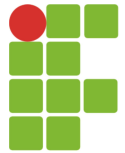
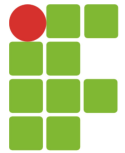
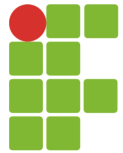


Sumário

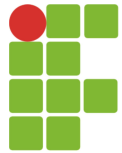


Sumário



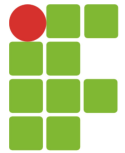
Introdução

- Agora que estudamos as gramáticas livres de contexto e os algoritmos relevantes sobre elas para realizar a análise sintática, chegou a hora de estudar a análise propriamente dita.



Introdução

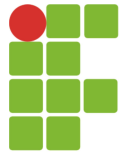
- Dada uma gramática, podemos gerar palavras a partir de sua definição e regras de produção.
- Contudo, o processo reverso não é tão simples. Isto é, dada uma palavra, como verificar se ela pode ser gerada pela gramática?
- Essencialmente essa é a pergunta que um analisador sintático deve responder. Ele recebe como entrada uma palavra, o programa, e deve dizer se aquele programa obedece as regras da gramática, isto é, se o programa está sintaticamente correto.
- Este problema é conhecido como **parsing problem**.



Introdução

Nesta aula estamos interessados com uma abordagem de análise sintática conhecida como top-down.

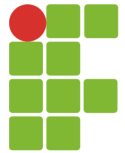
- Ela é conhecida por este nome pois o processo de análise começa do símbolo inicial da gramática e constrói uma árvore sintática a partir dessa raiz até chegar nas folhas.
- Ela é uma abordagem **preditiva**, pois o analisador deve prever qual regra aplicar para produzir a derivação correta.
- É LL(k). A entrada é processada da esquerda para direita (daí vem o primeiro L) e o analisador produz derivações mais à esquerda (daí vem o segundo L). Na análise, k símbolos de *lookahead* são utilizados para tomar as decisões.
- É descente recursiva: analisadores baseados nesta abordagem podem ser construídos através de uma série de procedimentos recursivos.



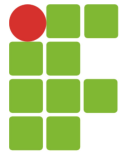
Introdução: análise sintática

Abordaremos duas categorias de analisadores LL (top-down):

- Analisador descendente recursivo.
- Analisador LL guiado por tabelas.



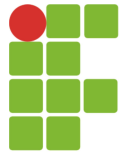
Sumário



Gramáticas LL(k)

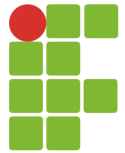
Essencialmente, um analisador para uma gramática LL(k):

- Possui um procedimento para cada não-terminal A. Este procedimento é encarregado de aplicar umaderivação ao escolher uma das regras de produção que tenham como A o não-terminal do lado esquerdo.
- Para escolher a produção adequada, o analisador examina os próximos k tokens, símbolos terminais, da entrada. O conjunto **predict** para uma produção $\mathrm{A} \rightarrow \alpha$ é o conjunto de tokens que causa a aplicação daquela regra. Para computar **predict** é necessário examinar o lado direito da produção. Pode ser que outras produções participem da computação do conjunto **predict** de uma produção.



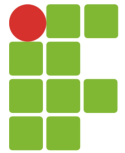
Gramáticas LL(k)

- Os k tokens são os símbolos de *lookahead*.
- Se é possível construir um analisador LL(k)para uma gramática que reconheça a linguagem gerada pela gramática, a gramática é dita LL(k).
- Um analisador LL(k)pode inspecionar os próximos k símbolos para decidir qual regra de produção aplicar.



Gramáticas LL(k)

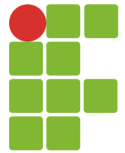
- Para escolher qual regra de produção aplicar, o analisador utiliza uma função $\mathrm{predict_k}(p)$
- Esta função considera uma regra de produção p e computa o conjunto de todas as palavras de tamanho k que predizem a aplicação da regra p.
- No caso em que $k = 1$, a função é simplesmente chamada de $\mathrm{predict}(p)$.



