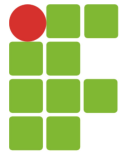


# CFG: exemplo

Tome a seguinte gramática:

$$\begin{aligned} E &\rightarrow P(E) \mid v\ T \\ P &\rightarrow f \mid \varepsilon \\ T &\rightarrow +E \mid \varepsilon \end{aligned}$$

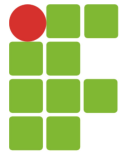
Em que $S = E$, $\mathcal{N} = \{E, P, T\}$ e $\Sigma = \{v, f, +, (, )\}$.
Efetivamente, é uma gramática que gera expressões de soma simples, envolvendo funções unárias e variáveis.



# CFG: exemplo

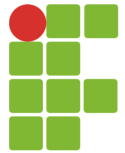
A partir de $E$, podemos obter as seguintes derivações:

$$
\begin{aligned}
E &\Rightarrow P(E) \\
P(E) &\Rightarrow f(E) \\
f(E) &\Rightarrow f(vT) \\
f(vT) &\Rightarrow f(v+E) \\
f(v+E) &\Rightarrow f(v+vT) \\
f(v+vT) &\Rightarrow f(v+v\varepsilon) = f(v+v)
\end{aligned}
$$

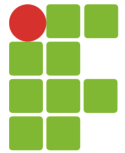


# CFG: Exemplo

- É correto dizer que $f(v+v)$ é gerado por $G$, isto é, que $f(v+v) \in \mathrm{L}(G)$, ou que $E \Rightarrow^{+} f(v+v)$.
- Também é correto dizer que: $\{P(E), f(E), f(vT), f(v+E), f(v+vT), f(v+v)\} \subseteq \mathrm{SF}(G)$.



# Sumário

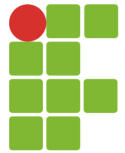


# Derivações mais à esquerda

## Definição

Uma derivação mais à esquerda é aquela que escolhe a opção mais à esquerda possível das regras de produção a cada passo de modo a produzir a string desejada .

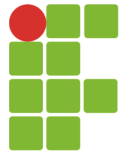
Para denotar derivações mais à esquerda, utilizamos $\Rightarrow_{\text{lm}}$, $\Rightarrow^*_{\text{lm}}$ e $\Rightarrow^+_{\text{lm}}$.



# Derivações mais à esquerda

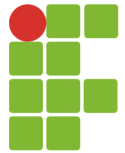
Considerando a gramática do exemplo anterior, a derivação mais à esquerda de $f(v+v)$ é:

$$
\begin{aligned}
E &\Rightarrow_{\text{lm}} P(E) \\
P(E) &\Rightarrow_{\text{lm}} f(E) \\
f(E) &\Rightarrow_{\text{lm}} f(vT) \\
f(vT) &\Rightarrow_{\text{lm}} f(v+E) \\
f(v+E) &\Rightarrow_{\text{lm}} f(v+vT) \\
f(v+vT) &\Rightarrow_{\text{lm}} f(v+v\varepsilon) = f(v+v)
\end{aligned}
$$

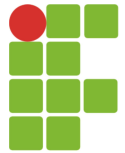


# Derivações mais à esquerda

- As sequências de derivações pela classe dos analisadores sintáticos top-down são derivações mais à esquerda.
- Dizemos que esses parsers produzem um parse mais à esquerda.



# Sumário

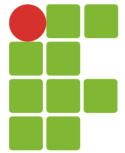


# Derivações mais à direita

## Definição

Uma derivação mais à direita é aquela que escolhe a opção mais à direta possível das regras de produção a cada passo de modo a produzir a string desejada .

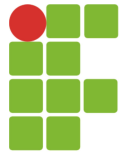
Para denotar derivações mais à direita, utilizamos $\Rightarrow_{\mathrm{rm}}$, $\Rightarrow_{\mathrm{rm}}^{*}$ e $\Rightarrow_{\mathrm{rm}}^{+}$.



# Derivações mais à direita

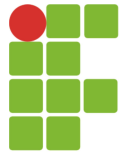
Considerando a gramática do exemplo anterior, a derivação mais à direita de $f(v+v)$ é:

$$
\begin{aligned}
E &\Rightarrow_{\text{rm}} P(E) \\
P(E) &\Rightarrow_{\text{rm}} P(vT) \\
P(vT) &\Rightarrow_{\text{rm}} P(v+E) \\
P(v+E) &\Rightarrow_{\text{rm}} P(v+vT) \\
P(v+vT) &\Rightarrow_{\text{rm}} P(v+v\varepsilon) = P(v+v) \\
P(v+v) &\Rightarrow_{\text{rm}} f(v+v)
\end{aligned}
$$

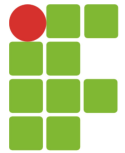


# Derivações mais à direita

- As sequências de derivações pela classe dos analisadores sintáticos bottom-up são derivações mais à esquerda.
- Dizemos que esses parsers produzem um parse mais à direita.



# Sumário



# Árvores sintáticas

- Derivações são representadas por uma árvore sintática, também chamada de árvore de derivações ou do inglês (*parse tree*).
- Uma árvore sintática possui as seguintes características:
  - Sua raiz é o símbolo inicial $S$ da gramática.
  - Cada nó é um símbolo do vocabulário da gramática ou $\varepsilon$.
  - Os nós internos são símbolos não-terminais. Um nó interno representando um não terminal $A$ terá filhos $X_1, X_2, \ldots, X_m$ se, e somente se, existir uma regra de produção $A \rightarrow X_1 X_2 \ldots X_m$.
  - Quando a derivação está completa, cada folha desta árvore é um símbolo terminal ou $\varepsilon$.

# Árvores sintáticas

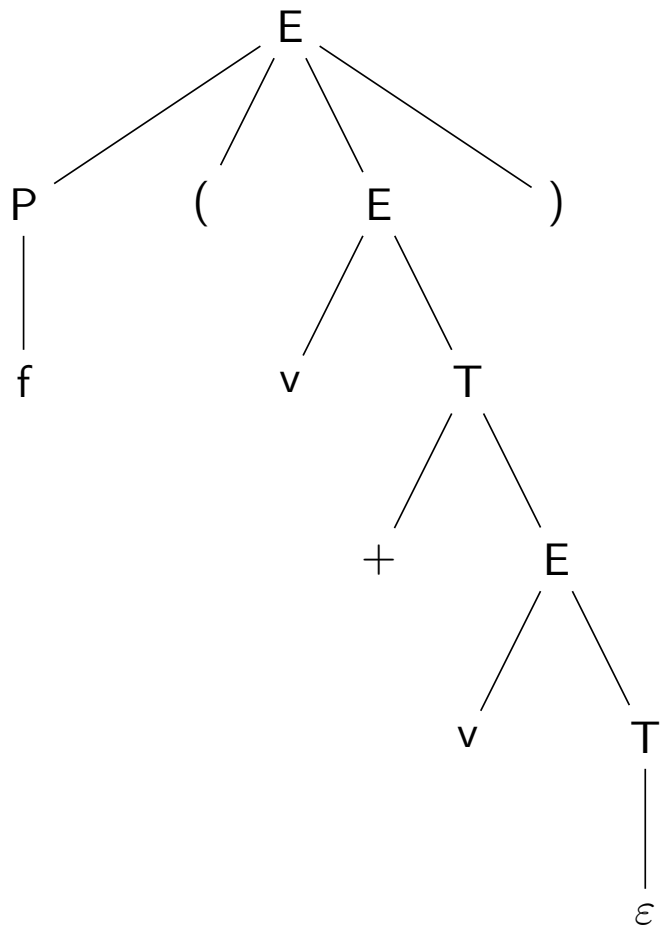
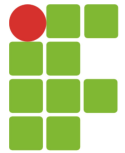



