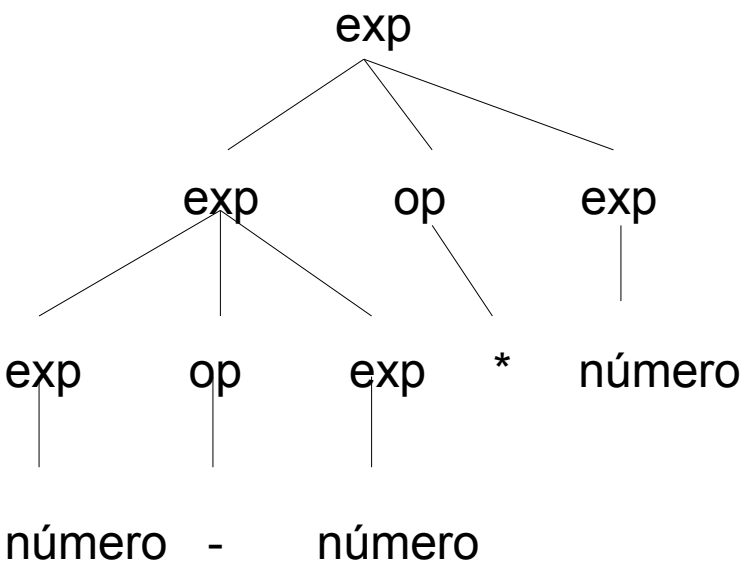
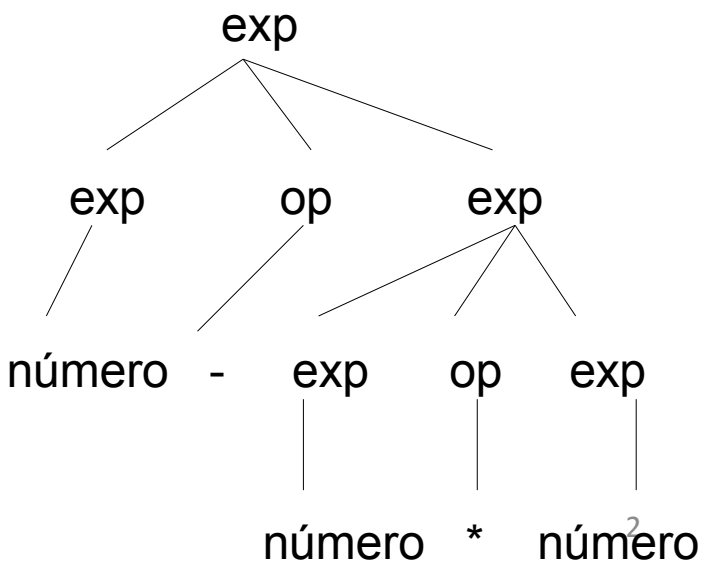
# Compiladores
# Análise Sintática
# Ambiguidade

Profa. Dra. Ana Carolina Lorena
UNIFESP

# Ambiguidade

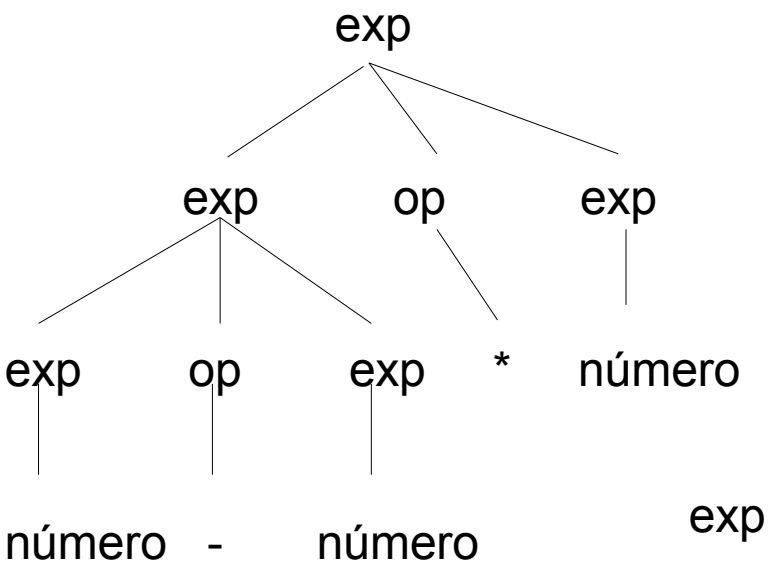
- Uma gramática pode permitir que uma cadeia tenha mais de uma árvore de análise sintática
  - Ex.: `exp → exp op exp | (exp) | número`
    `op → + | – | *`
    E a cadeia 34-3*42