Gramáticas LL(k)

- Considere a entrada $\alpha a \beta \in \Sigma^*$ e que o nosso analisador seja LL(1)
- Suponha que o analisador construiu uma derivação $\mathrm{S} \Rightarrow^*_{\mathrm{lm}} \alpha \mathrm{A} \mathrm{Y}_1 \ldots \mathrm{Y}_\mathrm{n}$. Ou seja, o analisador já conseguiu consumir a subpalavra α da entrada.
- O analisador agora precisa encontrar alguma produção de A que comeÇe com o símbolo a, visto que é o próximo símbolo a ser consumido.
- Em outras palavras, queremos computar o seguinte conjunto:

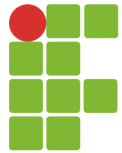
$$P = \{p \in \textsc{productions-for}(\mathrm{A}) \mid a \in \mathrm{predict}(p)\}$$



Gramáticas LL(k)

$$P = \{p \in \text{PRODUCTIONS-FOR}(\text{A}) \mid a \in \text{predict}(p)\}$$

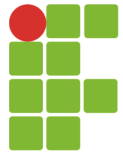
- Se $P = \emptyset$, então não há produção para A que satisfaça a entrada. A análise não deve continuar e um erro de sintaxe deve ser reportado, com a sendo o símbolo que causou o problema. As produções de A podem ser úteis para construir mensagens de erros mais úteis, indicando inclusive quais símbolos poderiam ser processados.



Gramáticas LL(k)

$$P = \{p \in \text{PRODUCTIONS-FOR}(\text{A}) \mid a \in \text{predict}(p)\}$$

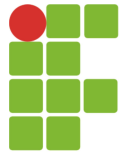
- Se P contém mais de uma produção, a análise deve continuar, mas um comportamento **não-determinístico** seria requerido para seguir, independentemente, cada produção de P. Por questões de eficiência, seria ideal que os analisadores sempre fossem **determinísticos**, portanto, os analisadores devem assegurar que este caso não ocorra.



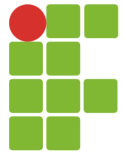
Gramáticas LL(k)

$$P = \{p \in \text{PRODUCTIONS-FOR}(\text{A}) \mid a \in \text{predict}(p)\}$$

- O terceiro caso é em que P é um conjunto unitário, isto é, apenas possui apenas uma regra de produção. Neste caso, uma derivação mais à esquerda pode ser produzida ao aplicar a única regra dada por P.

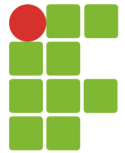


Sumário



Gramáticas LL(k)

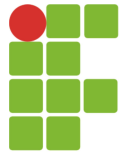
- Como implementar a função $\mathrm{predict}(p)$?
- Considere a produção $p : \mathrm{A} \to \mathrm{X_1} \ldots \mathrm{X_m}$, $m \geq 0$. Quando $m = 0$, a produção é do tipo $\mathrm{A} \to \varepsilon$
- Assim, o conjunto de símbolos previstos por uma produção são:
  - O conjunto de símbolos terminais que iniciam em uma derivação de $\mathrm{X_1} \ldots \mathrm{X_m}$.
  - O conjunto de símbolos que sucedam A em alguma forma sentencial, caso $\mathrm{A} \Rightarrow^*_{\mathrm{lm}} \varepsilon$.



Gramáticas LL(k)

Algorithm 1: PREDICT(p)

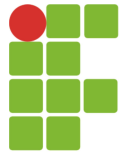
```
1 ans ← FIRST(RHS(p))
2 if( ruleDerivesEmpty[p] )
3 |   A ← LHS(p)
4 |   ans ← ans ∪ FOLLOW(A)
5 return ans
```



Predict

Tome a seguinte gramática:

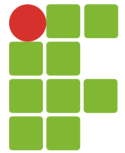
$$
\begin{array}{ll}
1 & \mathrm{S} \rightarrow \mathrm{AC\$} \\
2 & \mathrm{C} \rightarrow \mathrm{c} \\
3 & \mathrm{C} \rightarrow \varepsilon \\
4 & \mathrm{A} \rightarrow \mathrm{aBCd} \\
5 & \mathrm{A} \rightarrow \mathrm{BQ} \\
6 & \mathrm{B} \rightarrow \mathrm{bB} \\
7 & \mathrm{B} \rightarrow \varepsilon \\
8 & \mathrm{Q} \rightarrow \mathrm{q} \\
9 & \mathrm{Q} \rightarrow \varepsilon
\end{array}
$$



