

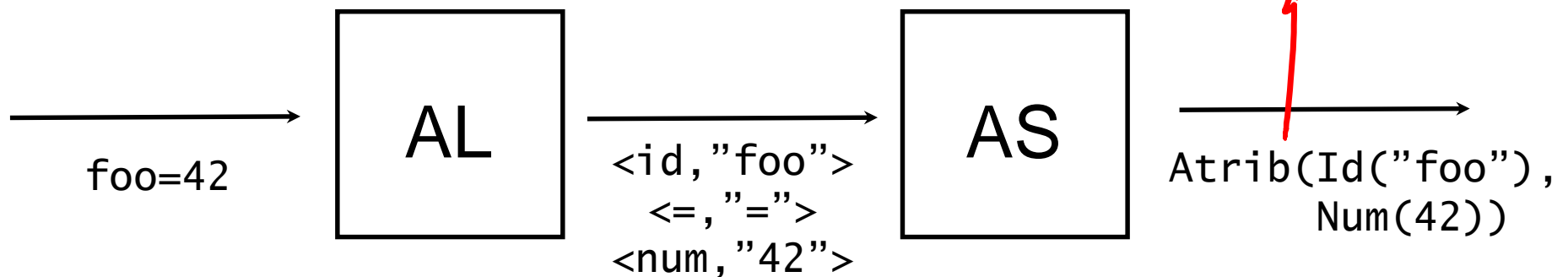
Compiladores - Gramáticas

Fabio Mascarenhas – 2015.1

<http://www.dcc.ufrj.br/~fabiom/comp>

Análise Sintática

- A análise sintática agrupa os tokens em uma *árvore sintática* de acordo com a estrutura do programa (e a gramática da linguagem)
- Entrada: sequência de tokens fornecida pelo analisador léxico
- Saída: árvore sintática do programa



Análise Sintática

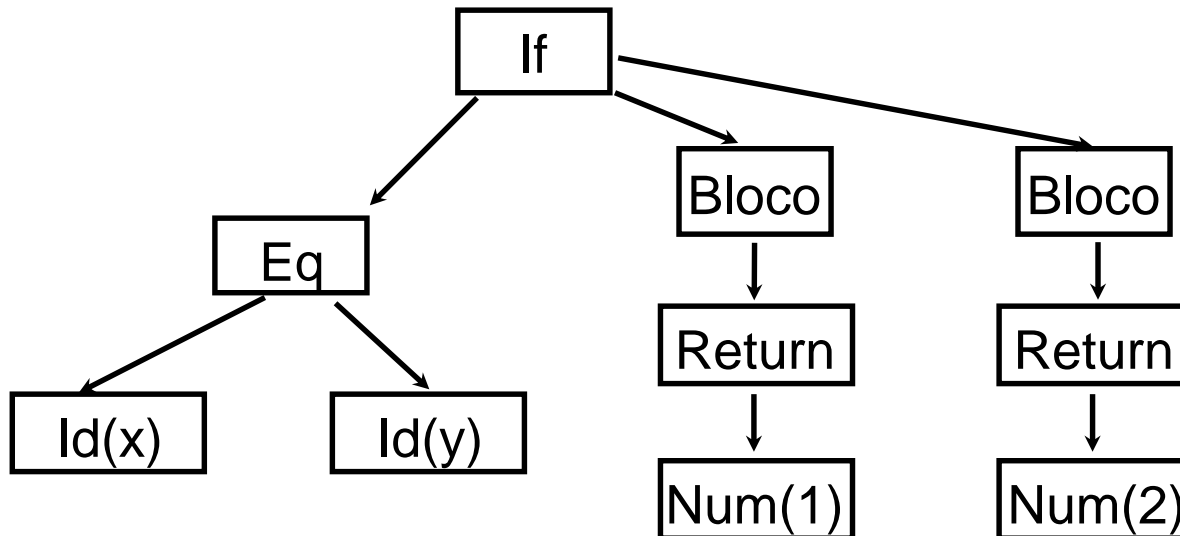
- Programa:

```
if | x | == | y | then | return | 1 | else | return | 2 | end |
```

- Tokens:

