# CTC-41

# Compiladores

**Carlos Alberto Alonso Sanches**

# CTC-41

# 3) Análise sintática

Notações para gramáticas, derivações,
árvores sintáticas, ambiguidades

# Análise sintática

- A principal tarefa do analisador sintático (*parser*) é, a partir dos *tokens* recebidos, reconhecer a linguagem que está sendo compilada e construir a sua correspondente *árvore sintática*.
- Um problema decorrente desta tarefa é o tratamento de erros: além de registrar uma mensagem, o *parser* precisa se recuperar e continuar o seu trabalho.
- Geralmente, as linguagens de programação são livres de contexto (tipo 2):
  - Além de incluírem as características das linguagens regulares, permitem também a recursão. É um ganho muito importante, porque as sintaxes das linguagens de programação costumam ser definidas de forma recursiva.
  - Como veremos a seguir, são chamadas de livres de contexto porque a parte esquerda das suas regras gramaticais (*produções*) possuem um único terminal.
  - No entanto, isto inviabiliza, entre outras coisas, a verificação de declarações de variáveis e assinaturas de funções. Por questões de simplicidade, essas tarefas e outras similares são deixadas para a posterior análise semântica.

# Gramáticas livres de contexto

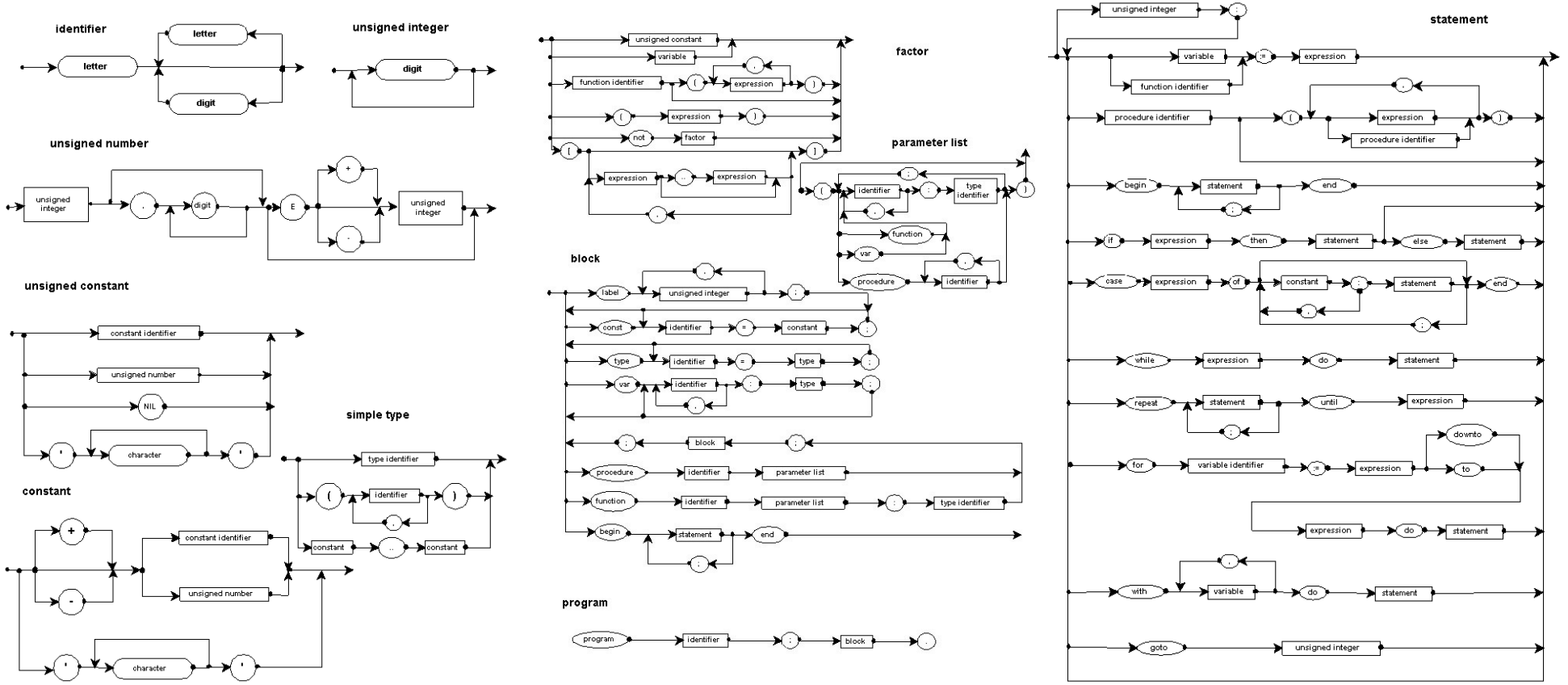
- Uma gramática *livre de contexto* é uma quádrupla $G = (T, N, P, S)$, onde:
  - T é um conjunto de terminais;
  - N é um conjunto de não-terminais, disjunto de T;
  - P é um conjunto de produções na forma $A \rightarrow \alpha$, onde $A \in N$ e $\alpha \in (T \cup N)^*$;
  - $S \in N$ é um símbolo inicial.
- O conjunto de símbolos de G é $T \cup N$.
- Dada $A \rightarrow \alpha \in P$, a cadeia (ou sentença) $\alpha \in (T \cup N)^*$ é uma *forma sentencial* de G. Podemos ter $\alpha = \varepsilon$.
- $\beta A \gamma \Rightarrow_G \beta \alpha \gamma$ é um *passo de derivação* sobre G, onde $\alpha, \beta, \gamma \in (T \cup N)^*$ e $A \rightarrow \alpha \in P$. Em outras palavras, é a substituição de um não-terminal de acordo com uma produção.
- Considerando $\Rightarrow_G^*$ como o fechamento reflexivo e transitivo de $\Rightarrow$, a linguagem livre de contexto $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$ é o conjunto de sentenças deriváveis de G.
- $S \Rightarrow_G^* w$ é uma derivação à esquerda se $\beta \in T^*$ em cada passo $\beta A \gamma \Rightarrow \beta \alpha \gamma$.
- $S \Rightarrow_G^* w$ é uma derivação à direita se $\gamma \in T^*$ em cada passo $\beta A \gamma \Rightarrow \beta \alpha \gamma$.