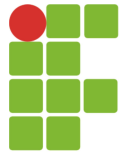
Predict

O conjunto predict para cada regra é:

Rule Number	A	$X_1 \ldots X_m$	First($X_1 \ldots X_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

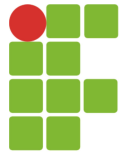


Sumário



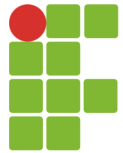
Gramáticas LL(1)

- Em uma gramática LL(1), as regras de produção para cada não-terminal A devem possuir conjuntos disjuntos de $\mathrm{predict}$.
- A maioria das linguagens de programação possuem uma gramática LL(1).
- Contudo, nem todas as CFGs são LL(1):
  - Algumas gramáticas necessitam de um *lookahead* maior, isto é, a gramática é LL(k)para $k > 1$.
  - A gramática pode ser ambígua, fazendo com que seja impossível obter qualquer analisador sintático determinístico.



Gramáticas LL(1)

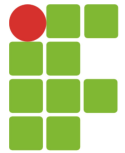
- Para determinar se uma gramática é LL(1), basta verificar se os conjuntos $\mathrm{predict}$ gerados para um dado não-terminal são disjuntos.



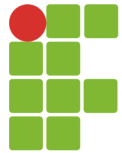
Gramáticas LL(1)

Algorithm 2: IS-LL1(G)

```
1 foreach A ∈ NON-TERMINALS() do
2     predictSet ← ∅
3     foreach p ∈ PRODUCTIONS-FOR(A) do
4         if( PREDICT(p) ∩ predictSet ≠ ∅ )
5             return False
6         predictSet ← predictSet ∪ PREDICT(p)
7 return True
```



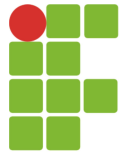
Sumário



Analisadores descendentes recursivos LL(1)

Antes de iniciar a disucussão sobre os analisadores descendentes recursivos, vamos assumir que a sequência de tokens, denotada por ts, oferece os seguintes métodos:

- ts.PEEK(): examina o próximo token da entrada sem avançar.
- ts.ADVANCE(): avança a entrada em um token.

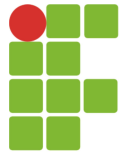


Analisadores descendentes recursivos

Com posse desses métodos, podemos implementar a função $\text{MATCH}(\text{ts}, \text{token})$, que verifica se um token específico se encontra na posição atual da sequência de tokens:

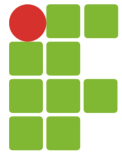
Algorithm 3: $\text{MATCH}(\text{ts}, \text{token})$

```
1 if( ts.PEEK() = token )
2     ts.ADVANCE()
3 else
4     REPORT-ERROR("Expected : ", token)
```



Analisadores descendentes recursivos

- A estrutura de um analisador descendente recursivo é padronizada.
- Todo não-terminal terá um procedimento associado.
- Se existem n regras, $p_1, \ldots, p_n$ associadas a um não-terminal, verificamos se o token atual está no conjunto $\mathrm{predict}(p_i)$, se sim, executamos o código correspondente à p_i. Caso contrário, analisamos a próxima produção, p_{i+1}.
- Se nenhuma regra é aplicável, um erro de sintaxe deve ser reportado.