IF ID EQ ID THEN RETURN NUM ELSE RETURN NUM END

- Árvore:



Programas válidos e inválidos

- Nem todas as sequências de tokens são programas válidos
- O analisador sintático tem que distinguir entre sequências válidas e inválidas
- Precisamos de:
 - Uma linguagem para *descrever sequências válidas de tokens* e a estrutura do programa → especificação → GRAMÁTICAS
 - Um *método* para distinguir sequências válidas de inválidas e extrair essa estrutura das sequências válidas → implementação, PARSERS

Estrutura recursiva

- A estrutura de uma linguagem de programação é *recursiva*
- Uma *expressão* é:
 - $\langle \text{expressão} \rangle + \langle \text{expressão} \rangle$
 - $\langle \text{expressão} \rangle == \langle \text{expressão} \rangle$
 - $(\langle \text{expressão} \rangle)$
 - ...
- *Gramáticas livres de contexto* são uma notação natural para esse tipo de estrutura recursiva

CFGs

- Uma gramática livre de contexto (CFG) é formada por:

- Um conjunto de *terminais* (T) → tokens (tipos)

- Um conjunto de *não-terminais* (V) → estrutura → expressão comando

- Um *não-terminal inicial* (S)

- Um conjunto de *produções* (P) → regras sintáticas

Produções

- Uma produção é um par de um *não-terminal* e uma cadeia (possivelmente vazia) de terminais e não-terminais
- Podemos considerar produções como *regras*; o não-terminal é o lado esquerdo da regra, e a cadeia é o lado direito
- É comum escrever gramáticas usando apenas as produções; os conjuntos de terminais e não-terminais e o não-terminal inicial podem ser deduzidos com a ajuda de algumas convenções tipográficas

$A \rightarrow \text{det}$
 $A \rightarrow \text{indef}$
 $\emptyset \rightarrow A \text{ sub}$

$S \rightarrow SJ \text{ verb}$
 $S \rightarrow SJ \text{ verb } \emptyset$
 $SJ \rightarrow \text{sub}$
 $SJ \rightarrow A \text{ sub}$

CFGs são geradores

- Uma CFG é um *gerador* para cadeias de alguma linguagem
- Para gerar uma cadeia, começamos com o não-terminal inicial
- Substituímos então um não-terminal presente na cadeia pelo lado direito de uma de suas regras
- Fazemos essas substituições até ter uma string apenas de terminais

Deriva em um passo/n passos

- Se obtemos a cadeia w a partir da cadeia v com uma substituição de não-terminal dizemos que v *deriva em um passo*: $v \rightarrow w$
- O fecho reflexivo-transitivo da relação *deriva em um passo* é a relação *deriva em n passos*:

- $v \rightarrow^* v$

- Se $v \rightarrow w$ então $v \rightarrow^* w$

- Se $u \rightarrow^* v$ e $v \rightarrow^* w$ então $u \rightarrow^* w$

- A *linguagem* da gramática G são as cadeias de **terminais** w tal que $S \rightarrow^* w$

Handwritten red annotations showing derivations:

- $S S \text{ val } (0) \rightarrow S S \text{ val } (A \text{ sub})$
- $(S S) \text{ val } 0 \rightarrow (S S) \text{ val } 0$

Quiz

- Quais das cadeias abaixo estão na gramática dada?