# Ambiguidade

- Ex.: árvores correspondem a duas derivações à esquerda

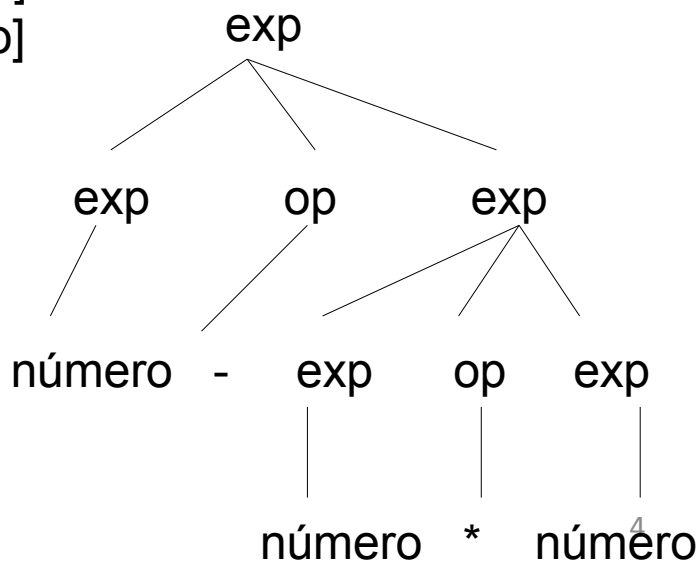



exp ⇒ exp op exp [exp → exp op exp]
⇒ exp op exp op exp [exp → exp op exp]
⇒ número op exp op exp [exp → número]
⇒ número – exp op exp [op → -]
⇒ número – número op exp [exp → número]
⇒ número – número * exp [op → *]
⇒ número – número * número [exp → número]

# Ambiguidade

- Ex.: árvores correspondem a duas derivações à esquerda

| | |
|---|---|
| exp ⇒ exp op exp | [exp → exp op exp] |
| ⇒ número op exp | [exp → número] |
| ⇒ número - exp | [op → -] |
| ⇒ número – exp op exp | [exp → exp op exp] |
| ⇒ número – número op exp | [exp → número] |
| ⇒ número – número * exp | [op → *] |
| ⇒ número – número * número | [exp → número] |



# Ambiguidade

- Uma gramática que gera uma cadeia com duas árvores de análise sintática distintas é denominada ambígua
  - Não especifica com precisão a estrutura sintática de um programa
  - É como um autômato não determinístico
  - Mas ambiguidade não é facilmente removível (não há algoritmo para isso)