Figura: Árvore sintática para $f(v + v)$



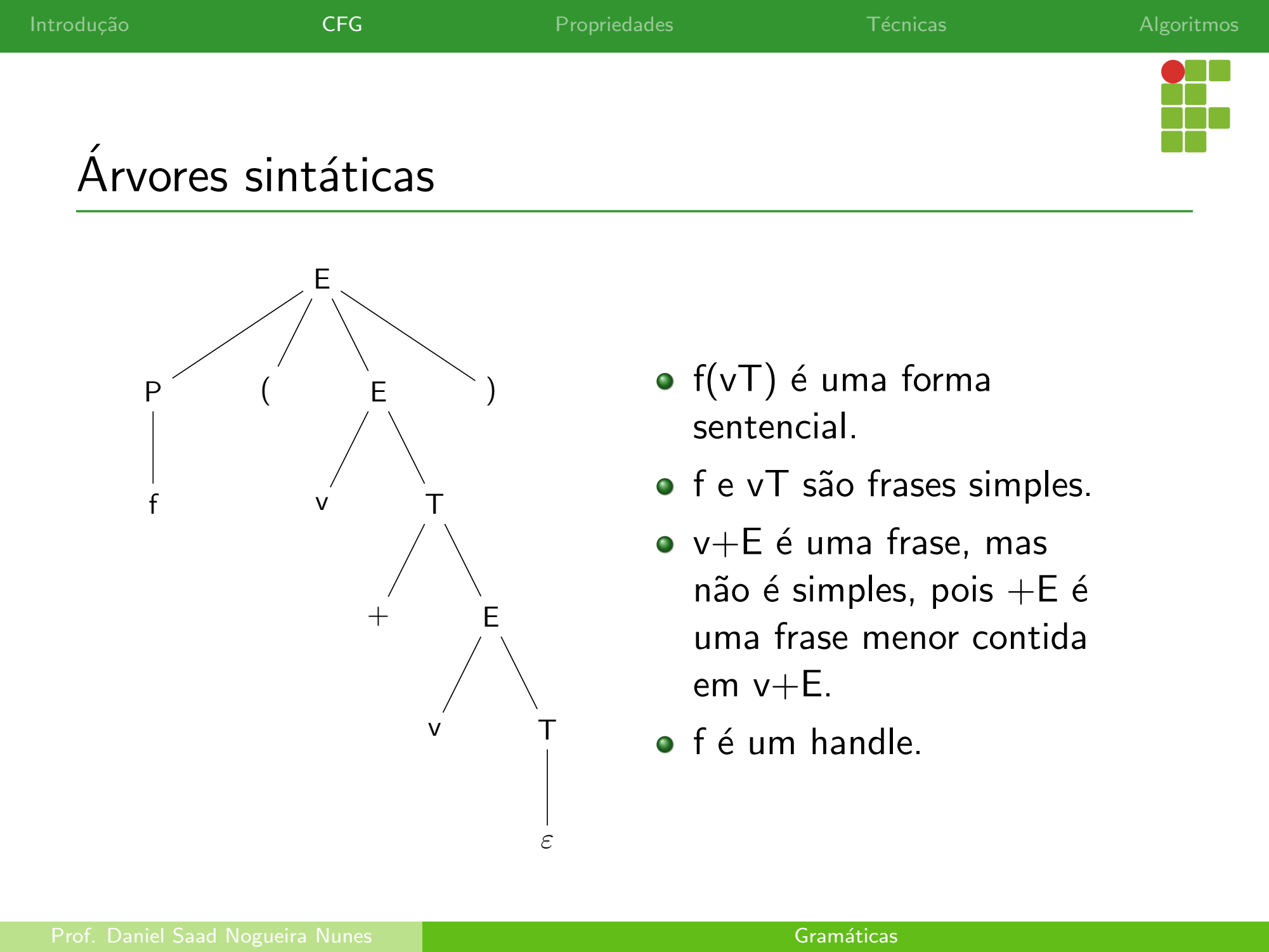
# Árvores Sintáticas

## Frases

Uma frase de uma árvore sintática é uma sequência de símbolos derivados de um mesmo não-terminal.

A frase é dita simples se não contém uma frase menor.

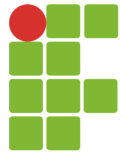
Um handle de uma forma sentencial é a frase simples mais à esquerda.



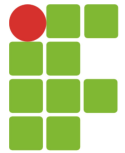
# Árvores sintáticas




- f(vT) é uma forma sentencial.
- f e vT são frases simples.
- v+E é uma frase, mas não é simples, pois +E é uma frase menor contida em v+E.
- f é um handle.

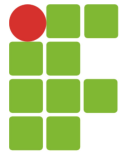


# Sumário

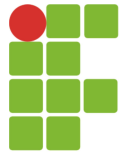


# Propriedades

- CFGs são mecanismos para descrever linguagens.
- Assim como temos vários programas que computam o mesmo resultado, várias gramáticas diferentes podem descrever a mesma linguagem.
- Algumas gramáticas são mais apropriadas que outras por não apresentar alguns problemas, tais como:
  - Símbolos inúteis.
  - Possuir diverentes árvores sintáticas para uma mesma string.
  - Incluir strings que não pertençam à linguagem ou excluir string que pertençam a linguagem.



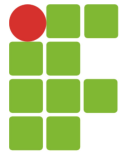
# Sumário



# Gramáticas reduzidas

## Definição

Uma gramática é dita **reduzida** se cada não-terminal e regras de produção participam da derivação de pelo menos uma string. Símbolos não-terminais que podem ser removidos sem prejuízo são chamados de inúteis.

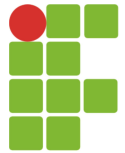


# Gramáticas reduzidas

Tome a seguinte gramática:

$$\begin{aligned} S &\rightarrow A \mid B \\ A &\rightarrow a \\ B &\rightarrow Bb \\ C &\rightarrow c \end{aligned}$$

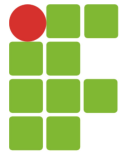
Claramente o não-terminal $C$ é inútil, visto que não pode aparecer em nenhuma frase. Além disso, qualquer frase que mencione $B$ não pode derivar uma sequência contendo apenas terminais.



# Gramáticas reduzidas

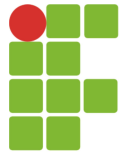
Quando $B$ e $C$ são removidos, bem como as regras de produção associadas à eles, temos a seguinte **gramática reduzida**:

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow a \end{aligned}$$

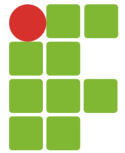


# Gramáticas reduzidas

- Uma gramática que não esteja na sua forma reduzida pode indicar erros provenientes de especificações erradas.



# Sumário



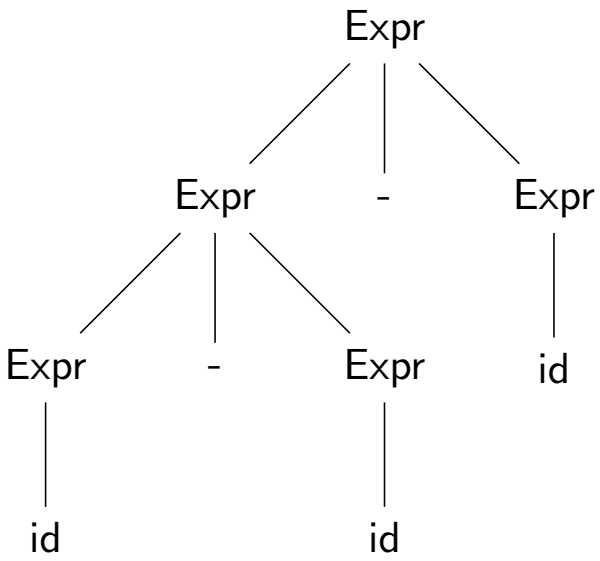
# Ambiguidade

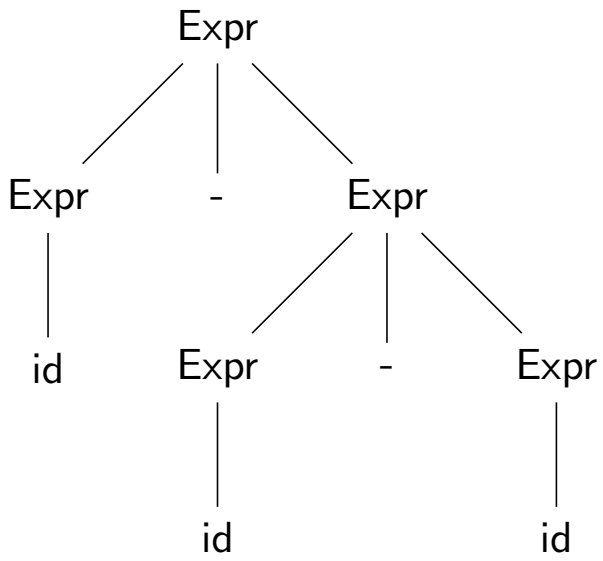
- Gramáticas ambíguas permitem que diferentes árvores sintáticas sejam construídas para uma mesma string derivada.
- Considere a seguinte gramática:

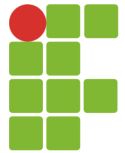
$$\text{Expr} \rightarrow \text{Expr} - \text{Expr} \mid \text{id}$$

- Ela possui duas árvores de derivação para a string id – id – id.

# Ambiguidade

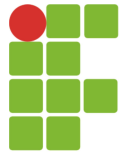




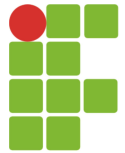
# Ambiguidade

- Se os símbolos id possuem valroes $3$, $2$ e $1$ a avaliação dada pela primeira árvore é $(3-2)-1=0$, enquanto a avaliação da segunda árvore nos dá $3-(2-1)=2$.

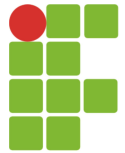


# Ambiguidade

- Gramáticas ambíguas raramente são utilizadas, pois uma tradução única não pode ser obtida.
- Infelizmente um algoritmo que verifica se uma dada gramática é ambígua não existe.
- O problema de verificar se uma gramática é ambígua é **indecidível**.

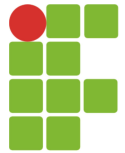


# Sumário



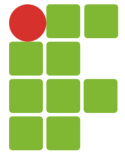
# Problemas de especicação

- Se a gramática for mal especificada, ela pode descrever strings indesejadas, isto é, strings que não deveriam pertencer à linguagem
- Pode ocorrer também dela deixar de descrever strings que pertençam à linguagem.

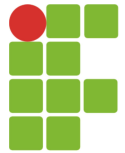


# Problemas de especificação

- A correção de uma gramática na prática é verificada através de testes.
- Infelizmente não há algoritmo que compare duas CFGs e determine se elas descrevem a mesma linguagem.
- O problema da equivalência das linguagens descritas por gramáticas é indecidível.

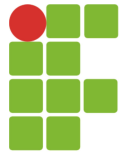


# Sumário



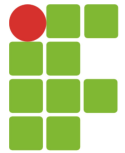
# Técnicas de análise sintática

- Compiladores devem avaliar se a estrutura sintática de um programa está correta.
- Em outras palavras, para uma entrada $x$, o compilador deve verificar se $x \in \mathrm{L}(G)$.
- O algoritmo que executa este teste é denominado **reconhecedor**.



