



Judson Santos Santiago

Transformação de Gramáticas

Compiladores

Introdução

- As **gramáticas** descrevem a sintaxe das linguagens

$$\begin{array}{l} \text{exp} \\ r \end{array} \quad \begin{array}{l} \text{expr} + \text{expr} \\ | \\ \text{expr} * \text{expr} \\ | \\ -\text{expr} \\ | \\ (\text{expr}) \\ | \\ \text{id} \end{array}$$

$$\begin{array}{l} \text{expr} \Rightarrow -\text{expr} \\ \Rightarrow -(\text{expr}) \\ \Rightarrow -(\text{expr} + \text{expr}) \\ \Rightarrow -(\text{id} + \text{expr}) \\ \Rightarrow -(\text{id} + \text{id}) \end{array}$$

- Através de uma gramática é possível **verificar** se uma sequência de caracteres é uma **cadeia válida** da linguagem
 - Processo conhecido por **derivação**

Introdução

- Existem dois tipos de derivação:
 - Derivação mais à esquerda
 - Derivação mais à direita
- Uma derivação pode ser representada por uma árvore
- Gramáticas podem ser ambíguas
 - Uma cadeia possui mais de uma árvore de derivação
 - Ou mais de uma derivação mais à esquerda
 - Ou mais de uma derivação mais à direita

Introdução

- Nem toda gramática é tratável pelos métodos de análise sintática mais eficientes:
 - Análise descendente
 - Análise ascendente
- Existem algumas transformações que podem ser utilizadas para adequar as gramáticas:
 - Remoção de ambiguidades
 - Eliminação da recursão à esquerda
 - Fatoração à esquerda