# Ambiguidade

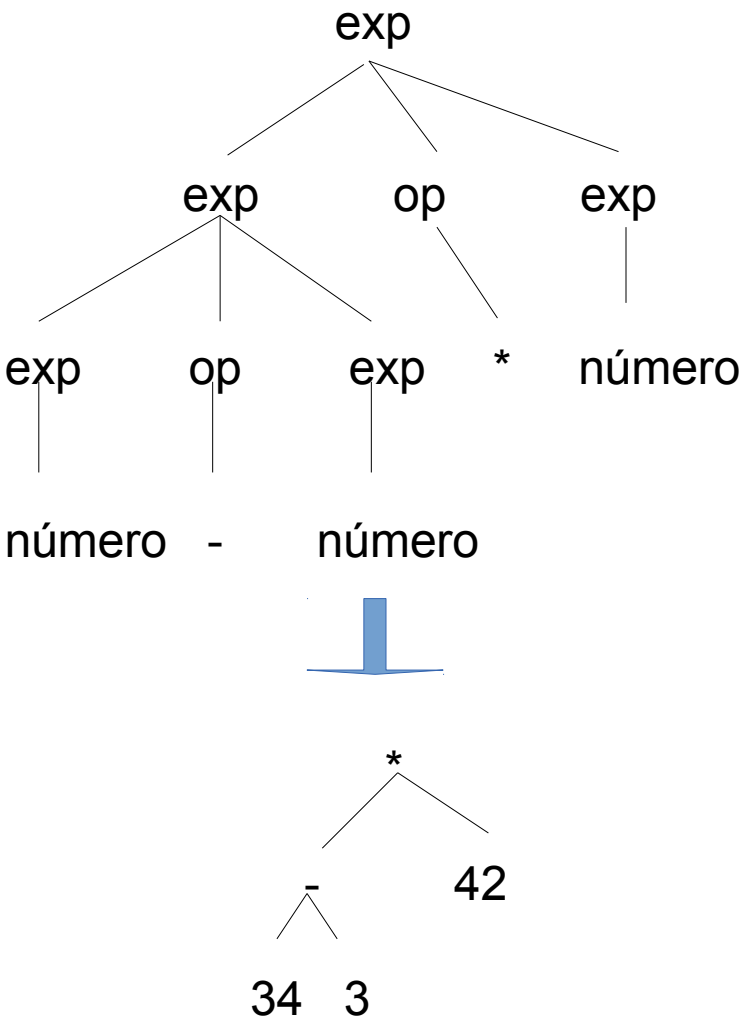
- Gramáticas ambíguas devem ser evitadas para projeto de analisador sintático
  - Nem sempre isso é possível
  - Mas há métodos básicos para tratar ambiguidades no projeto de compiladores

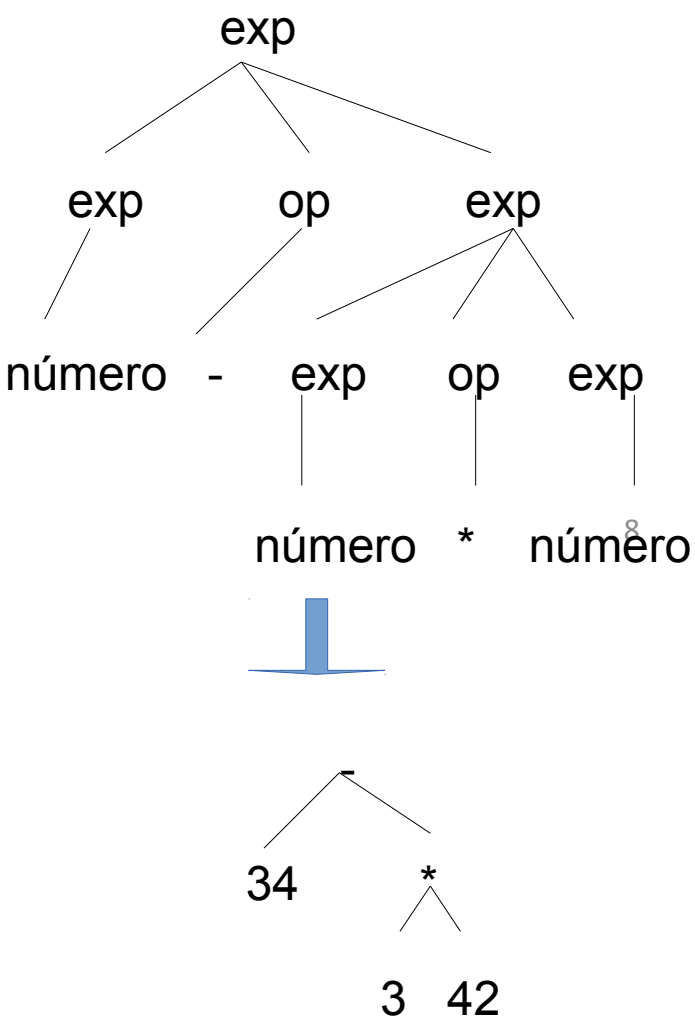
# Alternativas para lidar com ambiguidade