a b c b a ~~X~~

a c c a ~~X~~

a b a ✓

a b c b c b a ~~X~~

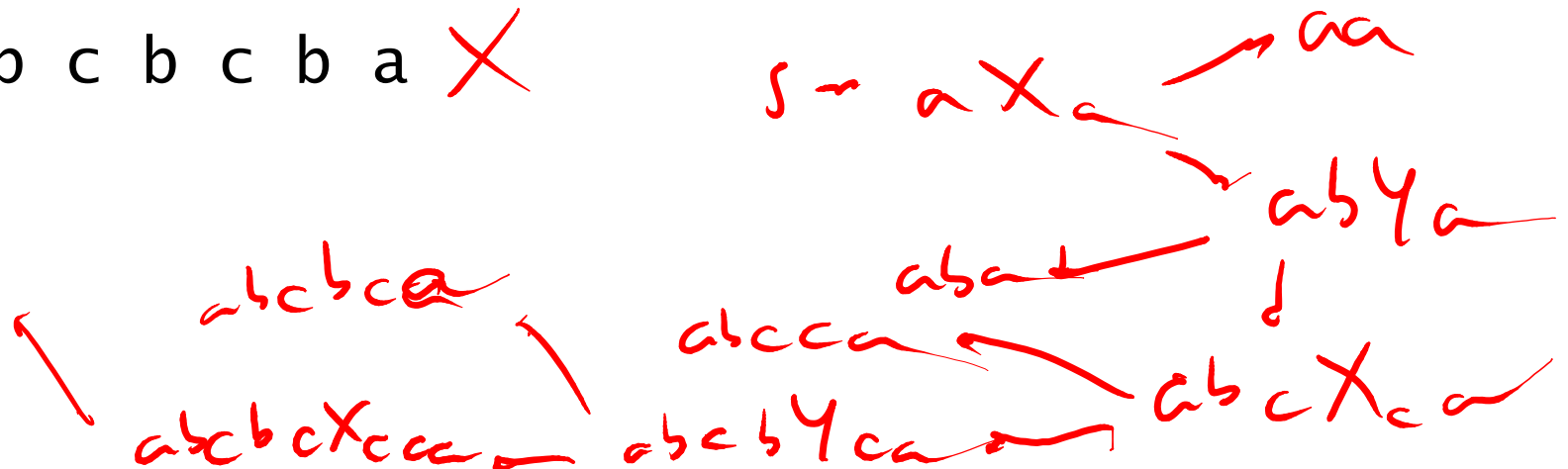
$S \rightarrow a X a$

$X \rightarrow$

$X \rightarrow b Y$

$Y \rightarrow$

$Y \rightarrow c X c$



Quiz

- Quais das cadeias abaixo estão na gramática dada?

a b c b a

a c c a

* a b a

a b c b c b a

$S \rightarrow a X a$

$X \rightarrow$

$X \rightarrow b Y$

$Y \rightarrow$

$Y \rightarrow c X c$

Exemplo - expressões aritméticas simples

- Uma gramática bastante simples mas que exemplifica várias questões de projeto de gramáticas

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{num} \end{aligned}$$

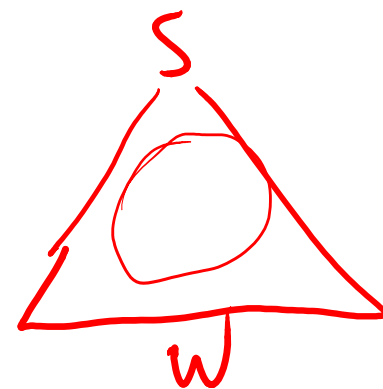
Forma é importante

- Na construção de compiladores estamos tão interessados nas gramáticas quanto as linguagens que elas geram
- Muitas gramáticas podem gerar a mesma linguagem, mas a gramática vai ditar a *estrutura* do programa resultante
- A estrutura é a saída mais importante da fase de análise sintática

Derivações

- Uma *derivação* de uma cadeia w é uma sequência de substituições que leva de S a w :

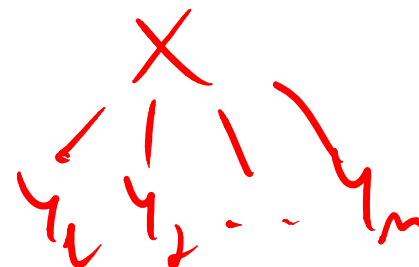
- $S \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow w$



- Uma derivação pode ser desenhada como uma árvore

- A raiz é S

- Para se uma substituição $X \rightarrow Y_1 \dots Y_n$ é usada acrescenta-se os filhos $Y_1 \dots Y_n$ ao nó X



Árvore sintática (árvore de parse)