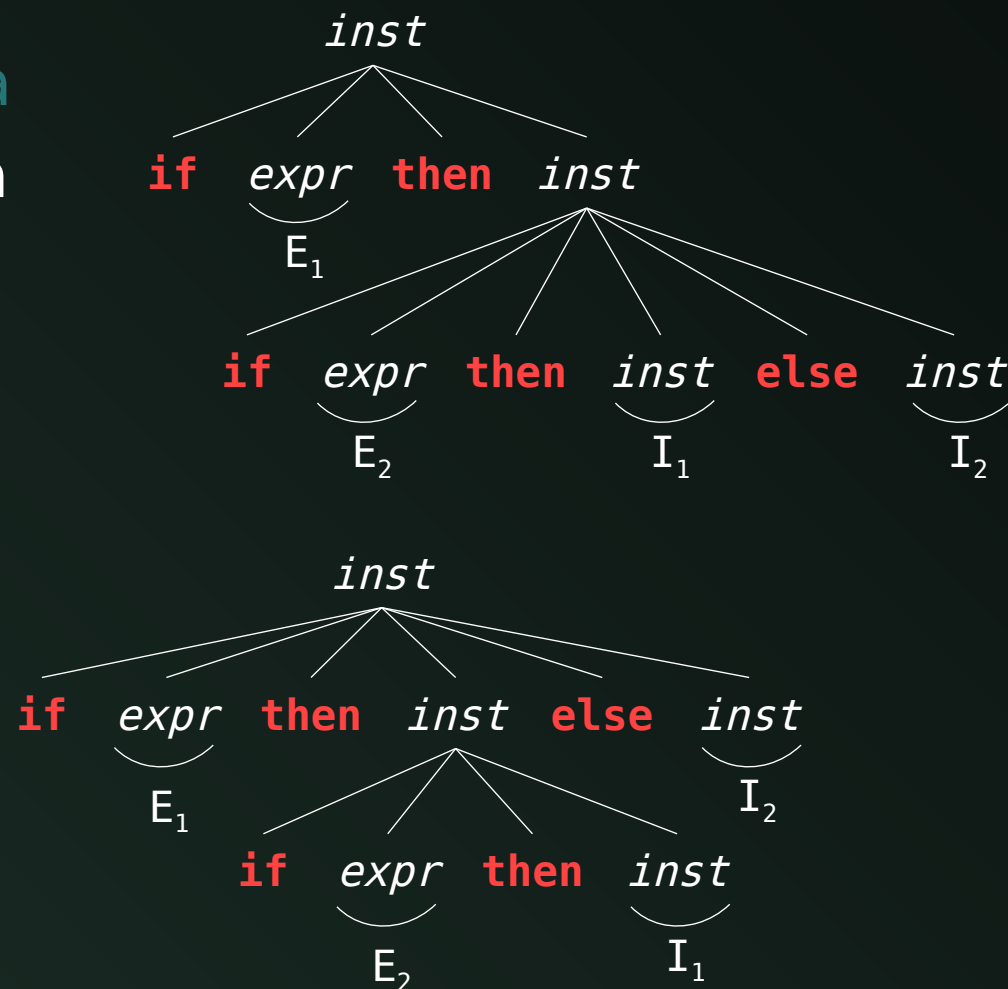
Eliminando Ambiguidade

- Às vezes uma gramática ambígua pode ser reescrita para eliminar a ambiguidade

inst ::= **if** *expr* **then** *inst*
 | **if** *expr* **then** *inst* **else** *inst*
 | **other**

A gramática é ambígua pois a cadeia abaixo possui duas árvores de derivação:

if *E*₁ **then** **if** *E*₂ **then** *I*₁ **else** *I*₂



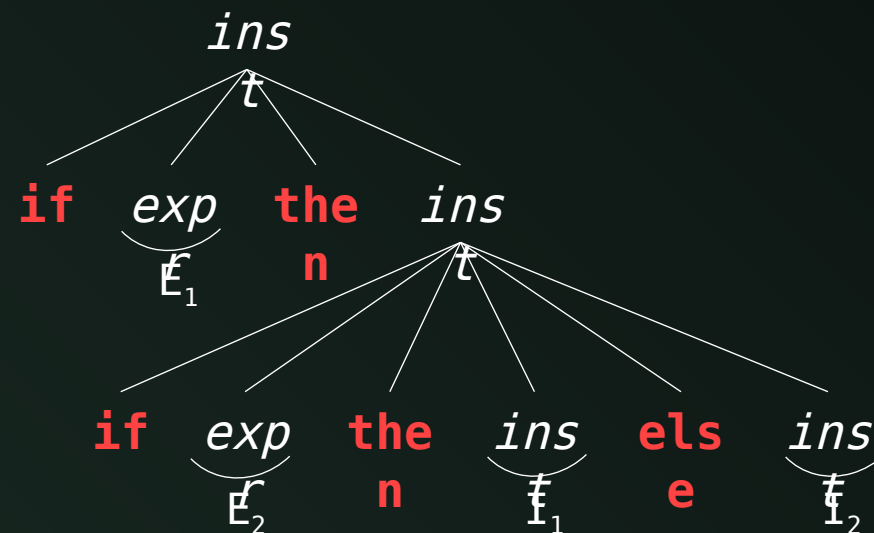
Eliminando Ambiguidade

- Para eliminar a ambiguidade precisamos **reescrever a gramática**
 - Evitando que **uma das árvores** seja derivável
 - Em um **if-then-else**, o **else** deve casar sempre com o **then** mais próximo

Derivações possíveis:




✓ **if** E_1 **then** **if** E_2 **then** I_1 **else**
 I_2 $\underbrace{\hspace{10em}}_{inst}$

✗ **if** E_1 **then** **if** E_2 **then** I_1 **else**
 I_2 $\underbrace{\hspace{10em}}_{inst}$



Eliminando Ambiguidade

- Podemos **forçar o casamento correto** com a gramática abaixo
 - Separando condicionais que terminam com if-then e if-then-else

```
inst       matched
          | open
matched  if expr then matched else matched
          | other
open     if expr then inst
          | if expr then matched else open
```