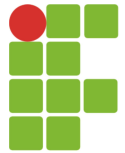
# Técnicas de análise sintática

- Além de determinar se uma string pertence ou não à linguagem descrita por uma gramática, também é necessário determinar a estrutura sintática dela, isto é, sua árvore sintática.
- A análise sintática, ou parser, é responsável por esta tarefa.



# Técnicas de análise sintática

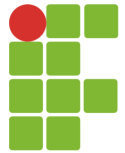
- Existem duas abordagens para análise sintática: top-down e bottom-up.



# Técnicas de análise sintática

## Top-down

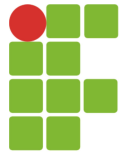
A análise sintática é considerada top-down se gera a árvore sintática começando da raiz (símbolo inicial) e expandindo a árvore através das regras de produção através de uma **busca em profundidade** em pré-ordem.



# Técnicas de análise sintática

## Bottom-up

A análise sintática é considerada bottom-up se a árvore sintática é gerada no sentido folhas para raiz. Um nó é inserido na árvore apenas após os seus filhos terem sido processados. Uma análise bottom-up corresponde à um percurso em pós-ordem da árvore sintática.



# Exemplo

Tome a seguinte gramática:

$$
\begin{aligned}
\text{Program} &\rightarrow \text{begin Stmts end \$} \\
\text{Stmts} &\rightarrow \text{Stmt ; Stmts} \mid \varepsilon \\
\text{Stmt} &\rightarrow \text{simplestmt} \mid \text{begin Stmts end}
\end{aligned}
$$