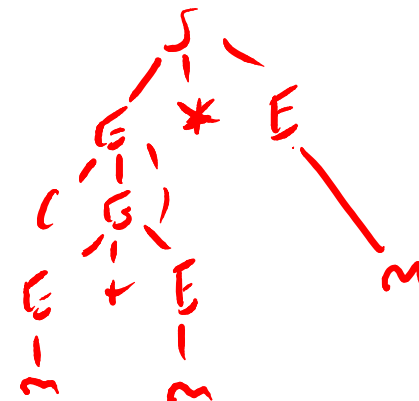
- Uma árvore sintática [concreta] tem

- terminais nas folhas
- não-terminais nos nós interiores

$$\begin{aligned} S &\rightarrow E * E \rightarrow (E) * E \rightarrow \\ &(E + E) * E \rightarrow \\ &(m + n) * E \rightarrow (m + n) * m \end{aligned}$$

- Percorrer as folhas da árvore *em ordem* dá a cadeia sendo derivada
- A árvore sintática dá a *estrutura e associatividade* das operações que a cadeia original não mostra

$$(m + n) * m$$



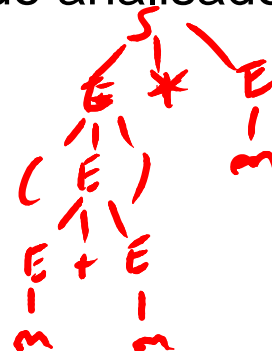
Mais à esquerda e mais à direita

- Qualquer sequência de substituições que nos leve de S a w é uma derivação de w , mas em geral estamos interessados em *derivações sistemáticas*
- Uma *derivação mais à esquerda* de w é uma sequência de substituições em que sempre substituímos o não-terminal *mais à esquerda*
- Uma *derivação mais à direita* de w é uma sequência de substituições em que sempre substituímos o não-terminal *mais à direita*
- Veremos que estratégias de análise sintática diferentes levam a derivações mais à esquerda ou mais à direita

Unicidade da árvore sintática

$$\begin{aligned} S &\rightarrow E * E \rightarrow E * m \\ &\rightarrow (E) * m \rightarrow (E + E) * m \\ &\rightarrow (E + m) * m \rightarrow (m + m) * m \end{aligned}$$

- Podemos ter várias *derivações* para uma mesma cadeia w , mas só pode haver **uma árvore sintática**
- A árvore sintática dá a estrutura do programa, e a estrutura se traduz no significado do programa
- Logo, um programa com mais de uma árvore sintática tem mais de uma possível interpretação!
- Já a diferença entre uma derivação mais à esquerda e mais à direita se traduz em uma diferença na implementação do analisador sintático, e não na estrutura do programa

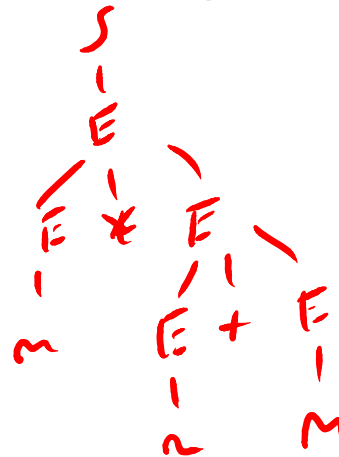
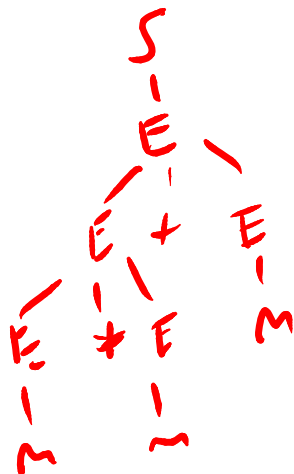


Uma cadeia, duas árvores

- Vamos voltar para a gramática de expressões:

$$\begin{aligned} S &\rightarrow E \\ E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ E &\rightarrow \text{num} \end{aligned}$$

- Podemos obter duas árvores diferentes para a cadeia $\text{num} * (\text{num} + \text{num})$