# Processo de compilação

- A principal diferença de uma gramática *sensível ao contexto* é permitir produções $\beta A \gamma \rightarrow \beta \alpha \gamma$, onde $A \in N$; $\beta, \gamma \in (T \cup N)^*$; $\alpha \in (T \cup N)^+$.
- As gramáticas sensíveis ao contexto são mais poderosas: as derivações de cada não-terminal dependem do seu contexto.
  - Na notação acima, a derivação $A \rightarrow \alpha$ depende de $\beta$ e $\gamma$.
- De acordo com a hierarquia de Chomsky, as linguagens geradas por gramáticas sensíveis ao contexto (tipo 1) englobam as linguagens geradas por gramáticas livres de contexto (tipo 2).
- Por razões de simplicidade, essas gramáticas não são utilizadas nas definições das linguagens de programação, deixando para o analisador semântico algumas tarefas de verificação de contexto.
- No processo de compilação, os terminais são os *tokens*, enquanto os não-terminais indicam as regras sintáticas.

# Diagramas sintáticos

- Antigamente, era comum descrever a sintaxe de uma linguagem de programação através de diagramas sintáticos.
- Exemplo para a linguagem Pascal:



# Forma Normal de Backus (BNF)

- Um formalismo amplamente utilizado na descrição de gramáticas livres de contexto é a *Forma Normal de Backus* (BNF), também chamada de *Forma Backus-Naur*.
- Esta notação possui primitivas de alternativa, concatenação e recursão.
- Exemplo de BNF para uma simples expressão aritmética **exp**:
  - **exp → exp op exp | (exp) | int**
    **op → + | - | * | /**
    **int → digit | int digit**
    **digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
- A primitiva de repetição pode ser expressa através da recursão.
- Exemplos de BNF para a linguagem $\{a^n, n \geq 1\}$:
  - **A → Aa | a** (recursiva à esquerda)
  - **A → aA | a** (recursiva à direita)