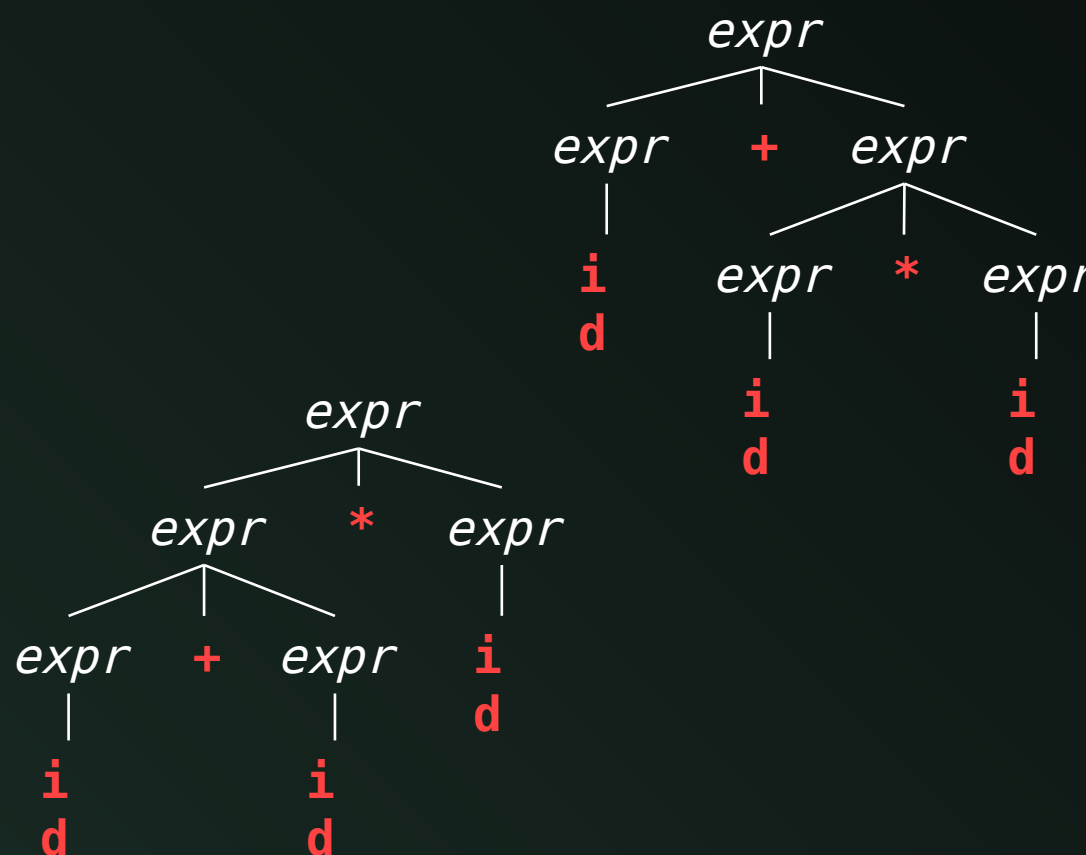
Não é mais possível: **if** E_1 **then** $\underbrace{\text{if } E_2 \text{ then } I_1}_{inst} \text{ else } \mathbf{\times}$
 I_2

Eliminando Ambiguidade

- Um exemplo com expressões

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

A gramática é ambígua porque a cadeia $id + id * id$ possui duas árvores de derivação



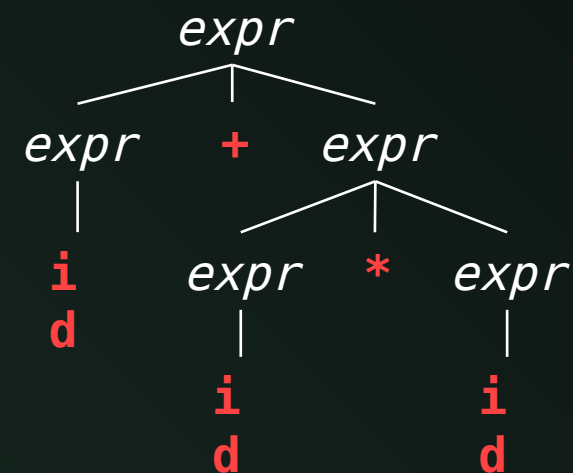
Eliminando Ambiguidade

- Para eliminar a ambiguidade precisamos **reescrever a gramática**
 - Evitando que **uma das árvores** seja derivável
 - Em uma expressão aritmética, multiplicação tem precedência sobre soma