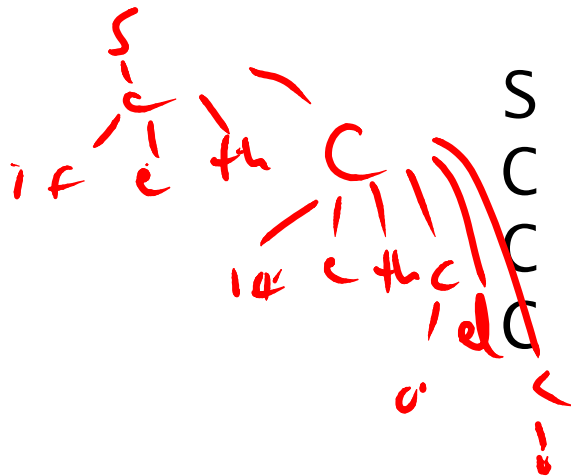


Ambiguidade

- Uma gramática é *ambígua* se existe alguma cadeia para qual ela tem mais de uma *árvore sintática*
 - De maneira equivalente, se existe mais de uma derivação *mais à esquerda* para uma cadeia
 - Ou se existe mais de uma derivação *mais à direita* para uma cadeia
 - As três definições são equivalentes
- Ambiguidade é ruim para uma linguagem de programação, pois leva a interpretações inconsistentes entre diferentes compiladores

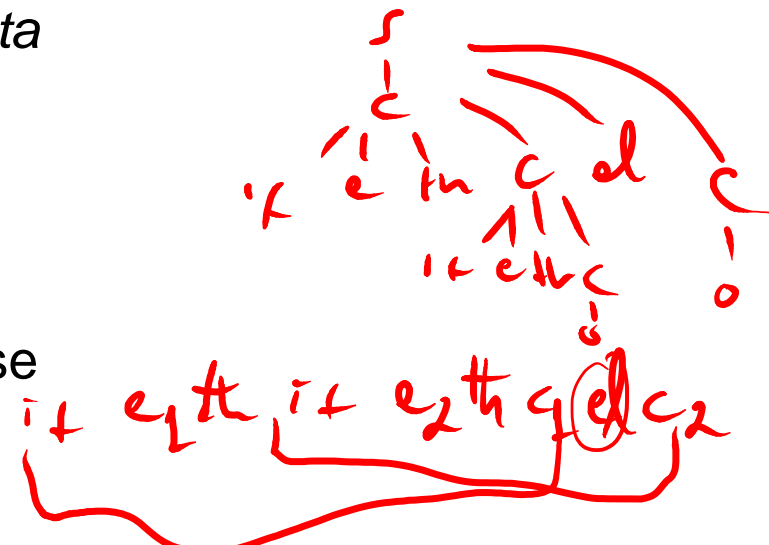
Detectando ambiguidade

- Infelizmente, não existe um algoritmo para detectar se uma gramática qualquer é ambígua ou não
- Mas existem *heurísticas*, a principal delas é verificar se existe uma regra misturando *recursão à esquerda e recursão à direita*
 - É o caso da gramática de expressões
 - Às vezes isso é bem sutil: ambiguidade do if-else



AMBIGUA!

$S \rightarrow C$
 $C \rightarrow \text{if exp then } C$
 $C \rightarrow \text{if exp then } C \text{ else } C$
 $C \rightarrow \text{outros}$



