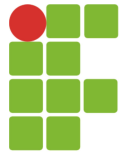
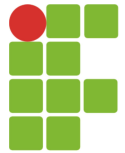
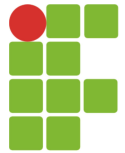


Sumário

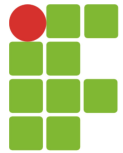


Sumário



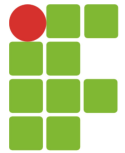
Introdução

- 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

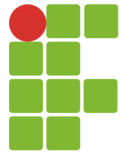
- Dada uma gramática, podemos gerar palavras a partir de sua definição e regras de produção.
- Contudo, o processo reverso não é tão simples. Isto é, dada uma palavra, como verificar se ela pode ser gerada pela gramática?
- Essencialmente essa é a pergunta que um analisador sintático deve responder. 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.

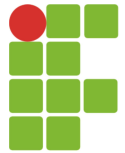
- Ela é conhecida por este nome pois o processo de análise começa do símbolo inicial da gramática e constrói uma árvore sintática a partir dessa raiz até chegar nas folhas.
- Ela é uma abordagem **preditiva**, pois o analisador deve prever qual regra aplicar para produzir a derivação correta.
- É LL(k). 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.



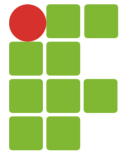
Introdução: análise sintática

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

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



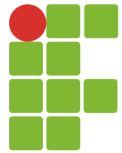
Sumário



Gramáticas LL(k)

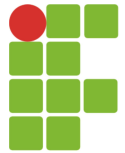
Essencialmente, um analisador para uma gramática LL(k):

- 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.



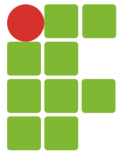
Gramáticas LL(k)

- 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)

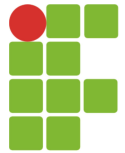
- Para escolher qual regra de produção aplicar, o analisador utiliza uma função $\mathrm{predict_k}(p)$
- Esta função considera uma regra de produção p e computa o conjunto de todas as palavras de tamanho k que predizem a aplicação da regra p.
- No caso em que $k = 1$, a função é simplesmente chamada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

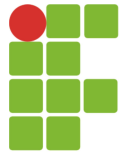
$$P = \{p \in \textsc{productions-for}(\mathrm{A}) \mid a \in \mathrm{predict}(p)\}$$



Gramáticas LL(k)

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

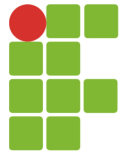
- Se $P = \emptyset$, então não há produção para A que satisfaça a entrada. A análise não deve continuar e um erro de sintaxe deve ser reportado, com a sendo o símbolo que causou o problema. As produções de A podem ser úteis para construir mensagens de erros mais úteis, indicando inclusive quais símbolos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



Gramáticas LL(k)

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

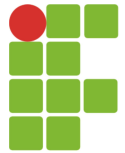
- Se P contém mais de uma produção, a análise deve continuar, mas um comportamento **não-determinístico** seria requerido para seguir, independentemente, cada produção de P. Por questões de eficiência, seria ideal que os analisadores sempre fossem **determinísticos**, portanto, os analisadores devem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



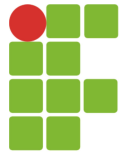
Gramáticas LL(k)

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

- 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.

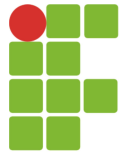


Sumário



Gramáticas LL(k)

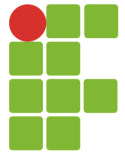
- Como implementar a função $\mathrm{predict}(p)$?
- Considere a produção $p : \mathrm{A} \rightarrow \mathrm{X_1} \ldots \mathrm{X_m}$, $m \geq 0$. Quando $m = 0$, a produção é do tipo $\mathrm{A} \rightarrow \varepsilon$
- Assim, o conjunto de símbolos previstos por uma produção são:
  - O conjunto 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)

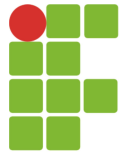
```
1 ans ← FIRST(RHS(p))
2 if( ruleDerivesEmpty[p] )
3 |   A ← LHS(p)
4 |   ans ← ans ∪ FOLLOW(A)
5 return ans
```



Predict

Tome a seguinte gramática:

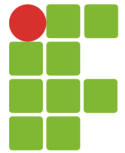
$$
\begin{aligned}
1 \quad & \mathrm{S} \rightarrow \mathrm{AC\$} \\
2 \quad & \mathrm{C} \rightarrow \mathrm{c} \\
3 \quad & \mathrm{C} \rightarrow \varepsilon \\
4 \quad & \mathrm{A} \rightarrow \mathrm{aBCd} \\
5 \quad & \mathrm{A} \rightarrow \mathrm{BQ} \\
6 \quad & \mathrm{B} \rightarrow \mathrm{bB}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



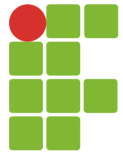
Predict

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,$

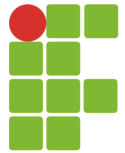


Sumário



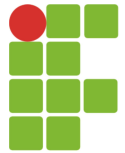
Gramáticas LL(1)

- 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)

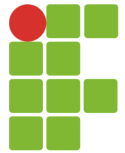
- Para determinar se uma gramática é LL(1), basta verificar se os conjuntos $\mathrm{predict}$ gerados para um dado não-terminal são disjuntos.



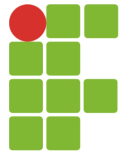
Gramáticas LL(1)

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
```



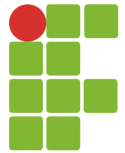
Sumário



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.

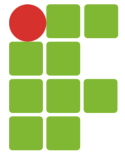


Analisadores descendentes recursivos

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:

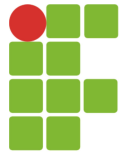
Algorithm 3: $\text{MATCH}(\text{ts}, \text{token})$

```
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.



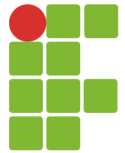
Analisadores descendentes recursivos

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

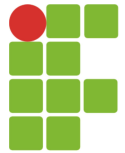
⋮

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



Analisadores descendentes recursivos

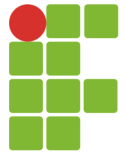
- O código relacionado a cada p_i depende da forma da regra.
- Se a regra p_i tem como lado direito $\mathrm{X_1} \ldots \mathrm{X_m}$, $m \geq 0$ temos as seguintes situações:
  - Se $m = 0$, então a regra é do tipo $\mathrm{A} \rightarrow \varepsilon$. Neste caso o código para p_i é simplesmente terminar 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?

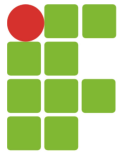
$$
\begin{array}{ll}
1 & \mathrm{S} \rightarrow \mathrm{AC\$} \\
2 & \mathrm{C} \rightarrow \mathrm{c} \\
3 & \mathrm{C} \rightarrow \varepsilon \\
4 & \mathrm{A} \rightarrow \mathrm{aBCd} \\
5 & \mathrm{A} \rightarrow \mathrm{BQ} \\
6 & \mathrm{B} \rightarrow \mathrm{bB} \\
7 & \mathrm{B} \rightarrow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



Analisadores descendentes recursivos

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

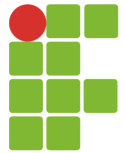
```
1 if( ts.PEEK() ∈ {a, b, q, c, $} )
2     A(ts)
3     C(ts)
4     MATCH(ts, $)
```



Analisadores descendentes recursivos

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

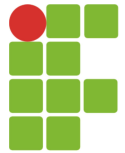
```
1 if( ts.PEEK() ∈ {c} )
2     MATCH(ts, c)
3 else if( ts.PEEK() ∈ {d, $} )
4     return
```



Analisadores descendentes recursivos

Algorithm 7: A(ts)

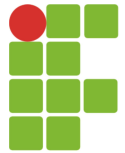
```
1 if( ts.PEEK() ∈ {a} )
2     MATCH(ts, a)
3     B(ts)
4     C(ts)
5     MATCH(ts, d)
6 else if( ts.PEEK() ∈ {b, q, c, $} )
7     B(ts)
8     Q(ts)
```



Analisadores descendentes recursivos

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

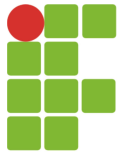
```
1 if( ts.PEEK() ∈ {b} )
2     MATCH(ts, b)
3     B(ts)
4 else if( ts.PEEK() ∈ {q, c, d, $} )
5     return
```



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
```