Derivações iniciais possíveis:




✓ $\underbrace{id + id}_{expr} * \underbrace{id}_{expr}$

✗ $\underbrace{\text{id} + \text{id}}_{\text{expr}} * \underbrace{\text{id}}_{\text{expr}}$



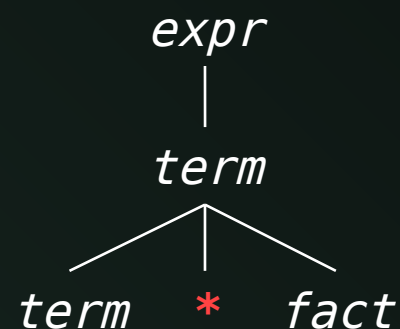
Eliminando Ambiguidade

- Podemos **forçar uma ordem** para a derivação
 - Impede que a multiplicação seja derivada antes da soma

exp  *expr* + *term*
r | *term*
ter  *term* * *fact*
m | *fact*
fac  (*expr*)
t | *id*

Não é mais
possível iniciar pela
multiplicação:

✗ *id* + *id* *
id
expr



✗

Não dá mais
para derivar

expr +
term

Eliminando Ambiguidade

- Necessitamos sempre de informações sobre o comportamento esperado das construções **para remover ambiguidades**
 - Um else deve casar com o if mais próximo
 - Multiplicação tem precedência sobre soma
- Não existe um **procedimento geral** para eliminar ambiguidade
 - A análise deve ser feita caso a caso
- Também **não existe algoritmo** para detectar gramáticas ambíguas
 - É preciso achar uma cadeia que gere ambiguidade

Eliminando Recursão

- Os métodos de **análise descendente** não funcionam com gramáticas recursivas à esquerda
 - Uma gramática possui **recursão à Esquerda** se ela tiver um não-terminal A tal que, para alguma cadeia α :

$$A \xRightarrow{+} A\alpha$$

- O caso mais básico é a **recursão à esquerda imediata**
- Existe uma produção da forma:

$$A \rightarrow A\alpha$$

Eliminando Recursão

- A recursão imediata pode ser **eliminada com substituições**

$A \rightarrow A\alpha$
 $\quad \mid \beta$

por

$A \rightarrow \beta R$
 $R \rightarrow \alpha R$
 $\quad \mid \epsilon$

Gramática com
recursão à esquerda

$expr \rightarrow expr + term$
 $\quad \mid term$
 $term \rightarrow term * fact$
 $\quad \mid fact$
 $fact \rightarrow (expr)$
 $\quad \mid id$

Gramática sem
recursão à
esquerda

$expr \rightarrow term \textit{plus}$
 $\textit{plus} \rightarrow + term \textit{plus}$
 $\quad \mid \epsilon$
 $term \rightarrow fact \textit{mult}$
 $\textit{mult} \rightarrow * fact \textit{mult}$
 $\quad \mid \epsilon$
 $fact \rightarrow (expr)$
 $\quad \mid id$

Eliminando Recursão

- A técnica pode ser usada para **qualquer quantidade de produções**

- **Agrupe as produções** de forma que nenhum β_i comece com A

$$A \Rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$


- Depois **substitua as produções** A por:

$$\begin{aligned} A &\Rightarrow \beta_1 R \mid \beta_2 R \mid \dots \mid \beta_n R \\ R &\Rightarrow \alpha_1 R \mid \alpha_2 R \mid \dots \mid \alpha_m R \mid \epsilon \end{aligned}$$

- O procedimento **elimina a recursão imediata** à esquerda das produções desde que nenhum α_i seja ϵ