# Linguagem e derivações

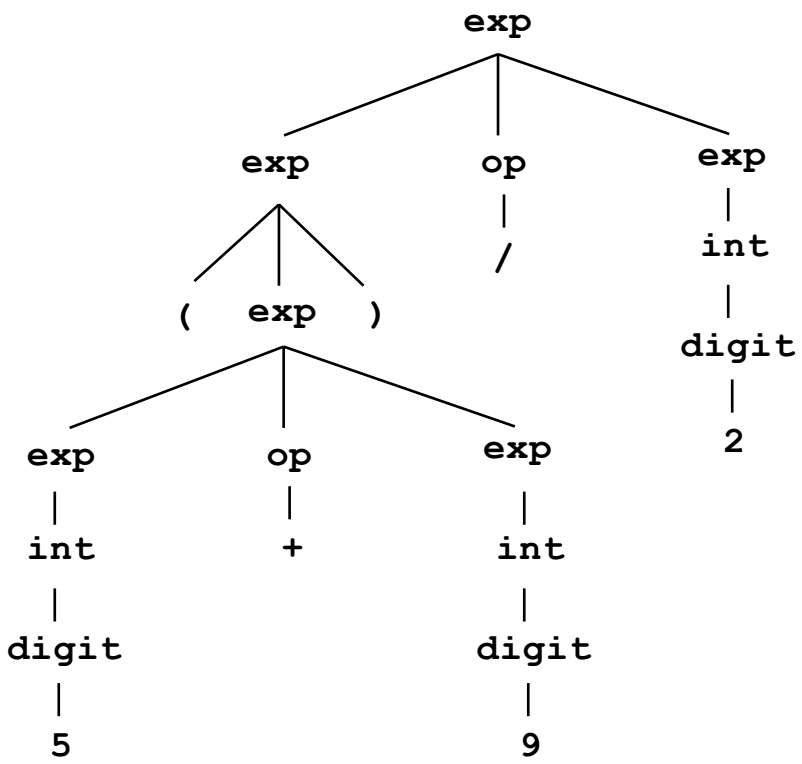
- A linguagem definida por uma gramática corresponde ao conjunto de cadeias de terminais obtido através de derivações válidas.
- Dada a gramática $G = (T, N, P, S)$, $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$.
- Consideremos novamente a gramática abaixo:
  - `exp → exp op exp | (exp) | int`
    `op → + | - | * | /`
    `int → digit | int digit`
    `digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
- Exemplo de derivação da cadeia `(5+9)/2`:
  - `exp ⇒ exp op exp ⇒ exp op int ⇒ (exp) op int ⇒`
    `(exp op exp) op int ⇒ (int op exp) op int ⇒`
    `(int op int) op int ⇒ (int + int) op int ⇒`
    `(int + int) / int ⇒ (digit + int) / int ⇒`
    `(digit + digit) / int ⇒ (digit + digit) / digit ⇒`
    `(5 + digit) / digit ⇒ (5 + 9) / digit ⇒ (5 + 9) / 2`
- Pode haver mais de uma sequência de derivações para uma determinada cadeia.

# Árvore sintática

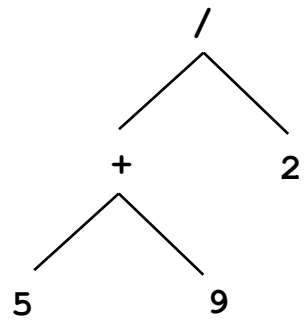
- Uma sequência de derivações pode ser representada em uma *árvore sintática* :
  - a raiz é o símbolo inicial da gramática;
  - cada nó interno é um não-terminal;
  - cada folha é um terminal ou ε;
  - se um nó interno $A \in N$ tiver n filhos $X_1, X_2, \ldots, X_n$, então $A \rightarrow X_1X_2\ldots X_n \in P$.
- De modo geral, sequências distintas de derivações podem ser representadas por uma mesma árvore sintática.
- Cada árvore sintática tem uma única derivação à esquerda e uma única derivação à direita.
- As árvores sintáticas estruturam os *tokens* (são suas folhas) e representam os passos de derivação da análise sintática. Cada nó possui campos com atributos úteis no processo de compilação.
- Dependendo do modo como são construídas, caracterizam *parsers* ascendentes ou descendentes.

# Exemplo

- Consideremos novamente a gramática abaixo:
  - `exp → exp op exp | ( exp ) | int`
    `op → + | - | * | /`
    `int → digit | int digit`
    `digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`
- Árvore sintática de `(5+9)/2`:

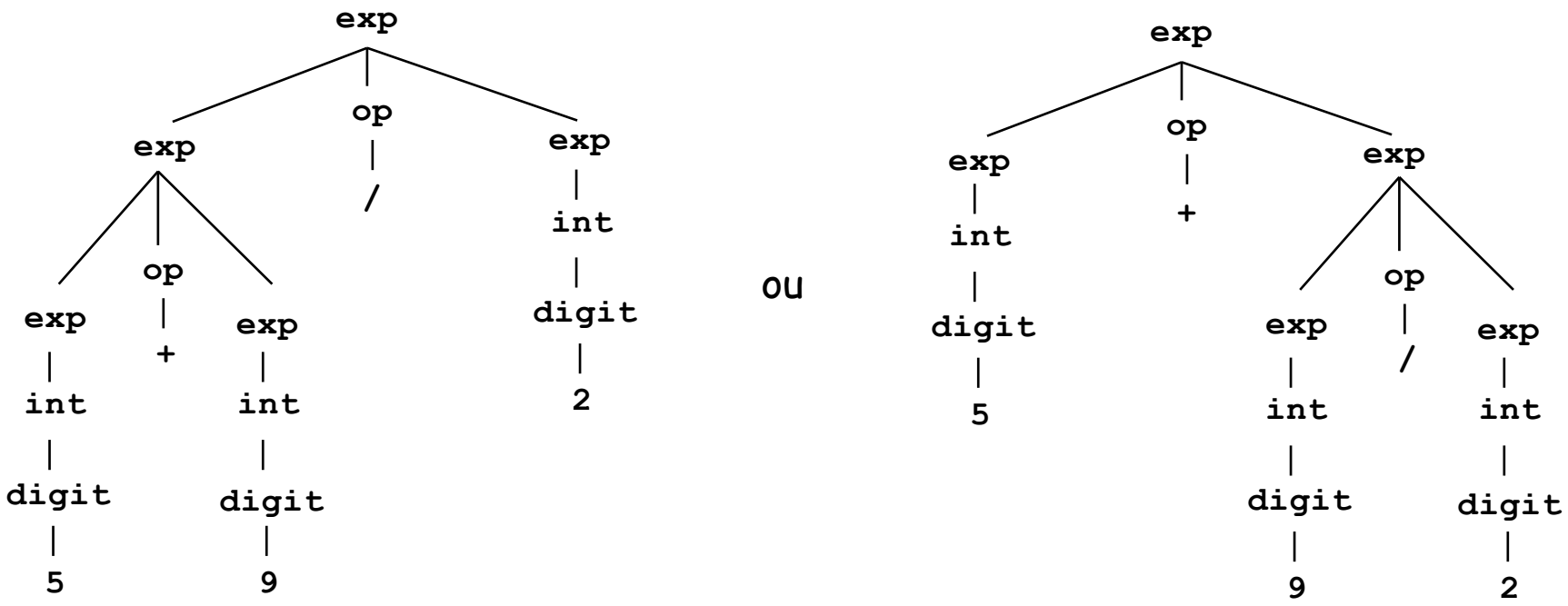



Em um formato simplificado:



# Ambiguidade

- No entanto, a sentença `5+9/2` pode ter duas árvores sintáticas:




- Uma gramática é *ambígua* se ao menos uma das suas sentenças possui duas ou mais árvores sintáticas distintas.
  - É uma característica indesejável, pois torna imprecisa a análise sintática. No exemplo acima, as árvores geram resultados distintos: 7 ou 9,5.
  - Ou se estabelecem regras que definem uma única árvore sintática para cada sentença, ou se altera a gramática.