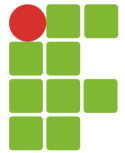
Analisadores descendentes recursivos

Algorithm 4: $\mathrm{A}(\mathrm{ts})$

1 **if**($\mathrm{ts.PEEK}() \in \mathrm{PREDICT}(\mathrm{p_1})$)
   `// Código para` p_1
2 **else if**($\mathrm{ts.PEEK}() \in \mathrm{PREDICT}(\mathrm{p_2})$)
   `// Código para` p_2

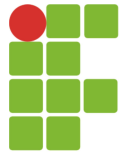
⋮

3 **else if**($\mathrm{ts.PEEK}() \in \mathrm{PREDICT}(\mathrm{p_n})$)
   `// Código para` p_n
4 **else**
   `// Erro de sintaxe`



Analisadores descendentes recursivos

- O código relacionado a cada p_i depende da forma da regra.
- Se a regra p_i tem como lado direito $\mathrm{X_1} \ldots \mathrm{X_m}$, $m \geq 0$ temos as seguintes situações:
  - Se $m = 0$, então a regra é do tipo $\mathrm{A} \rightarrow \varepsilon$. Neste caso o código para p_i é simplesmente terminar o procedimento para A.
  - Se $\mathrm{X_i}$ é um terminal, então uma chamada a $\textsc{match}(\mathrm{ts}, \mathrm{X_i})$ é realizada. Em caso de sucesso, $\mathrm{X_i}$ é consumido da entrada, caso contrário um erro é emitido.
  - Se $\mathrm{X_i}$ é um não-terminal, uma chamada para o procedimento $X_i(\mathrm{ts})$ é realizada.



Analisadores descendentes recursivos

Tome a seguinte gramática. Como ficaria o código do analisador?

1 $S \rightarrow AC\$$

2 $C \rightarrow c$

3 $C \rightarrow \varepsilon$

4 $A \rightarrow aBCd$

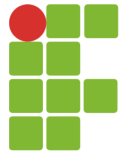
5 $A \rightarrow BQ$

6 $B \rightarrow bB$

7 $B \rightarrow \varepsilon$

8 $Q \rightarrow q$

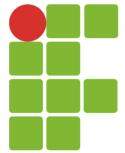
9 $Q \rightarrow \varepsilon$



Analisadores descendentes recursivos

Algorithm 5: $\mathrm{S}(\mathrm{ts})$

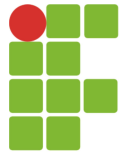
```
1 if( ts.PEEK() ∈ {a, b, q, c, $} )
2     A(ts)
3     C(ts)
4     MATCH(ts, $)
```



Analisadores descendentes recursivos

Algorithm 6: $\mathrm{C}(\mathrm{ts})$

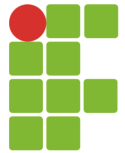
```
1 if( ts.PEEK() ∈ {c} )
2     MATCH(ts, c)
3 else if( ts.PEEK() ∈ {d, $} )
4     return
```



Analisadores descendentes recursivos

Algorithm 7: A(ts)

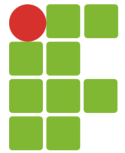
```
1 if( ts.PEEK() ∈ {a} )
2     MATCH(ts, a)
3     B(ts)
4     C(ts)
5     MATCH(ts, d)
6 else if( ts.PEEK() ∈ {b, q, c, $} )
7     B(ts)
8     Q(ts)
```



Analisadores descendentes recursivos

Algorithm 8: $\mathrm{B}(\mathrm{ts})$

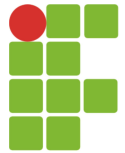
```
1 if( ts.PEEK() ∈ {b} )
2     MATCH(ts, b)
3     B(ts)
4 else if( ts.PEEK() ∈ {q, c, d, $} )
5     return
```



Analisadores descendentes recursivos

Algorithm 9: $\mathrm{B}(\mathrm{ts})$

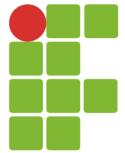
```
1 if( ts.PEEK() ∈ {b} )
2     MATCH(ts, b)
3     B(ts)
4 else if( ts.PEEK() ∈ {q, c, d, $} )
5     return
```



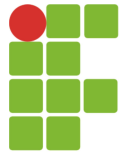
Analisadores descendentes recursivos

Algorithm 10: $\mathrm{Q}(\mathrm{ts})$

```
1 if( ts.PEEK() ∈ {q} )
2     MATCH(ts, q)
3 else if( ts.PEEK() ∈ {c, $} )
4     return
```

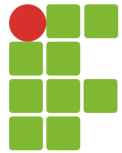


Sumário



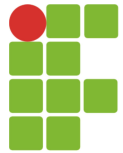
LL(1) guiado por tabelas

- O tamanhho de um analisador sintático descendente recursivo cresce de maneira proporcional ao tamanho da gramática.
- Além disso, é um processo repetivo e mecânico, mas, felizmente, automatizável.
- Podemos criar um analisaro LL(1)baseado em tabelas.
- O resultado é um código mais compacto e eficiente, visto que não é necessário realizar diversas chamadas de função.