- Estabelecer regra de eliminação de ambiguidade: especifica, em caso de ambiguidade, que árvore de análise sintática é a correta
  - Não precisa então alterar a gramática

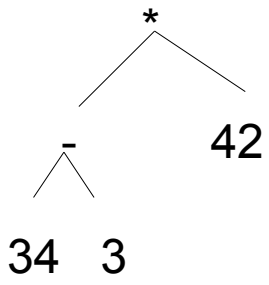
- Alterar a gramática: forçar a construção da árvore de análise sintática correta
  - Removendo a ambiguidade

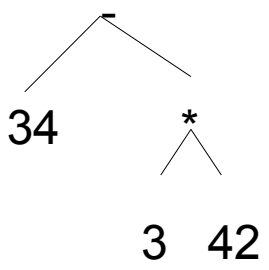
# Exemplo


exp
exp op exp
exp op exp * número
número - número
*
- 42
34 3


exp
exp op exp
número - exp op exp
número * número
-
34 *
3 42

# Exemplo



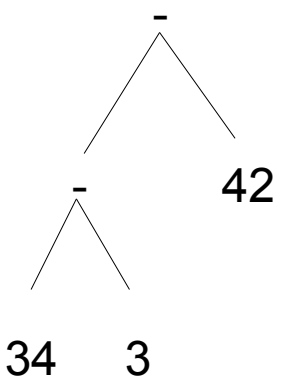

Avaliar primeiro a subtração, e depois a multiplicação

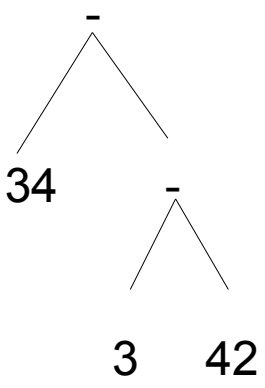
Avaliar primeiro a multiplicação, e depois a subtração

Qual é a interpretação correta?

# Eliminando a ambiguidade

- $\Rightarrow$ estabelecer regra de eliminação de ambiguidade com precedências das operações
  - Ainda não elimina completamente ambiguidade
  - Ex.: 34-3-42




Qual é a interpretação correta?

# Eliminando a ambiguidade

- Tem então que estabelecer regras também de associatividade das operações (cada uma)

- Outra alternativa: exigir parênteses

```
exp → fator op fator | fator
fator → (exp) | número
op → + | - | *
```

  - Mas alterou a gramática e também a linguagem reconhecida

# Precedência

- Reescrevendo a gramática sem alterar a linguagem reconhecida
  - Agrupar operadores em classes de igual precedência
  - Para cada precedência, escrever uma regra distinta
  - Ex.:

```
exp → exp soma exp | termo
soma → + | -
termo → termo mult termo | fator
mult → *
fator → (exp) | número
```

Agrupou multiplicação sob regra termo
E adição e subtração sob regra exp

# Precedência

- Analisando a gramática:

```
exp → exp soma exp | termo
soma → + | -
termo → termo mult termo | fator
mult → *
fator → (exp) | número
```

Como base para exp é termo, a adição e a subtração aparecerão acima nas árvores (com menor precedência)

Esse padrão é chamado cascata de precedências

# Associatividade

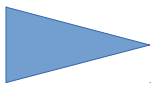
- Gramática ainda não resolveu problema de ambiguidade
  - Não especifica associatividade de operadores
  - Recursão dos dois lados de um operador possibilita que qualquer um dos lados case com repetições do operador
  - Solução: substituir uma das recursões pelo caso base, forçando casamentos repetidos para o lado com recursão
  - Ex.: exp → exp soma termo | termo
    - Torna adição e subtração associativas à esquerda
  - Ex.: exp → termo soma exp | termo
    - Torna adição e subtração associativas à direita