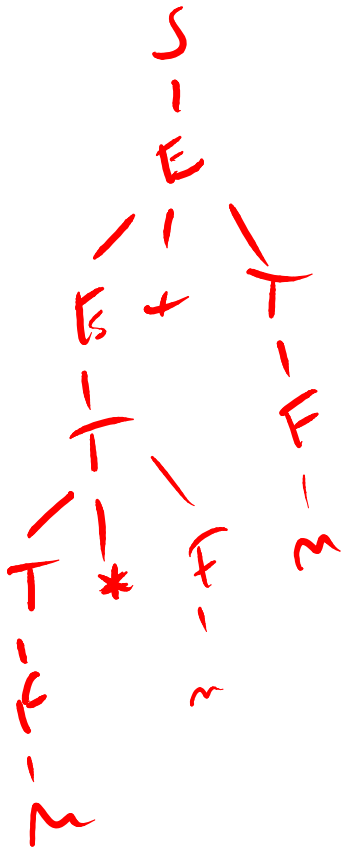
an + m * m

Removendo ambiguidade

- Do mesmo modo, não há um algoritmo para remover ambiguidade
- Se a ambiguidade está na gramática, e não na própria linguagem, o jeito é encontrar a *fonte* da ambiguidade e reescrever a gramática para eliminá-la
- No caso de ambiguidade em gramáticas de expressões e operadores, a ambiguidade vem da gramática não estar levando em conta as regras de associatividade e precedência dos operadores
- Em uma gramática de expressões, cada nível de precedência tem que ganhar seu próprio não-terminal
- Operadores que devem ser associativos à esquerda precisam usar recursão à esquerda, e associativos à direita precisam de recursão à direita

Expressões simples, sem ambiguidade

- Assumindo que $*$ tem precedência sobre $+$, e ambos são associativos à esquerda (ou seja, $\text{num} + \text{num} + \text{num}$ deve ser interpretado como $(\text{num} + \text{num}) + \text{num}$)



S	\rightarrow	E
E	\rightarrow	$E + T$
E	\rightarrow	T
T	\rightarrow	$T * F$
T	\rightarrow	F
F	\rightarrow	(E)
F	\rightarrow	num

$n * n + n$
 $n + n * n$



Expressões simples, sem ambiguidade

- Assumindo que \wedge tem precedência sobre $*$ que tem precedência sobre $+$, \wedge é associativo à direita, $*$ e $+$ são associativos à esquerda (ou seja, $\text{num} + \text{num} + \text{num}$ deve ser interpretado como $(\text{num} + \text{num}) + \text{num}$)

$$\begin{array}{l} S \rightarrow E \\ E \rightarrow E + T \\ E \rightarrow E - T \\ E \rightarrow T \\ T \rightarrow T * F \\ T \rightarrow F \\ F \rightarrow A \wedge F \\ F \rightarrow A \\ A \rightarrow (E) \\ A \rightarrow \text{num} \end{array}$$

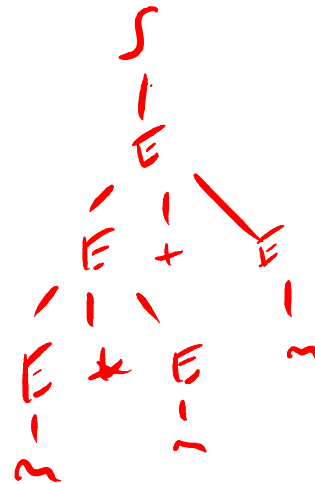
T → T / ≠

Duas derivações, uma árvore

- Duas derivações de uma frase podem dar a mesma árvore de uma for mais à esquerda e outra mais à direita

S \rightarrow E
-1- \rightarrow E + E
-2- \rightarrow E * E + E
-4- \rightarrow num * E + E
-4- \rightarrow num * num + E
-4- \rightarrow num * num + num

S \rightarrow E
-1- \rightarrow E + E
-4- \rightarrow E + num
-2- \rightarrow E * E + num
-4- \rightarrow E * num + num
-4- \rightarrow num * num + num



Duas derivações, duas árvores

- Duas derivações mais à esquerda dão duas árvores diferentes: gramática ambígua

S \rightarrow E
-1- \rightarrow E + E
-2- \rightarrow E * E + E
-4- \rightarrow num * E + E
-4- \rightarrow num * num + E
-4- \rightarrow num * num + num

S \rightarrow E
-2- \rightarrow E * E
-4- \rightarrow num * E
-1- \rightarrow num * E + E
-4- \rightarrow num * num + E
-4- \rightarrow num * num + num



If-else sem ambiguidade

if () { if () ; ~~else~~ ; }

- Uma solução adotada por diversas linguagens é acrescentar um delimitador que fecha o if:

S → C
C → if exp then C **end**
C → if exp then C else C **end**
C → outros

if exp then C end

- Uma desvantagem é que agora é necessário ter uma construção “else-if” para ter ifs em cascata sem uma multiplicação de ends

if exp then if exp then C else C end

- E claro, estamos mudando a linguagem!

if exp then if exp then C else C end

if exp then if exp then C end else C end

If-else sem ambiguidade, com a “cara de C”