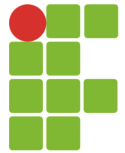
LL(1) guiado por tabelas

- Utilizaremos uma pilha.
- Sempre que um terminal está no topo da pilha, a função $\textsc{match}()$ é chamada e, em caso de sucesso, o terminal é desempilhado.
- Se o topo da pilha é um não-terminal A, obtemos a regra $\mathrm{A} \rightarrow \mathrm{X_1} \ldots \mathrm{X_m}$ adequada através de uma consulta a uma tabela e inserimos os símbolos $\mathrm{X_m} \ldots \mathrm{X_1}$ na pilha (ordem inversa).
- O processo termina quando o símbolo terminal $\$$ é consumido.
- Caso uma regra apropriada não exista ou não seja possível consumir um símbolo não terminal, um erro é reportado.



LL(1) guiado por tabelas

- A tabela deve ser preenchida antes do analisador sintático iniciar o seu processamento.
- É realizada de acordo com a função PREDICT(), conforme explorado no analisador descendente recursivo.

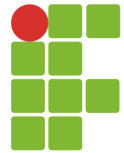


LL(1) guiado por tabelas

Preenchimento da tabela

Algorithm 11: FILL-TABLE(lltable)

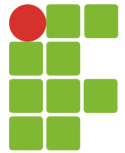
```
1 foreach A ∈ NONTERMINALS() do
2     foreach a ∈ TERMINALS() do
3         lltable[A][a] ← ⊥
4 foreach A ∈ NONTERMINALS() do
5     foreach p ∈ PRODUCTIONS-FOR(A) do
6         foreach a ∈ PREDICT(p) do
7             lltable[A][a] ← p
```



LL(1) guiado por tabelas

Algorithm 12: LL1-PARSER(ts)

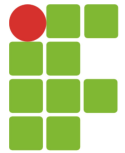
```
1  stck ← STACK()
2  stck.PUSH(S)
3  accept ← False
4  while not accept do
5      top ← stck.TOP()
6      if( IS-TERMINAL(top) )
7          MATCH(ts, top)
8          if( top = $ )
9              accept ← True
10             stck.POP()
11     else
12         p ← lltable[top][ts.PEEK()]
13         if( p = ⊥ )
14             ERROR("syntax error, no producton applicable")
15         else
16             APPLY(stck, p)
```



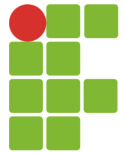
LL(1) guiado por tabelas

Algorithm 13: APPLY(stck, p)

```
1 stck.POP()
2 rhs ← RHS(p)
3 for( i ← |rhs| − 1; i ≥ 0 ; i ← i − 1 )
4   stck.PUSH(rhs[i])
```

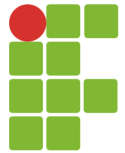


Sumário



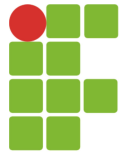
Projeto de gramáticas LL(1)

- Pode ser um pouco difícil para projetistas de compiladores iniciantes criar gramáticas LL(1), visto que essas gramáticas requerem predições únicas utilizando apenas um símbolo de *lookahead*.
- Felizmente, uma parte dos conflitos das regras de predição que impedem uma gramática de ser LL(1)podem ser enquadradas em duas categorias:
  - Prefixos comuns.
  - Recursão à esquerda.

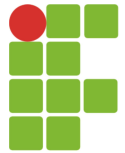


Projeto de gramáticas LL(1)

- É possível modificar a gramática de modo a eliminar esses conflitos, possivelmente tornando a gramática LL(1).
- Examinaremos essas transformações.