# Exemplo: top-down

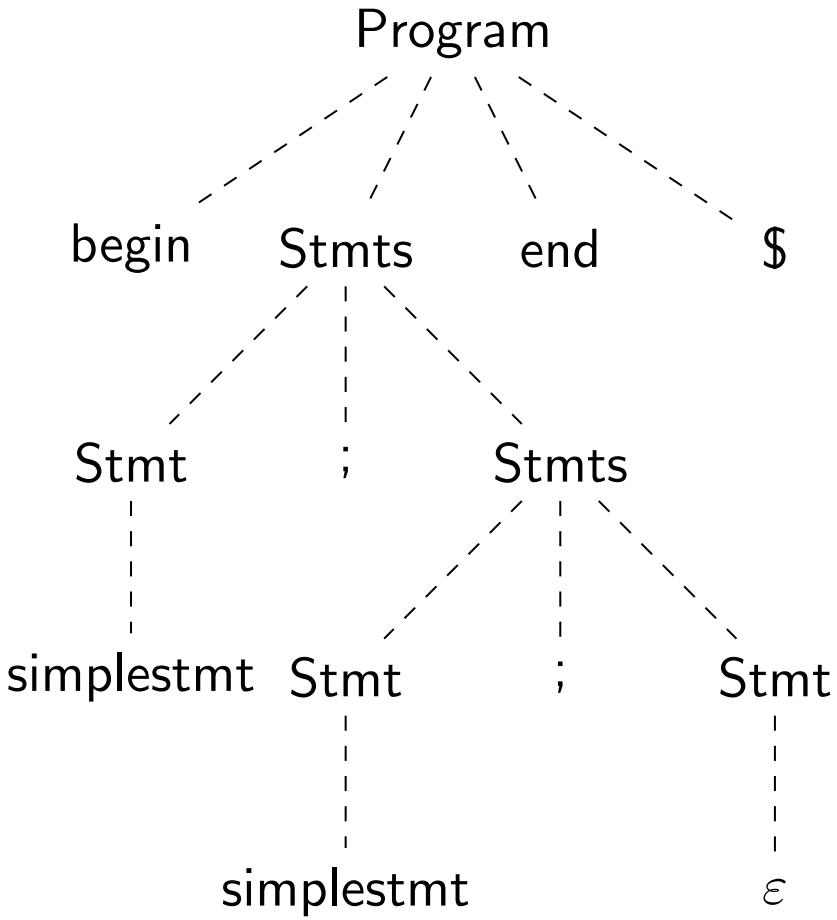



Figura: Expansão top-down

# Exemplo: top-down

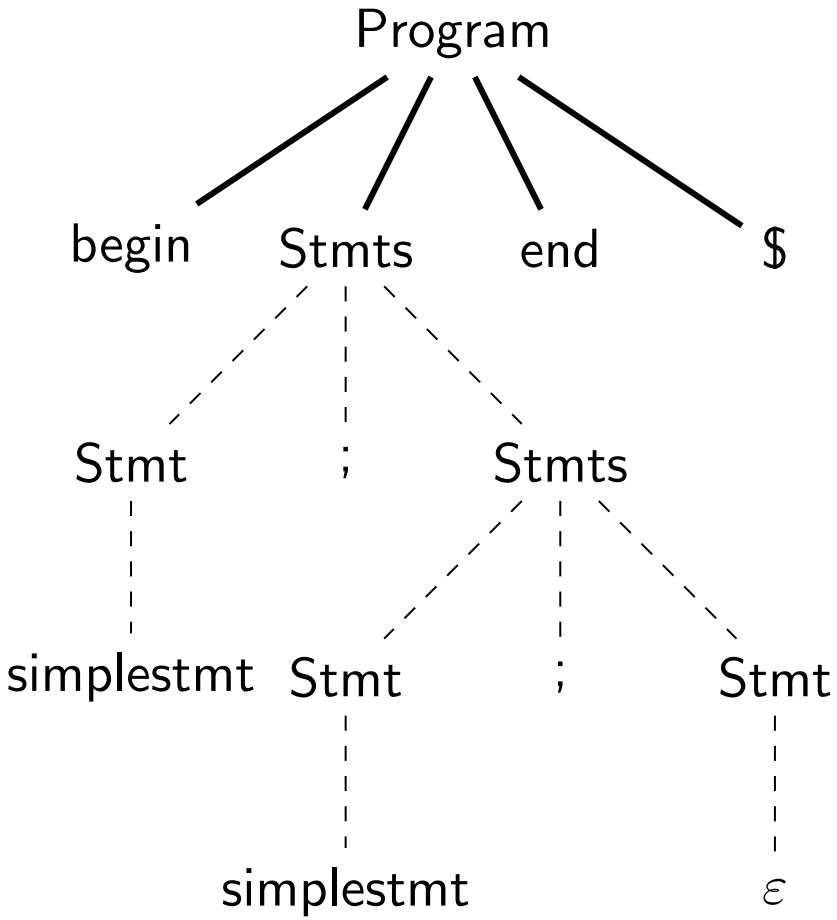



Figura: Expansão top-down

# Exemplo: top-down

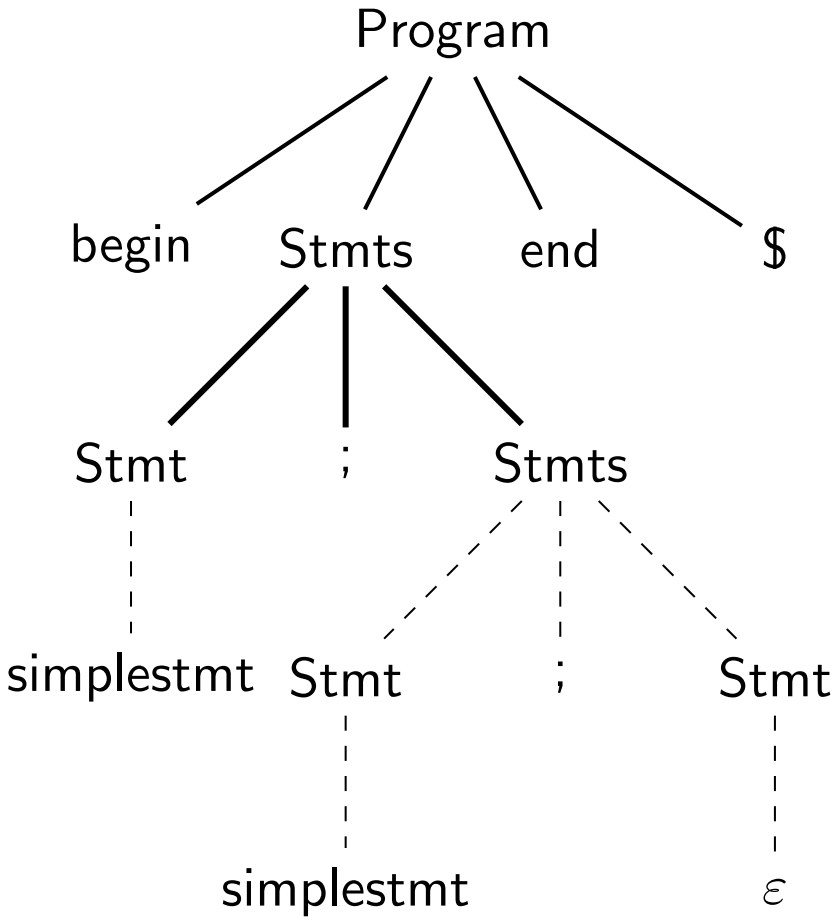



Figura: Expansão top-down

# Exemplo: top-down

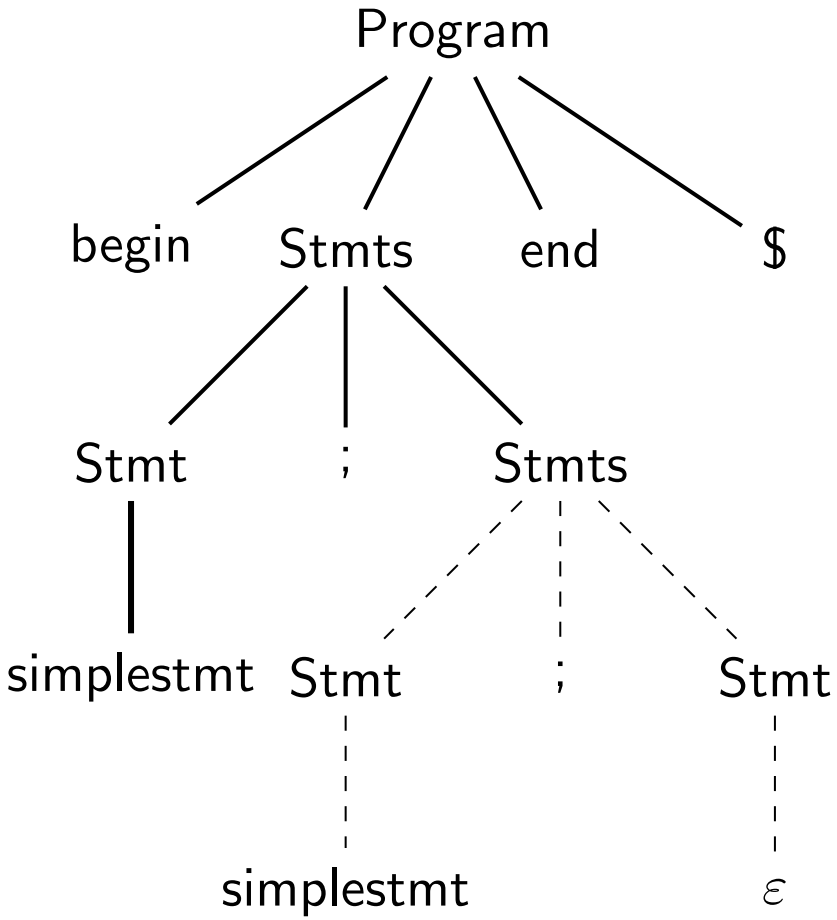



Figura: Expansão top-down

# Exemplo: top-down

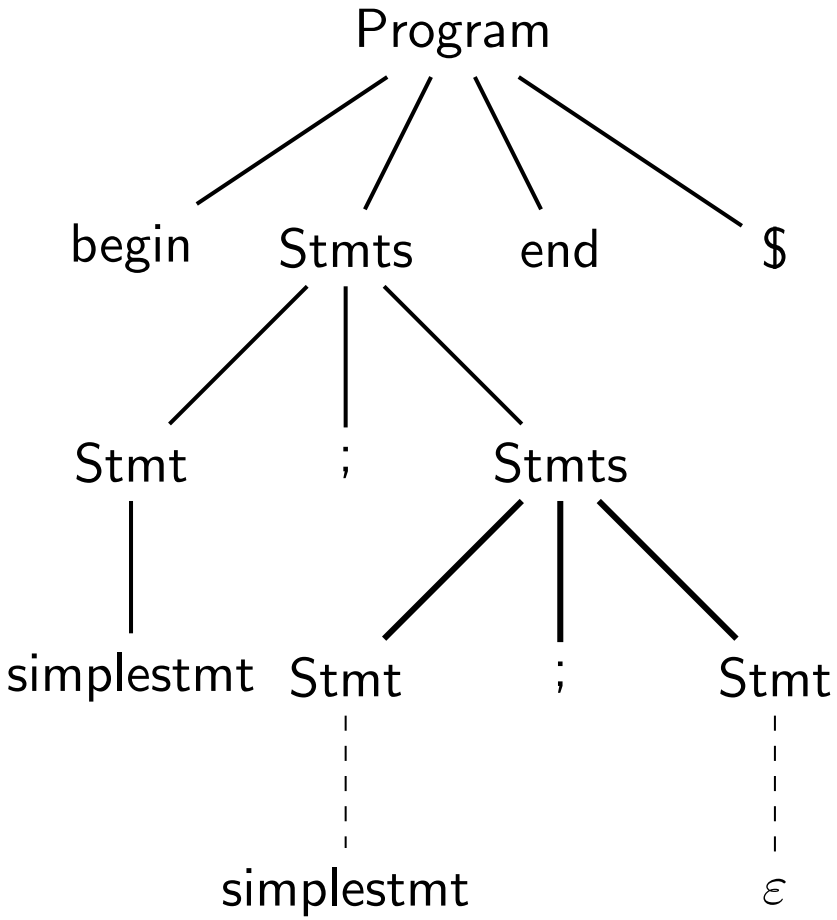



Figura: Expansão top-down

# Exemplo: top-down

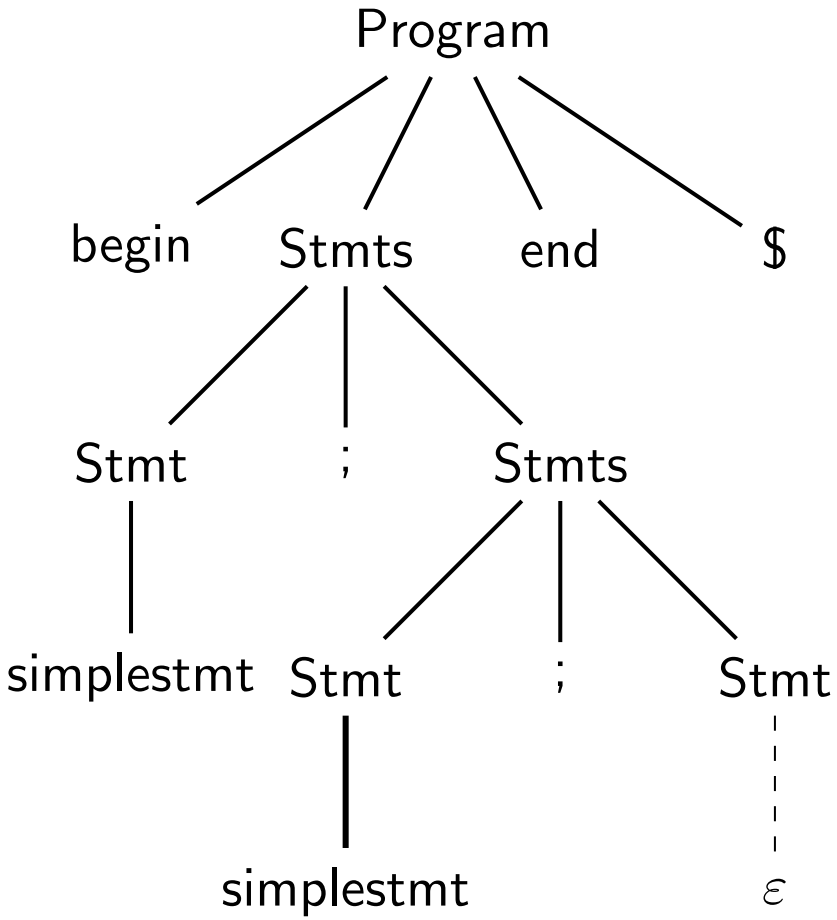



Figura: Expansão top-down

# Exemplo: top-down

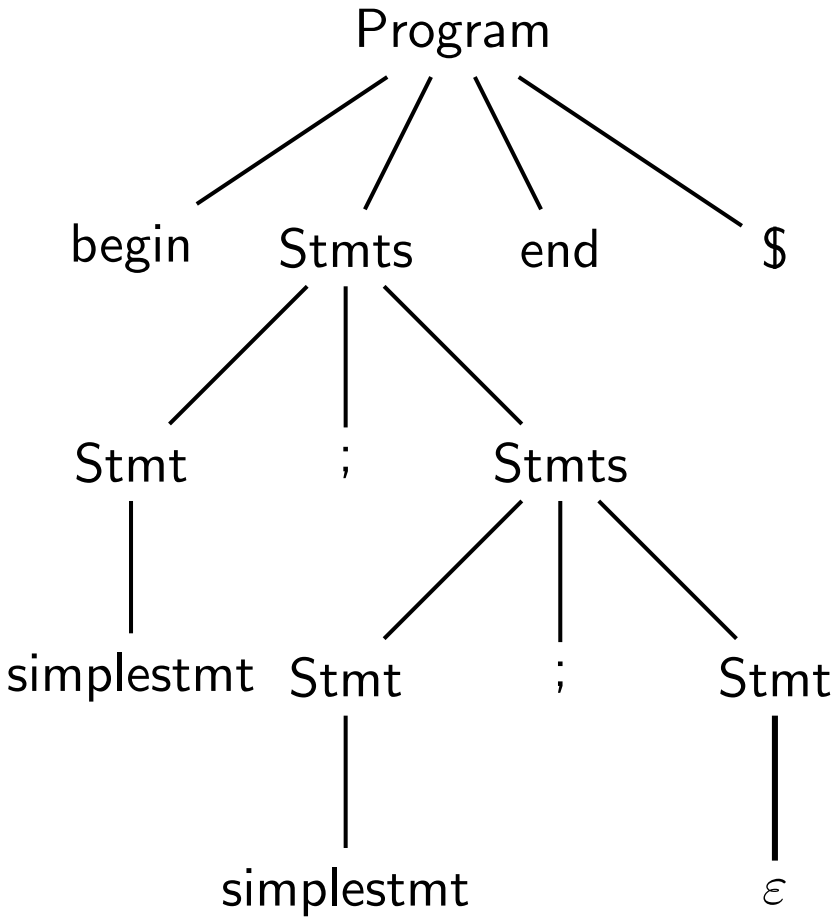



Figura: Expansão top-down

# Exemplo: top-down

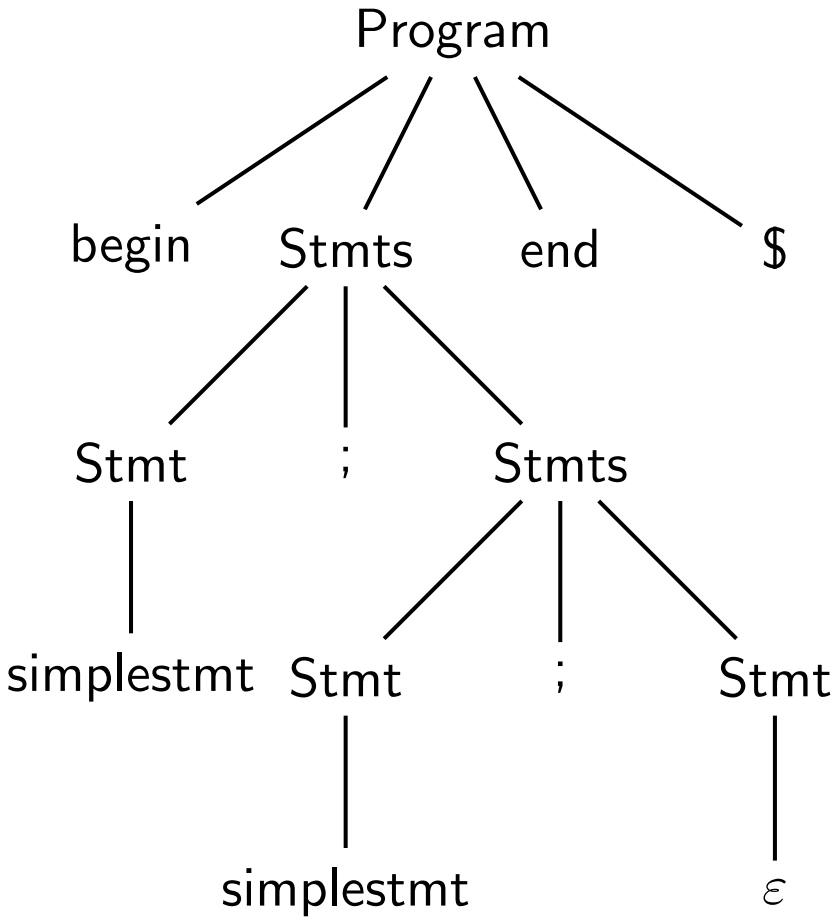



Figura: Expansão top-down

# Exemplo: bottom-up

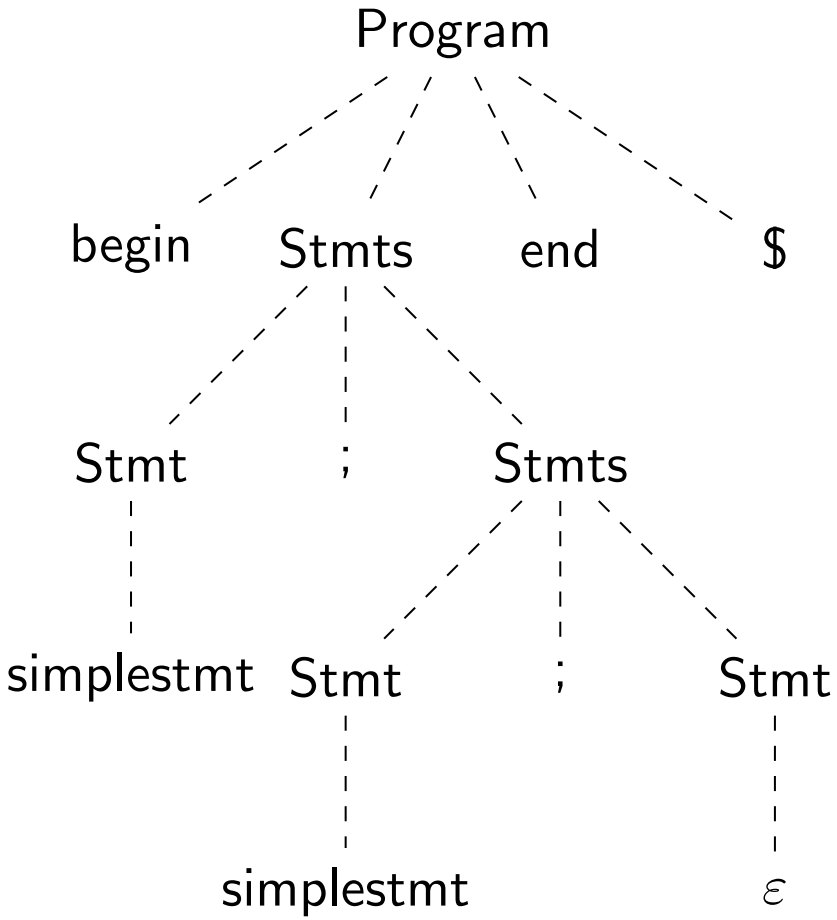



Figura: Expansão bottom-up

# Exemplo: bottom-up

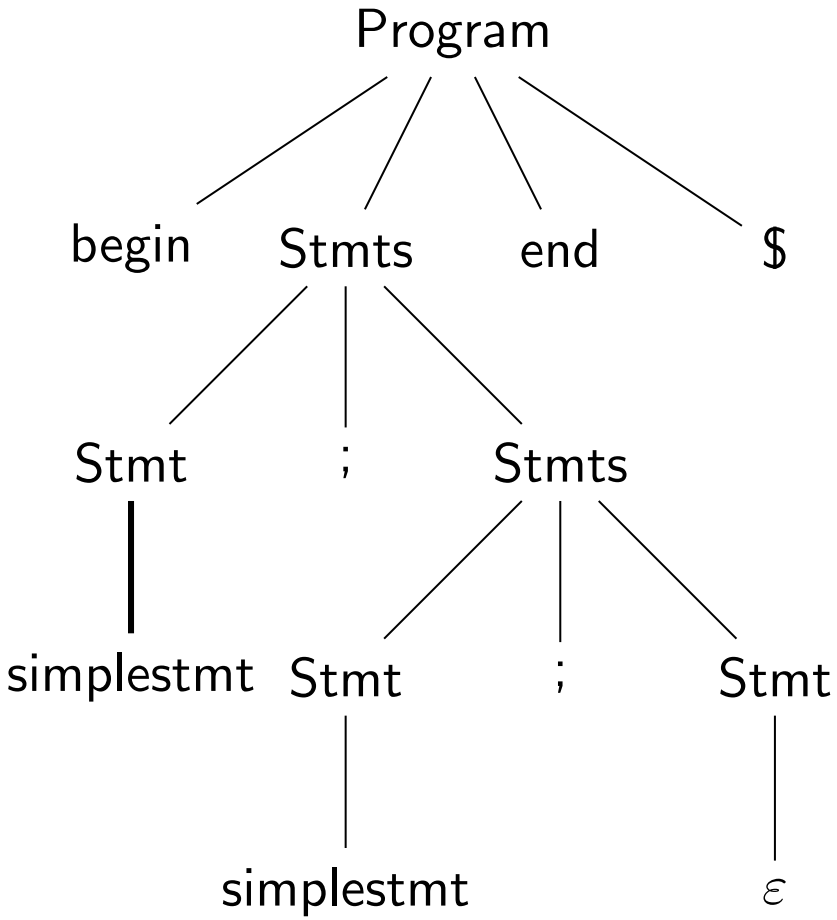



Figura: Expansão bottom-up

# Exemplo: bottom-up

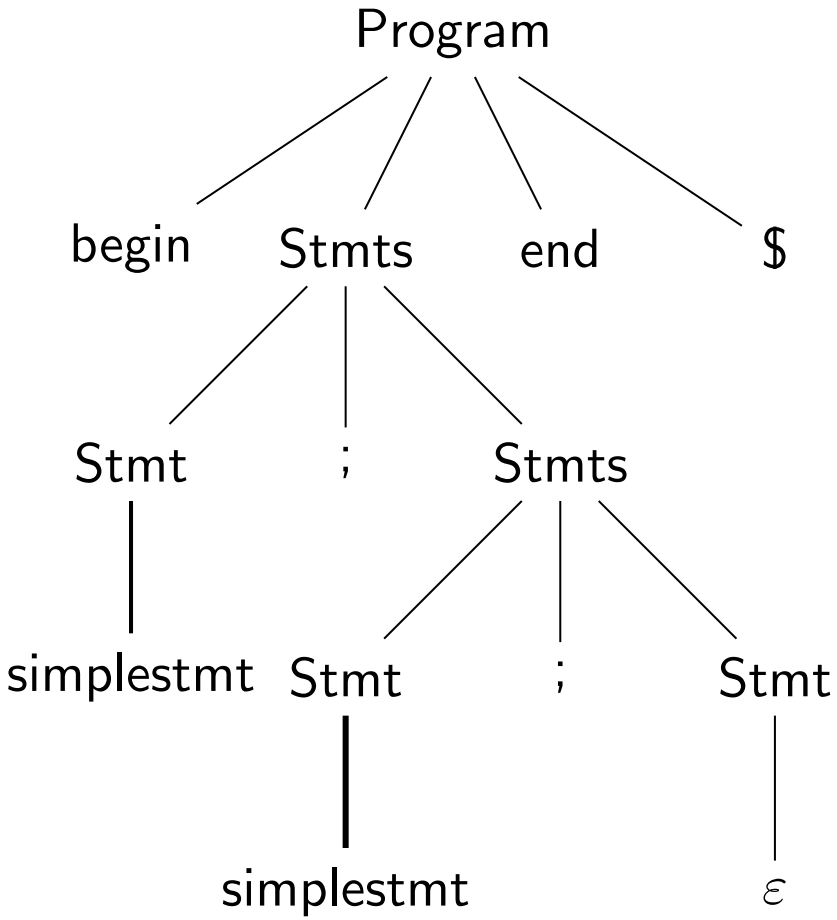



Figura: Expansão bottom-up

# Exemplo: bottom-up

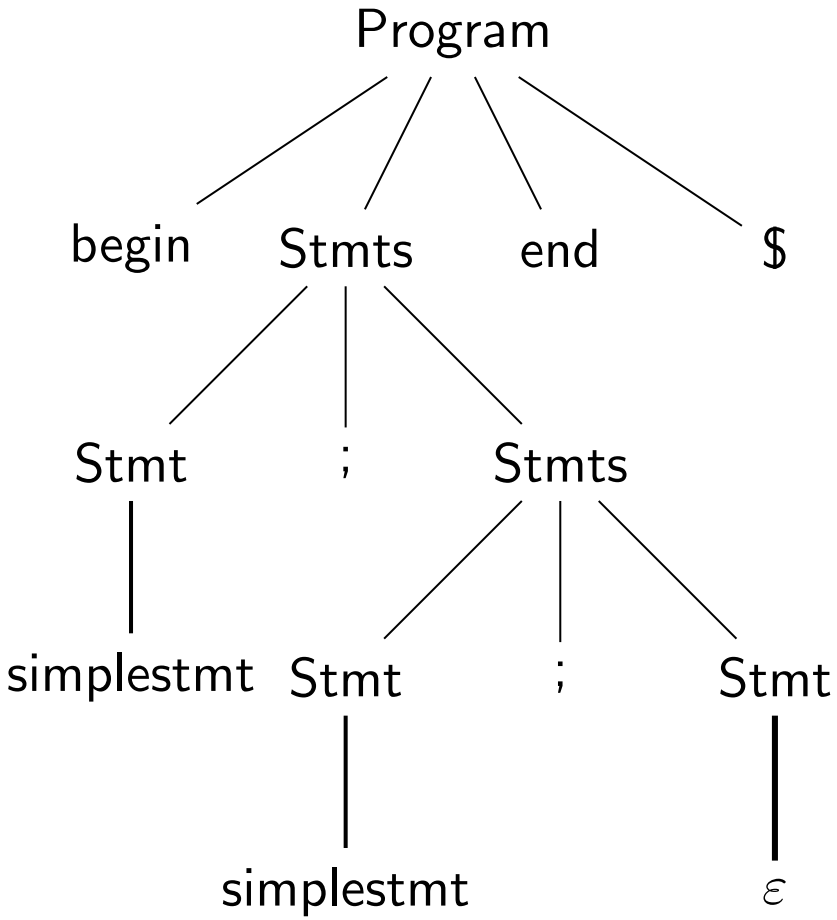



Figura: Expansão bottom-up

# Exemplo: bottom-up

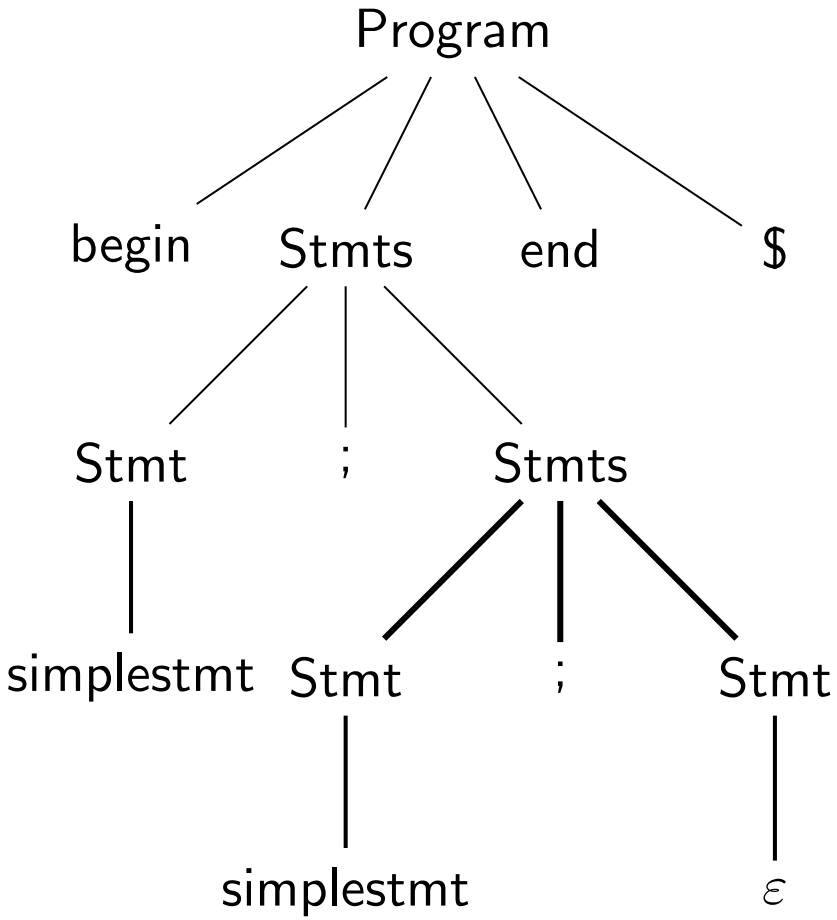



Figura: Expansão bottom-up

# Exemplo: bottom-up

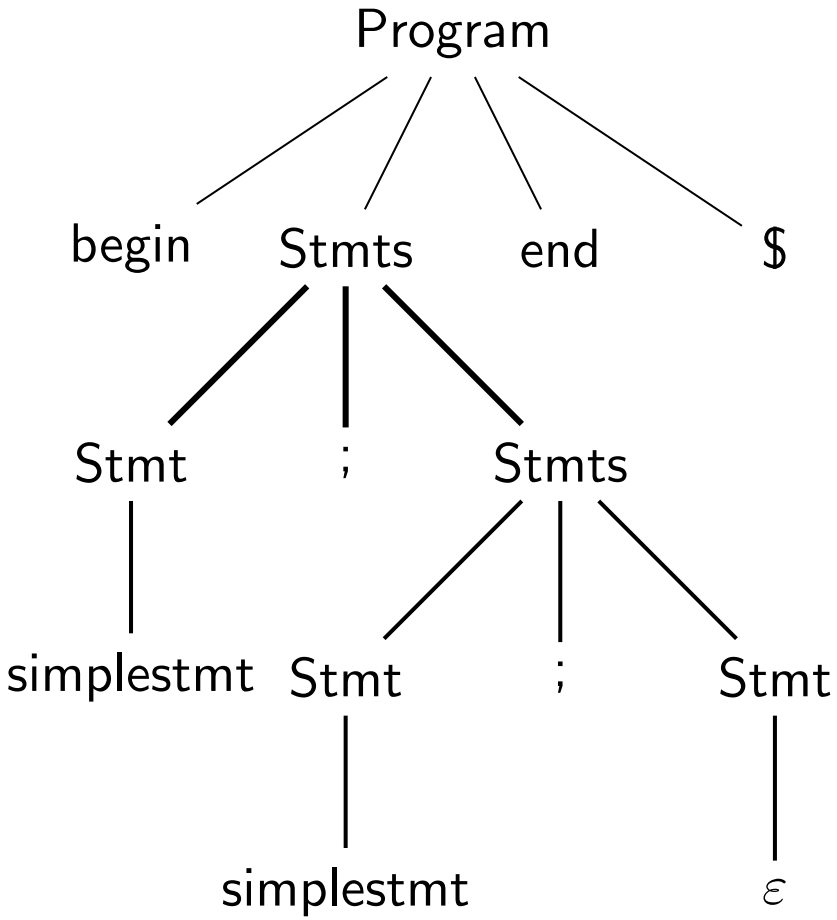



Figura: Expansão bottom-up

# Exemplo: bottom-up

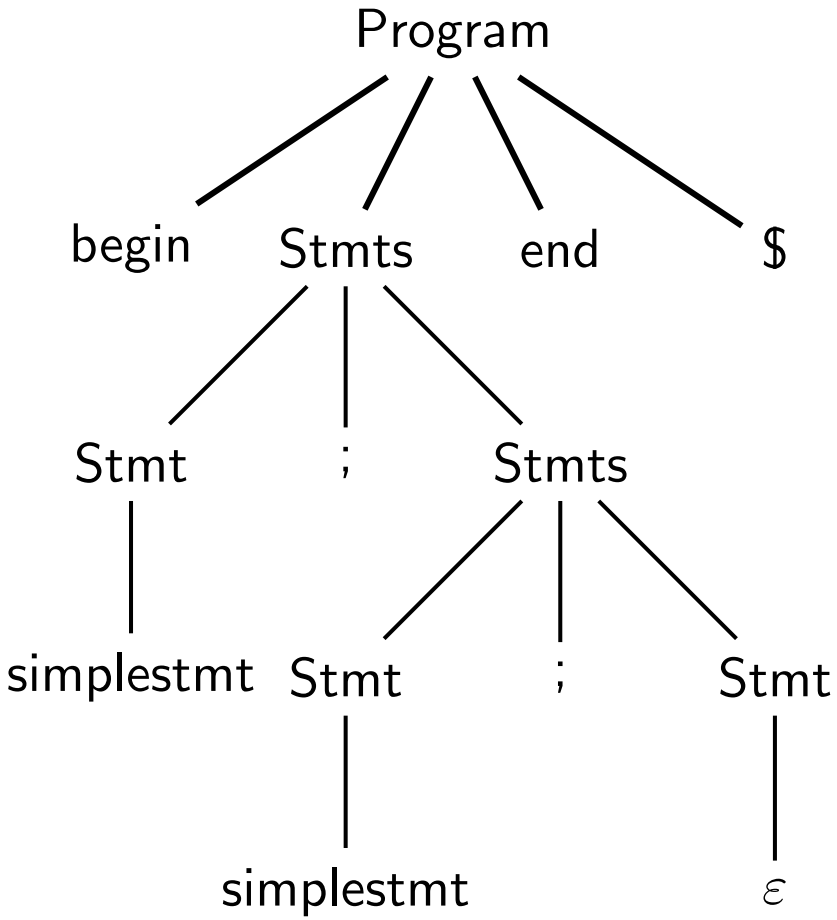



Figura: Expansão bottom-up

# Exemplo: bottom-up

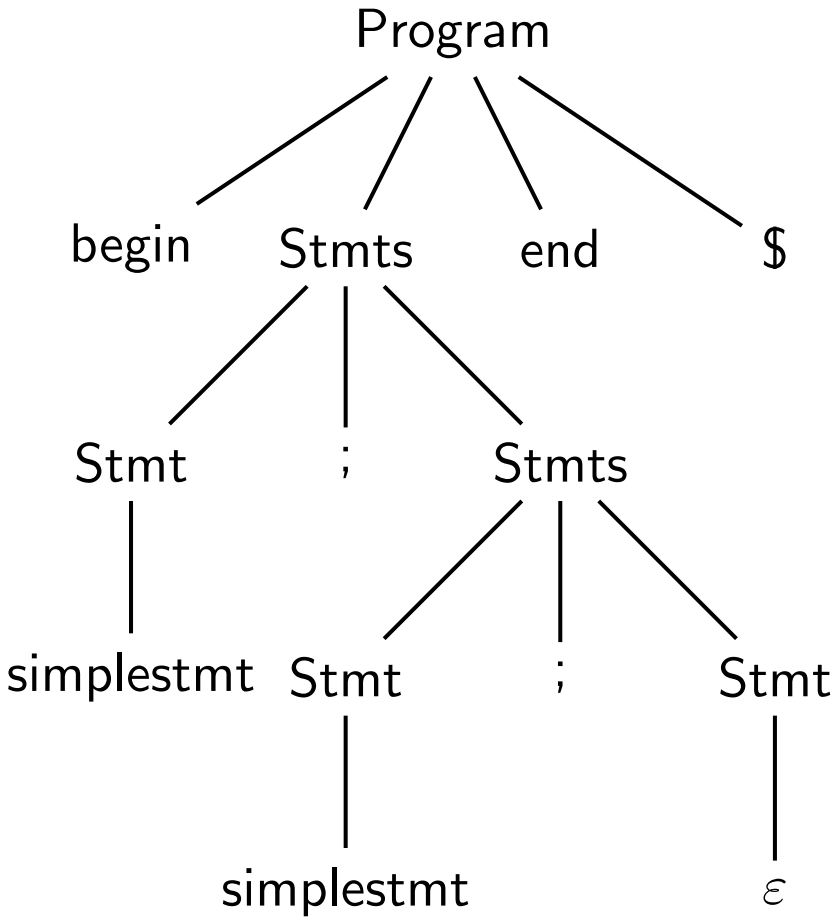
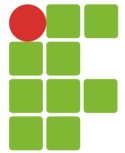
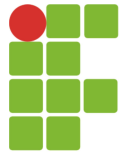



Figura: Expansão bottom-up