- Uma solução adotada por diversas linguagens é acrescentar um delimitador que fecha o if:

do if

$$\begin{aligned} S &\rightarrow C \\ C &\rightarrow \text{if}(\text{exp})\{ C \} \\ C &\rightarrow \text{if}(\text{exp})\{ C \} \text{ else } \{ C \} \\ C &\rightarrow \text{outros} \end{aligned}$$

*if (e) {
} else {
if (e) {
} else { if (e) }
} ||*

- Uma desvantagem é que agora é necessário ter uma construção “else-if” para ter ifs em cascata sem uma multiplicação de ends

if (e) { if (e) { } else { } }

- E claro, estamos mudando a linguagem!

if (e) { if (e) { } } else { }

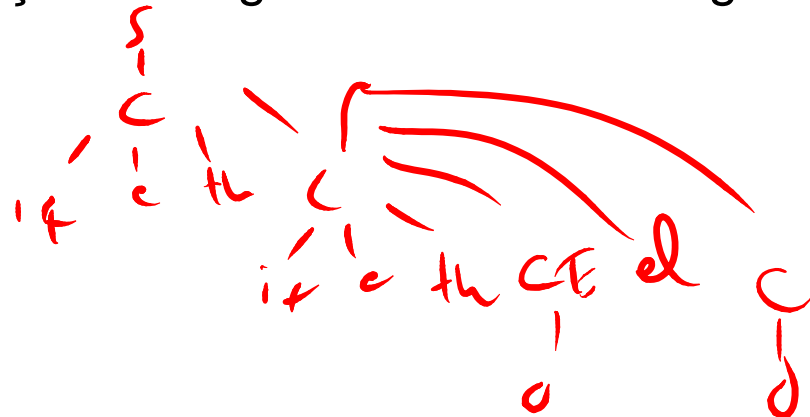
If-else sem mudar a linguagem *if e then if e then else o*

- Outra solução é separar os ifs em dois tipos, com não-terminais diferentes:

S → C
C → if exp then C
C → if exp then CE else C
C → outros
[CE → if exp then CE else CE
CE → outros *if else*

if then if then else

- Notem a semelhança com a gramática não ambígua de expressões



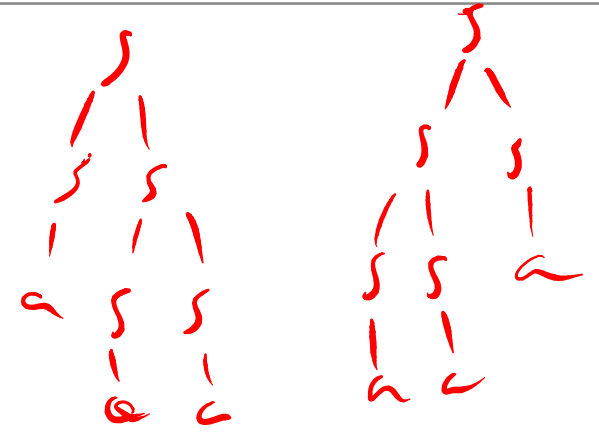
Quiz

- Qual a versão não ambígua da gramática:

Handwritten notes in red ink:

- $S \rightarrow S a$
- $S \rightarrow S b$
- $S \rightarrow a$
- $S \rightarrow b$

$S \rightarrow S S$
 $S \rightarrow a$
 $S \rightarrow b$



$S \rightarrow S a$
 $S \rightarrow S b$
 $S \rightarrow$

Handwritten red text: "inclui"

$S \rightarrow S S'$
 $S' \rightarrow a$
 $S' \rightarrow b$

~~$S \rightarrow S$
 $S \rightarrow S'$
 $S' \rightarrow a$
 $S' \rightarrow b$~~

Handwritten red text: "S -> S"