# Eliminação de ambiguidades

- No exemplo anterior, uma possível forma de eliminar a ambiguidade seria estabelecer regras de precedência entre os operadores.
  - Dando precedência à divisão, 5+9/2 teria resultado 9,5.
  - No entanto, permaneceria ambiguidade em 5-9-2: resultado pode ser -6 ou -2. Seria preciso estabelecer também regras de associatividade.
- Outra solução: exigir parênteses
  - ```
    exp → fator op fator | fator
    fator → ( exp ) | int
    op → + | - | * | /
    int → digit | int digit
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    ```
  - No entanto, além de alterar a gramática, haverá alteração na linguagem reconhecida. Por exemplo, 5+9/2 deixaria de ser uma cadeia válida.
- A seguir, veremos como alterar esta gramática sem mudar a linguagem reconhecida, estabelecendo regras de precedência e de associatividade entre os operadores.
    ```

# Reescrevendo a gramática

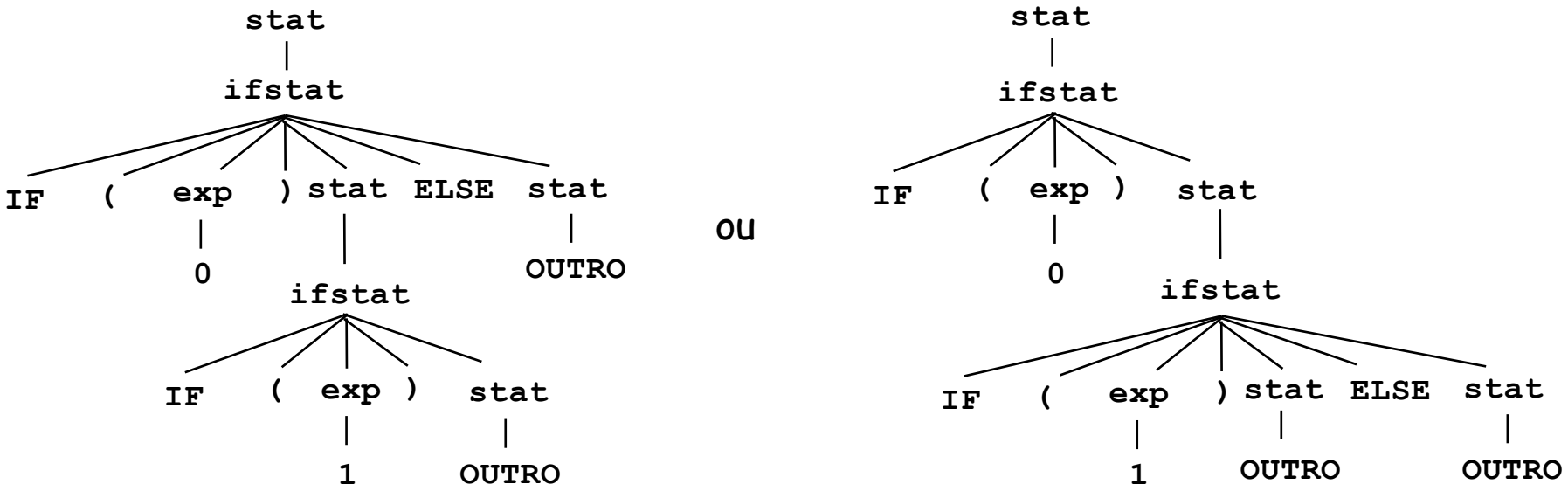
- Precedência: agrupar operadores de mesma precedência, deixando no topo da árvore sintática os de menor precedência.
  - ```
    exp → exp op1 exp | termo
    op1 → + | -
    termo → termo op2 termo | fator
    op2 → * | /
    fator → ( exp ) | int
    int → digit | int digit
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    ```
    + e - com menor precedência

    * e / com maior precedência
- Associatividade: alterar as recursões para forçar associatividade à esquerda ou à direita.
  - ```
    exp → exp op1 termo | termo
    op1 → + | -
    termo → fator op2 termo | fator
    op2 → * | /
    fator → ( exp ) | int
    int → digit | int digit
    digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    ```
    + e - associativos à esquerda

    * e / associativos à direita

# Outro caso de ambiguidade

- Considere a gramática abaixo:
  - `stat → ifstat | OUTRO`
    `ifstat → IF ( exp ) stat | IF ( exp ) stat ELSE stat`
    `exp → 0 | 1`
- Há duas árvores para a sentença `IF (0) IF (1) OUTRO ELSE OUTRO` :




- No primeiro caso, `OUTRO` é executado; no segundo, não.
- É o conhecido problema do `ELSE` pendente: pode ser associado a ambos comandos `IF`.

# Soluções para este problema

- Uma solução para esta ambiguidade é a regra do aninhamento mais próximo, ou seja, associar o `ELSE` ao último `IF`.
  - ```
    stat → casado | naocasado
    casado → IF ( exp ) casado ELSE casado | OUTRO
    naocasado → IF ( exp ) stat | IF ( exp ) casado ELSE naocasado
    exp → 0 | 1
    ```
- Nos comandos `IF`, `casado` ocorre antes de `ELSE`: isso força os `ELSE` a se casarem assim que possível.
- Outras soluções:
  - O *parser* poderia ser configurado para obedecer à regra do aninhamento mais próximo como uma exceção, sem necessidade de alterar a gramática.
  - A linguagem LISP exige a presença do `ELSE`, mesmo vazio.
  - Outras linguagens usam um *token* específico para finalizar o comando `IF`:
    - ```
      stat → ifstat | outro
      ifstat → IF ( exp ) stat ENDIF | IF ( exp ) stat ELSE stat ENDIF
      exp → 0 | 1
      ```