# Técnicas de análise sintática

- A grande parte das técnicas de análise sintática também são caracterizadas como LL (top-down) e LR (bottom-up).
- O primeiro símbolo indica como a sequência de tokens produzida pelo analisador léxico é encarada. Se for L, indica que a sequência é inspecionada da esquerda para a direita. Se for R, a sequência é inspecionada da direita para a esquerda.
- O segundo símbolo indica se derivações mais à esquerda (L) ou à direita (R) são produzidas.
- Portanto, um analisador LL é um analisador que processa a sequência de tokens da esquerda para direita enquanto busca realizar derivações mais à esquerda para verificar se a sequência está gramaticalmente correta.

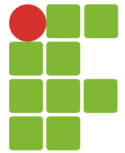


# Técnicas de Análise Sintática

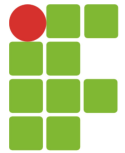
## LL($k$) ou LR($k$)

Denotamos por LL($k$) ou LR($k$) se $k$ é o número de símbolos de *l*ookahead utilizados para realizar a análise.

Esta quantidade diz respeito ao número de tokens, além do token atual, que o analisador utiliza para fazer suas escolhas. Analisadores LL(1) e LR(1) são os mais comuns.

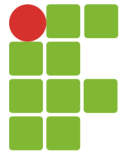


# Sumário



# Algoritmos

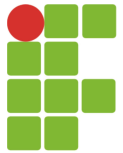
- Nesta seção examinaremos diversos algoritmos que atuam sobre gramáticas.
- Estes algoritmos são importantes para construção de um analisador sintático.



# Algoritmos

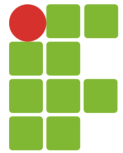
Para padronizar a discussão, relembremos os seguintes conceitos:

- Conjunto: coleção não ordenada de elementos distintos.
- Lista: coleção ordenada de elementos. Os elementos podem ocorrer múltiplas vezes.