$S \rightarrow S a$
 $S \rightarrow S b$
 $S \rightarrow a$
 $S \rightarrow b$



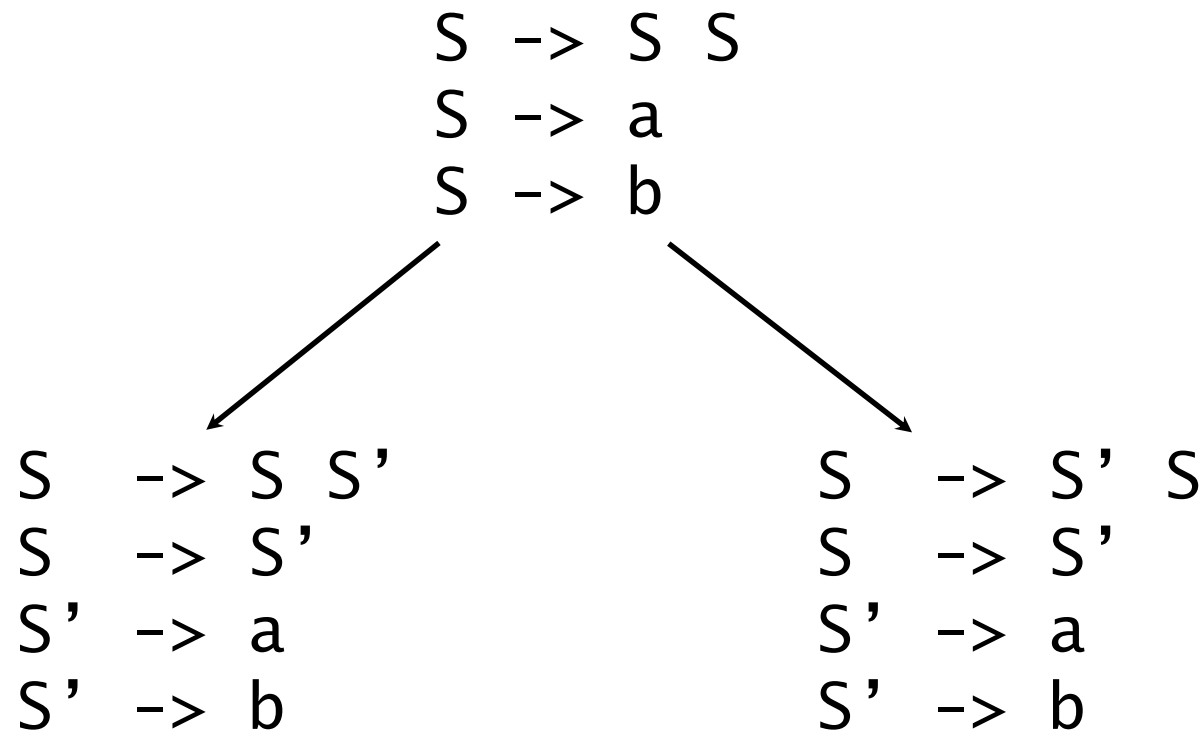
Quiz

- Qual a versão não ambígua da gramática:

$$\begin{aligned} S &\rightarrow S S \\ S &\rightarrow a \\ S &\rightarrow b \end{aligned}$$
$$\begin{aligned} S &\rightarrow S a \\ S &\rightarrow S b \\ S &\rightarrow \end{aligned}$$
$$\begin{aligned} S &\rightarrow S S' \\ S' &\rightarrow a \\ S' &\rightarrow b \end{aligned}$$
$$\begin{aligned} S &\rightarrow S \\ S &\rightarrow S' \\ S' &\rightarrow a \\ S' &\rightarrow b \end{aligned}$$
$$\begin{aligned} * S &\rightarrow S a \\ S &\rightarrow S b \\ S &\rightarrow a \\ S &\rightarrow b \end{aligned}$$

Quiz

- Qual a versão não ambígua da gramática:



Contornando ambiguidade

- Na prática, um uso judicioso de ambiguidade pode simplificar a gramática, e deixar ela mais natural
- Tanto a versão ambígua da gramática de expressões simples quanto a gramática do if-else são mais simples que suas versões não ambíguas!
- Podemos eliminar a ambiguidade não na gramática, mas na *implementação do analisador sintático*
- As ferramentas de geração de analisadores possuem regras de eliminação de ambiguidade, e diretivas de precedência e associatividade que permitem controlar como essa eliminação é feita