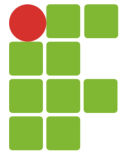


Sumário



Prefixos comuns

- Nesta categoria de conflitos enquadram-se as regras cujo lado direito compartilham um prefixo comum.
- Isto é, regras cujo lado direito começam com os mesmos símbolos.

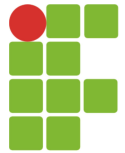


Prefixos comuns

Tome a seguinte gramática:

1. $\text{Stmt} \rightarrow \text{if Expr then StmtList endif}$
2. $\text{Stmt} \rightarrow \text{if Expr then StmtList else StmtList endif}$
3. $\text{StmtList} \rightarrow \text{StmtList ; Stmt}$
4. $\text{StmtList} \rightarrow \text{Stmt}$
5. $\text{Expr} \rightarrow \text{var} + \text{Expr}$
6. $\text{Expr} \rightarrow \text{var}$

Claramente as regras 1 e 2 não podem ser distinguidas com apenas um símbolo de *lookahead*.

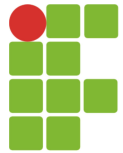


Prefixos comuns

Tome a seguinte gramática:

1. $\text{Stmt} \rightarrow \text{if Expr then StmtList endif}$
2. $\text{Stmt} \rightarrow \text{if Expr then StmtList else StmtList endif}$
3. $\text{StmtList} \rightarrow \text{StmtList ; Stmt}$
4. $\text{StmtList} \rightarrow \text{Stmt}$
5. $\text{Expr} \rightarrow \text{var} + \text{Expr}$
6. $\text{Expr} \rightarrow \text{var}$

Mesmo permitindo mais símbolos, o else que faz a distinção pode bem à direita.

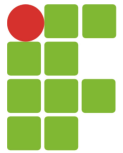


Prefixos comuns

Tome a seguinte gramática:

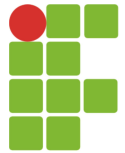
1. $\text{Stmt} \rightarrow \text{if Expr then StmtList endif}$
2. $\text{Stmt} \rightarrow \text{if Expr then StmtList else StmtList endif}$
3. $\text{StmtList} \rightarrow \text{StmtList ; Stmt}$
4. $\text{StmtList} \rightarrow \text{Stmt}$
5. $\text{Expr} \rightarrow \text{var} + \text{Expr}$
6. $\text{Expr} \rightarrow \text{var}$

O tamanho da StmtList pode ser arbitrariamente maior que qualquer constante k, inviabilizando qualquer analisador LL(k)



Prefixos comuns

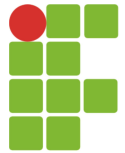
- Para eliminar prefixos comuns, usamos a fatoração deles.
- Se existirem duas regras $\mathrm{A} \rightarrow \alpha\beta_1$ e $\mathrm{A} \rightarrow \alpha\beta_2$, significa que existe um prefixo comum α entre elas.
- Criamos um **novo** não-terminal X e as seguintes regras de produção: $\mathrm{X} \rightarrow \beta_1$ e $\mathrm{X} \rightarrow \beta_2$.
- Por fim, colapsamos as duas regras originais $\mathrm{A} \rightarrow \alpha\beta_1$ e $\mathrm{A} \rightarrow \alpha\beta_2$ em uma única regra $\mathrm{A} \rightarrow \alpha\mathrm{X}$, eliminando o prefixo comum.



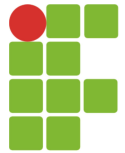
Prefixos comuns

A gramática modificada fica como:

1. $\text{Stmt} \rightarrow \text{if Expr then StmtList } V_1$
2. $V_1 \rightarrow \text{endif}$
3. $V_1 \rightarrow \text{else StmtList endif}$
4. $\text{StmtList} \rightarrow \text{StmtList ; Stmt}$
5. $\text{StmtList} \rightarrow \text{Stmt}$
6. $\text{Expr} \rightarrow \text{var } V_2$
7. $V_2 \rightarrow + \text{ Expr}$
8. $V_2 \rightarrow \varepsilon$

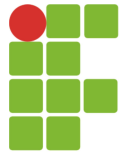


Sumário



Recursão à esquerda

- Uma produção é recursiva à esquerda se o seu símbolo do lado esquerdo é igual ao primeiro símbolo do lado direito. Exemplo: $\mathrm{A} \rightarrow \mathrm{ABC}$.
- É impossível construir um analisador LL(1) nesta situação, visto que o analisador descendente recursivo entraria em uma recursão infinita.
- Na implementação baseada em tabelas, os símbolos ABC seriam infinitamente inseridos na pilha.