- Iterador: enumera o conteúdo de uma lista ou conjunto.



# Algoritmos

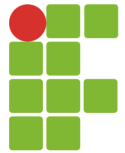
- Uma produção $A \to X_1 \ldots X_m$ de uma gramática é identificada através do não-terminal $A$ e do seu lado direito, que é representado através de uma lista de elementos (símbolos terminais e não terminais).
- A palavra vazia $\varepsilon$ não é representada explicitamente como um símbolo. Uma produção do tipo $A \to \varepsilon$ tem como o seu lado direito a lista vazia.



# Algoritmos

## Operações Básicas

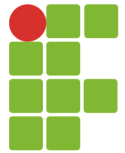
- GRAMMAR(S): cria uma nova gramática com símbolo de início $S$.
- ADD-PRODUCTION(A, rhs): cria uma nova regra de produção com o não-terminal $A$ como LHS e $rhs$ como sendo a sequência de símbolos a ser adiconada na lista da regra de produção. Retorna um descritor para a regra de produção.
- ADD-NONTERMINAL(A): adiciona o não-terminal $A$ ao conjunto de não terminais e retorna um descritor para o não-terminal adicional. Caso $A$ esteja no conjunto de terminais, um erro é reportado.



# Algoritmos

## Operações Básicas

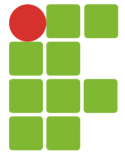
- ADD-TERMINAL(x): adiciona $x$ ao conjunto de terminais e retorna um descritor para o terminal adicionado. Um erro é reportado se $x$ se encontra no conjunto de não-terminais.
- IS-TERMINAL(X): retorna verdadeiro se $X$ é um terminal, e falso caso contrário.



# Algoritmos

## Operações Básicas

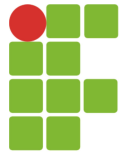
- TERMINALS(): retorna um iterador para o conjunto de terminais.
- PRODUCTIONS(): retorna um iterador para cada produção, sem uma ordem específica.