Eliminando Recursão

- Exemplo:

A		A	c
		A	a d
		b d	
		ϵ	

Agrupando as produções:

$A \langle \text{icon} \rangle A\alpha_1 \mid A\alpha_2 \mid \beta_1 \mid \beta_2$

$\alpha_1 = \mathbf{c}$

$\alpha_2 = \mathbf{a d}$

$\beta_1 = \mathbf{b d}$

$\beta_2 = \epsilon$

Substituindo as produções A por:

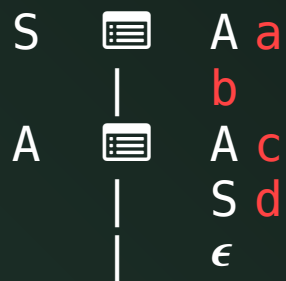
$A \langle \text{icon} \rangle \beta_1 R \mid \beta_2 R$

$R \langle \text{icon} \rangle \alpha_1 R \mid \alpha_2 R \mid \epsilon$

A		b d	R
		R	
R		c	R
		a d	R
		ϵ	

Eliminando Recursão

- A técnica anterior funciona para a recursão imediata mas não para derivações em dois ou mais passos



S é recursivo à esquerda porque
 $S \Rightarrow A \mathbf{a} \Rightarrow S \mathbf{d} \mathbf{a}$

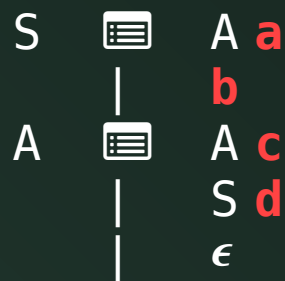
- O algoritmo a seguir elimina sistematicamente a recursão
 - Contanto que não existam ciclos ($A^+ \Rightarrow A$)

Eliminando Recursão

Algoritmo: eliminar recursão à esquerda de uma gramática

```

arrume os não-terminais em uma ordem  $A_1, A_2, \dots, A_n$ 
para (cada  $i$  de 1 até  $n$ )
    para (cada  $j$  de 1 até  $i-1$ )
        substitua cada produção da forma  $A_i \rightarrow A_j \gamma$ , onde
         $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  são produções- $A_j$ , pelas
        produções  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ 
    elimine as recursões imediatas nas produções- $A_i$ 
  
```



Ordenando os não-terminais: $S(A_1), A(A_2)$

- Para $i = 1$, não há recursões imediatas para $S(A_1)$
- Para $i = 2$, substitua $A \rightarrow S \text{ **d**}$ por $A \rightarrow A \text{ **a d**} \mid \text{**b d**}$
- Elimine as recursões imediatas nas produções A

Eliminando Recursão

- Aplicando algoritmo sobre a gramática recursiva à esquerda:



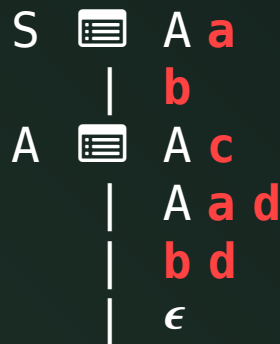
Ordenando os não-terminais como: $S(A_1)$, $A(A_2)$

- Para $i = 1$, não há recursões imediatas para $S(A_1)$

- Para $i = 2$, substitua

$A \rightarrow Sd$ por $A \rightarrow Aa d \mid b d$

- Elimine as recursões imediatas nas produções A



Fatoração à Esquerda

- Na derivação de uma cadeia, um **reconhecedor sintático preditivo** escolhe uma produção apenas olhando o próximo símbolo terminal

- Não podem existir duas produções iniciando com o mesmo símbolo t

```
| if expr then inst else  
| inst  
| other
```

- A fatoração à esquerda é uma **transformação** que torna uma gramática **adequada para o reconhecimento preditivo**