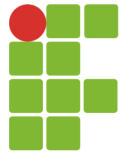


Eliminação da recursão à esquerda

- Suponha as seguintes regras:

$$\mathrm{A} \rightarrow \mathrm{A}\alpha \mid \beta$$

- Podemos dizer que $\mathrm{A} \Rightarrow^{+} \beta\alpha^{*}$.
- Para eliminar a recursão à esquerda, basta trocar a regra $\mathrm{A} \rightarrow \mathrm{A}\alpha \mid \beta$ pelas regras $\mathrm{A} \rightarrow \beta\mathrm{A}'$ e $\mathrm{A}' \rightarrow \alpha\mathrm{A}' \mid \varepsilon$, em que A' é um novo não-terminal.



Prefixos comuns

Tome a seguinte gramática:

1 $\text{Stmt} \rightarrow \text{if Expr then StmtList } V_1$

2 $V_1 \rightarrow \text{endif}$

3 $V_1 \rightarrow \text{else StmtList endif}$

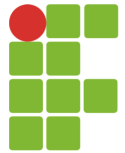
4 $\text{StmtList} \rightarrow \text{StmtList ; Stmt}$

5 $\text{StmtList} \rightarrow \text{Stmt}$

6 $\text{Expr} \rightarrow \text{var } V_2$

7 $V_2 \rightarrow + \text{ Expr}$

8 $V_2 \rightarrow \varepsilon$

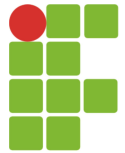


Prefixos comuns

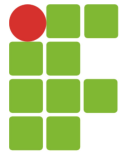
Eliminando a recursão à esquerda, temos:

1 $\text{Stmt} \rightarrow \text{if Expr then StmtList } V_1$
2 $V_1 \rightarrow \text{endif}$
3 $V_1 \rightarrow \text{else StmtList endif}$
4 $\text{StmtList} \rightarrow \text{Stmt } V_3$
5 $V_3 \rightarrow \text{StmtList ; } V_3$
6 $\text{Expr} \rightarrow \text{var } V_2$
7 $V_2 \rightarrow + \text{ Expr}$
8 $V_2 \rightarrow \varepsilon$

Que é uma gramática LL(1).

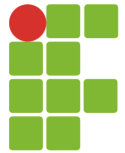


Sumário



Propriedades das gramáticas LL(1)

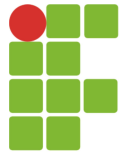
- Gramáticas LL(1)são extremamente interessantes do ponto de vista prático por possuirem propriedades desejáveis.



Propriedades das gramáticas LL(1)

Derivações mais à esquerda corretas

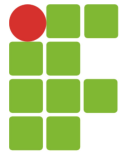
Um analisador LL(1) obtém derivações mais à esquerda. E elas são únicas, pois os conjutos preditos pelas regras considerando cada não-terminal são disjuntos.



Propriedades das gramáticas LL(1)

Gramáticas não ambíguas

Gramáticas LL(1) não são ambíguas, pois cada derivação mais à esquerda obtida é única. Em gramáticas ambíguas, uma mesma string pode ser produzida por duas ou mais derivações mais à esquerda. Ao comparar essas derivações, conclui-se que há um ponto em que pelo menos duas regras distintas, com o mesmo não-terminal do lado esquerdo, puderam ser aplicadas, gerando as derivações distintas.



Propriedades das gramáticas LL(1)

Eficiência

Os analisadores guiados por tableas das gramáticas LL(1) operam em tempo e espaço linear no tamanho da string de entrada.