- NONTERMINALS(): retorna um iterador para o conjunto de não-terminais.



# Algoritmos

## Operações Básicas

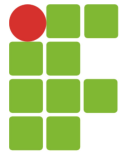
- $\textsc{rhs}(\mathrm{p})$: retorna um iterador para o lado direito de uma regra de produção descrita por $p$.
- $\textsc{lhs}(\mathrm{p})$: retorna o não-terminal da regra de produção descrita por $p$.
- $\textsc{productions-for}(\mathrm{A})$: retorna um iterador que visita cada produção que tenha como lado esquerdo o não-terminal $A$.



# Algoritmos

## Operações Básicas

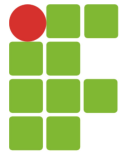
- $\textsc{rhs}(\mathrm{p})$: retorna um iterador para o lado direito de uma regra de produção descrita por $p$.
- $\textsc{lhs}(\mathrm{p})$: retorna o não-terminal da regra de produção descrita por $p$.
- $\textsc{productions-for}(\mathrm{A})$: retorna um iterador que visita cada produção que tenha como lado esquerdo o não-terminal $A$.



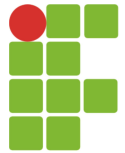
# Algoritmos

## Operações Básicas

- $\textsc{occurrences}(\mathrm{X})$: retorna um iterador que visita cada ocorrência de $X$ no lado direito de todas as regras. Em outras palavras, retorna um iterador para todos os pares $(p, i)$, em que $p$ é o descritor de uma produção em que $X$ ocorre como $i$-ésimo símbolo do lado direito.
- $\textsc{production}(\mathrm{O})$: retorna um descritor para a regra $A \to \alpha$, em que $\alpha$ contém uma ocorrência $O$ de algum símbolo do vocabulário da gramática.
- $\textsc{tail}(\mathrm{p}, \mathrm{i})$: Dada uma regra de produção $p$ que descreve $A \to X_1 \ldots X_m$, retorna $X_{i+1} \ldots X_m$.