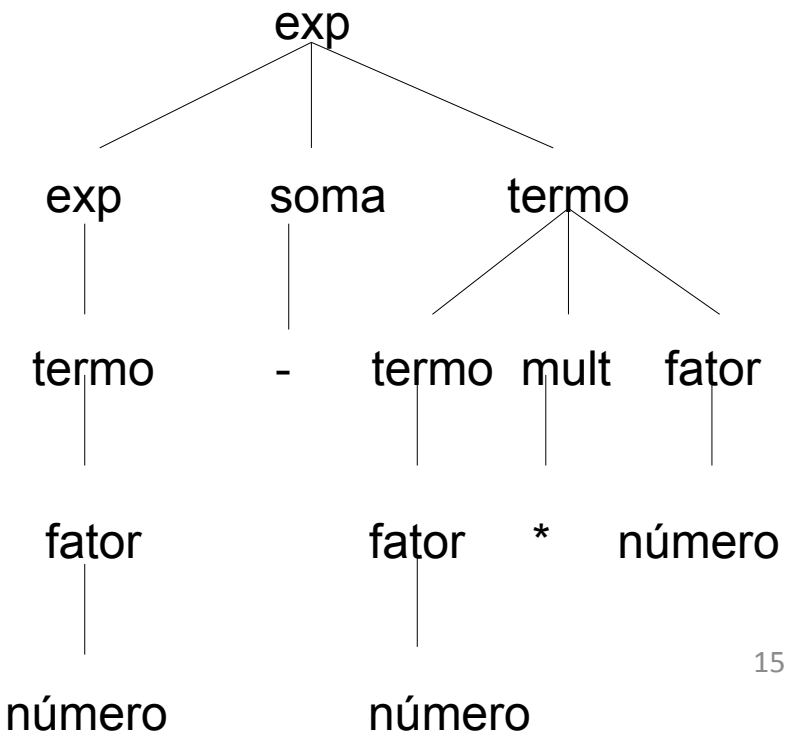
# Precedência e associatividade

- Reescrevendo a gramática:

```
exp → exp soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → *
fator → (exp) | número
```