Fatoração à Esquerda

- Quando a escolha entre duas produções não é clara, podemos reescrever as produções para adiar a decisão

$$\begin{array}{l} A \Rightarrow \alpha \beta_1 \\ \quad | \quad \alpha \beta_2 \end{array}$$
$$\begin{array}{l} A \Rightarrow \alpha R \\ R \Rightarrow \beta_1 \mid \beta_2 \end{array}$$
$$\begin{array}{l} inst \\ t \end{array} \Rightarrow \begin{array}{l} \text{if } expr \text{ then } inst \\ | \text{ if } expr \text{ then } inst \text{ else } \\ \quad inst \\ | \text{ other} \end{array}$$
$$\begin{array}{l} inst \\ t \end{array} \Rightarrow \begin{array}{l} \text{if } expr \text{ then } inst \ opt \\ | \text{ other} \end{array}$$
$$\begin{array}{l} opt \\ \end{array} \Rightarrow \begin{array}{l} \text{else } inst \\ | \epsilon \end{array}$$

Gramáticas

- As gramáticas descrevem a maior parte, mas não toda a sintaxe de uma linguagem de programação
 - Não podem ser descritos por uma gramática livre de contexto:
 - A exigência dos identificadores serem declarados antes do seu uso
 - Verificação do número de parâmetros em uma chamada de função
- As sequências de *tokens* aceitos pelo analisador sintático representam um superconjunto da linguagem de programação
 - A fase de análise semântica deve analisar o resultado para garantir o cumprimento de todas as regras da linguagem

Exercícios

1. Fatore a gramática a seguir:

```
stm   while (expr) stmt1  
t
```

```
| do stmt1 while (expr);
```


```
| for (expr; expr; expr) stmt1
```

```
| for (expr; decl : container)  
    stmt1
```

```
stmt   while (expr) stmt1
```

```
| do stmt1 while (expr);
```

```
| for (expr; for_tail
```

```
for_tail   expr; expr) stmt1  
l
```

```
    decl : container)  
    stmt1
```


Exercícios

2. A gramática a seguir define **expressões regulares** sobre **a** e **b**

a) Fatore a gramática

Já está fatorada.

b) A fatoração torna ela adequada para análise sintática descendente?
Não, ela é recursiva à esquerda.

```
rexpr  ::= rexpr | rterm
rterm  ::= rterm rfactor
rfactor ::= rfactor *
rprimary ::= a
         ::= b
```

Exercícios

2. A gramática a seguir define **expressões regulares** sobre **a** e **b**

c) Elimine a recursão à esquerda

```
rexpr  ::= rterm union
union  ::= | rterm union |
ε
rterm  ::= rfactor concat
concat ::= rfactor concat |
ε
rfactor ::= rprimary closure
closure ::= * closure | ε
rprimary ::= a | b
```

```
rexpr  ::= rexpr | rterm
        | rterm
rterm  ::= rterm rfactor
        | rfactor
rfactor ::= rfactor *
        | rprimary
rprimary ::= a
         | b
```

Exercícios

2. A gramática a seguir define **expressões regulares** sobre **a** e **b**

- d) A gramática resultante é adequada para análise sintática descendente?

Se não for ambígua,
sim.

```
rexpr  ::= rterm union
union  ::= | rterm union |
ε



rterm  ::= rfactor concat
concat ::= rfactor concat |
ε

rfactor ::= rprimary closure
closure ::= * closure | ε

rprimary ::= a | b
```

Exercícios

3. A gramática a seguir é proposta para remover a “ambiguidade do else vazio”:

```
inst       if expr then inst  
          | matched  
matched  if expr then matched else inst  
          | other
```

Mostre que a gramática ainda é ambígua.

Resumo

- Existem várias técnicas de transformação de gramáticas
 - Visam adequá-las para os métodos de reconhecimento sintático:
 - Análise descendente
 - Análise ascendente
- Para um reconhecedor sintático descendente preditivo, as seguintes técnicas são imprescindíveis:
 - Remoção de ambiguidades
 - Eliminação da recursão à esquerda
 - Fatoração à esquerda