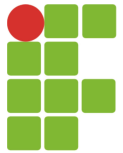


# Sumário



# Derivação da string vazia

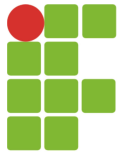
- Um dos algoritmos mais comuns sobre gramáticas é verificar se um não-terminal deriva uma string vazia.
- Não é trivial, pois a derivação pode levar mais de um passo.
- Exemplo: $\mathrm{A} \Rightarrow \mathrm{BCD} \Rightarrow \mathrm{BC} \Rightarrow \mathrm{B} \Rightarrow \varepsilon$



# Derivação da string Vazia

A ideia do algoritmo é seguinte:

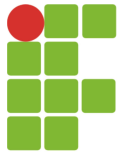
1. Primeiro marcamos com falso todos os símbolos nao-terminais para indicar que nenhum deles deriva $\varepsilon$.
2. Depois verificamos, para cada produção, se seu lado direito corresponde à $\varepsilon$. Se for o caso, marcamos o não-terminal associado como verdadeiro e o incluímos em uma fila. Caso contrário, guardamos em uma variável $\mathrm{cont}[p]$ o tamanho da sequência do lado direito da produção.



# Derivação da string Vazia

A ideia do algoritmo é seguinte:

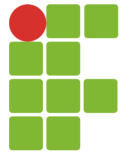
3. Enquanto a fila não estiver vazia, retiramos um não-terminal $X$ da fila e, para cada produção $p$ em que $X$ ocorre do lado direito, decrementamos $\text{cont}[p]$ de $1$. Se $\text{cont}[p]$ chegou a zero, é porque o não-terminal associado àquela produção também deriva a string vazia e ele deve ser incluído na fila.
4. Ao final do processo, todas os não-terminais que derivam $\varepsilon$ estarão marcados com verdadeiro.



# Derivação da string vazia

**Algorithm 1:** DERIVES-EMPTY-STRING()

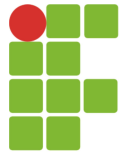
```
1  foreach A ∈ NONTERMINALS() do
     symbolDerivesEmpty[A] ← False
2  foreach p ∈ PRODUCTIONS() do
3      cont[p] ← |RHS(p)|
4      CHECK-EMPTY(p)
5  while not Q.EMPTY() do
6      X ← Q.POP()
7      foreach occ ∈ OCCURRENCES(X) do
8          p ← PRODUCTION(occ)
9          count[p] ← count[p] − 1
10         CHECK-EMPTY(p)
```



# Derivação da string vazia

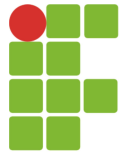
**Algorithm 2:** CHECK-EMPTY(p)

```
1 if( cont[p] = 0 )
2     A ← LHS(p)
3     if( not symbolDerivesEmpty[A] )
4         symbolDerivesEmpty[A] ← True
5         Q.PUSH(A)
```



# Derivação da string vazia

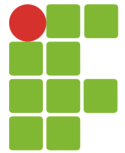
- Podemos adaptar o algoritmo anterior para incluir um vetor que indica, para cada produção, se ela deriva ou não $\varepsilon$.
- Basta manter uma variável $\mathrm{ruleDerivesEmpty}$ e marcar a regra como verdadeira quando o contador para a regra chegar a $0$.



# Derivação da string vazia

**Algorithm 3:** DERIVES-EMPTY-STRING()

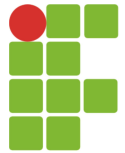
```
1  foreach A ∈ NONTERMINALS() do
     symbolDerivesEmpty[A] ← False
2  foreach p ∈ PRODUCTIONS() do
3      ruleDerivesEmpty[p] ← False
4      cont[p] ← |RHS(p)|
5      CHECK-EMPTY(p)
6  while not Q.EMPTY() do
7      X ← Q.POP()
8      foreach occ ∈ OCCURRENCES(X) do
9          p ← PRODUCTION(occ)
10         count[p] ← count[p] − 1
11         CHECK-EMPTY(p)
```



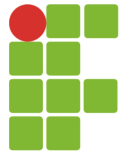
# Derivação da string vazia

**Algorithm 4:** CHECK-EMPTY(p)

```
1 if( cont[p] = 0 )
2     ruleDerivesEmpty[p] ← True
3     A ← LHS(p)
4     if( not symbolDerivesEmpty[A] )
5         symbolDerivesEmpty[A] ← True
6         Q.PUSH(A)
```



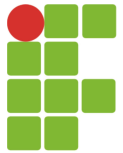
# Sumário



# FIRST

- Um conjunto muito utilizado pelos analisadores sintáticos é o $\mathrm{First}$
- Corresponde ao conjunto de todos os símbolos terminais que iniciam uma dada forma sentencial derivável de uma string $\alpha$.
- Formalmente:

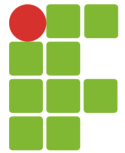
$$\mathrm{First}(\alpha) = \{a \in \Sigma | \alpha \Rightarrow^* a\beta\}$$



## FIRST

$\mathrm{First}(\alpha)$ pode ser computado da seguinte forma:

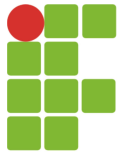
- Se $\alpha$ começa com um símbolo $a \in \Sigma$, então $\mathrm{First}(\alpha) = \{a\}$.
- Caso contrário, $\alpha$ começa com um não-terminal, digamos $X$. Devemos computar, recursivamente $\mathrm{First}(\mathrm{rhs})$ para cada $\mathrm{rhs}$, lado direito de uma produção com o não-terminal $X$. Cada um desses conjuntos integrará $\mathrm{First}(\alpha)$.
- Se $X$ ainda deriva $\varepsilon$, é necessário aplicar, recursivamente o procedimento para o símbolo que sucede $X$ em $\alpha$.
- Temos que ter um cuidado de marcar os não-terminais como visitados para evitar um loop infinito.



# FIRST

**Algorithm 5:** FIRST($\alpha$)

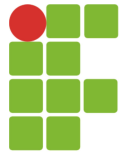
1 **foreach** $A \in$ NONTERMINALS() **do**
2 visited[A] $\leftarrow$ False
3 **return** INTERNAL-FIRST($\alpha$)



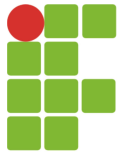
# FIRST

**Algorithm 6:** INTERNAL-FIRST(X$\beta$)

```
1  if( Xβ = [ ] ) return ∅
2  if( IS-TERMINAL(X) ) return {X}
3  ans ← ∅
4  if( not visited[X] )
5  |   visited[X] ← True
6  |   foreach p ∈ PRODUCTIONS-FOR(X) do
7  |   |   rhs ← RHS(p)
8  |   |   ans ← ans ∪ INTERNAL-FIRST(rhs)
9  if( symbolDerivesEmpty[X] )
10 |   ans ← ans ∪ INTERNAL-FIRST(β)
11 return ans
```



# Sumário

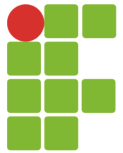


# FOLLOW

- Outra operação muito importante nos analisadores sintáticos é determinar o conjunto de terminais que se encontram imediatamente a direita de um não-terminal em alguma forma setencial.
- Estamos interessados em quais terminais podem ser gerados a partir de uma expansão de um não-terminal qualquer.
- Em outras palavras:

$$\text{Follow}(\text{A}) = \{b \in \Sigma \mid \text{S} \Rightarrow^{+} \alpha \text{A} b \beta\}$$

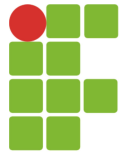
- Esta operação está sempre definida, pois as gramáticas incluem o símbolo especial \$ de fim de entrada. Qualquer não-terminal, com exceção de $S$, deve ser seguido por algum não-terminal.



# FOLLOW

Para computar $\mathrm{Follow(A)}$, realizamos o seguinte:

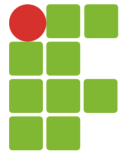
- Para toda ocorrência $(p, i)$ de $A$ no lado direito de uma regra de produção, computamos $\textsc{first}(\textsc{tail}(\mathrm{p}, \mathrm{i}))$. Pela definição de $\mathrm{First}$, os símbolos achados nesse processo devem estar em $\mathrm{Follow(A)}$.
- Caso a cauda de cada ocorrência de $A$ derive uma string vazia, então realizamos o procedimento recursivamente para o não-terminal que derivou essa ocorrência de $A$. Isto procede pois se temos $\mathrm{X} \Rightarrow^{+} \alpha A \beta$ , $\beta \Rightarrow^{+} \varepsilon$, e $\mathrm{S} \Rightarrow^{+} \gamma X \delta$, os símbolos terminais que seguirão de $\mathrm{X}$ também seguirão de $\mathrm{A}$.
- Para evitar um loop infinito, marcamos cada não-terminal visitado.



# FOLLOW

**Algorithm 7:** FOLLOW(A)

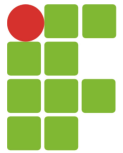
```
1 foreach A ∈ NONTERMINALS() do
2     visited[A] ← False
3 return INTERNAL-FOLLOW(A)
```



# FOLLOW

**Algorithm 8:** INTERNAL-FOLLOW(A)

```
1 ans ← ∅
2 if( not visited[A] )
3 |   visited[A] ← True
4 |   foreach (p, i) ∈ OCCURRENCES(A) do
5 |   |   ans ← ans ∪ FIRST(TAIL(p, i))
6 |   |   if( ALL-DERIVE-EMPTY(TAIL(p, i)) )
7 |   |   |   lhs ← LHS(PRODUCTION(p, i))
8 |   |   |   ans ← ans ∪ INTERNAL-FOLLOW(lhs)
9 return ans
```



# FOLLOW

**Algorithm 9:** ALL-DERIVE-EMPTY($\gamma$)

```
1 foreach X ∈ γ do
2     if( IS-TERMINAL(X) or not symbolDerivesEmpty[X] )
3         return False
4 return True
```