Ex. 34 – 3 * 42



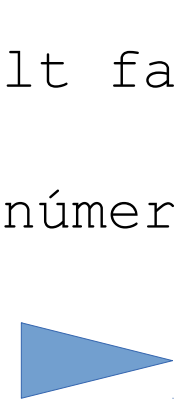


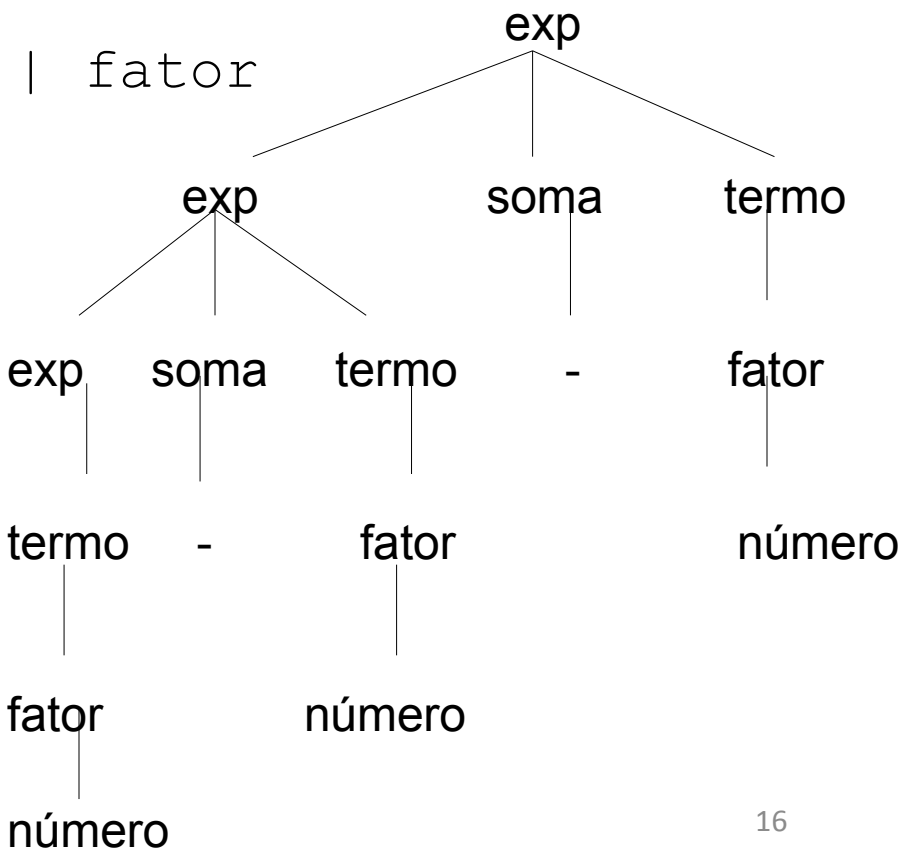
# Precedência e associatividade

- Reescrevendo a gramática:

```
exp → exp soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → *
fator → (exp) | número
```

Ex. 34 – 3 - 42





# Problema do else pendente

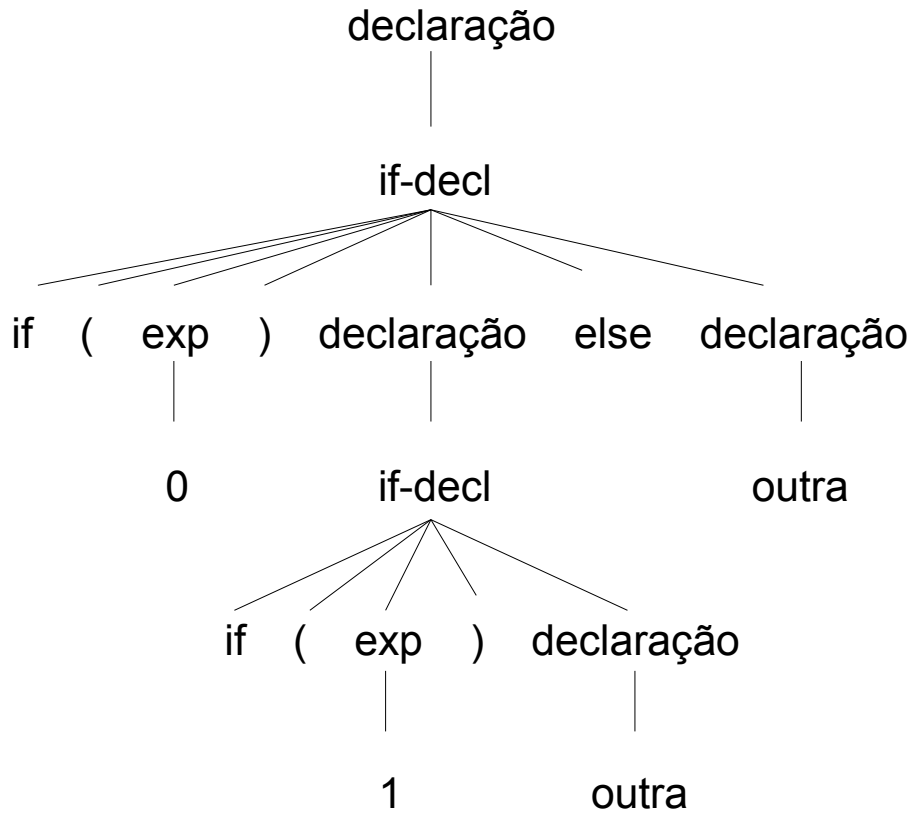
- Considere a gramática:

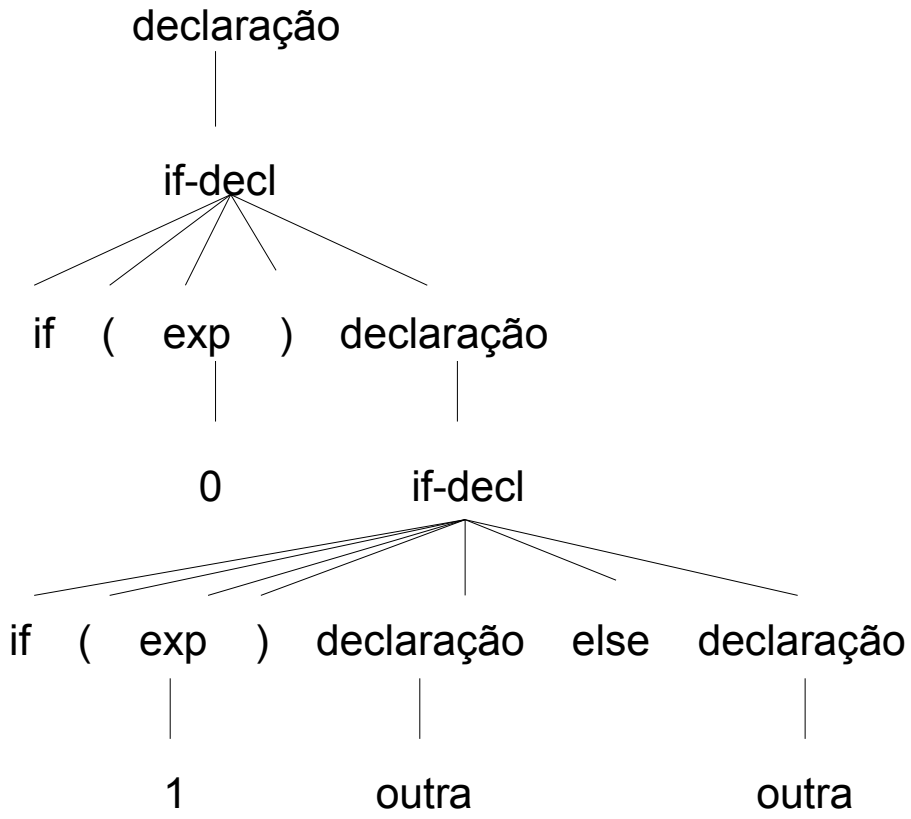
```
declaração → if-decl | outra
if-decl → if (exp) declaração |
          if (exp) declaração else declaração
exp → 0|1
```

  - Ela é ambígua em decorrência do else opcional

# Problema do else pendente

- Ex.: if (0) if (1) outra else outra




Qual é a interpretação correta?

# Problema do else pendente

- Normalmente se usa a regra do aninhamento mais próximo
  - Reescrevendo a gramática em BNF seria:

```
declaração → casam-decl | sem-casam-decl
casam-decl → if (exp) casam-decl else casam-decl |
             outra
sem-casam-decl → if (exp) declaração |
                 if (exp) casam-decl else sem-casam-decl
exp → 0 | 1
```

casam-decl ocorre antes de um else em uma declaração if,
o que força as partes else a casar assim que possível

# Problema do else pendente

- A regra do aninhamento mais próximo em BNF em geral não é construída e a opção preferida é aplicar a regra de eliminação de ambiguidade
  - É mais fácil configurar o analisador sintático para fazer obedecer à regra do aninhamento mais próximo como uma "exceção"

# Problema do else pendente

- Há linguagens de programação que são projetadas para que o problema do else pendente não ocorra
  - Lisp: exige a presença da parte else
  - Algol60 e Ada: usam palavra-chave de marcação de bloco para o if (exemplo end if)

```
if-decl → if condição then seq-decl end if |
          if condição then seq-decl else seq-decl end if
```

# Análise Sintática

- Bibliografia consultada

Capítulo 3: LOUDEN, K. C. **Compiladores: princípios e práticas.** São Paulo: Pioneira Thompson Learning, 2004