ICC043/IEC582 – Paradigmas de Linguagens de Programação

# Gramáticas, BNF e Hierarquia de Chomsky



**Prof. Dr. Rafael Giusti**
rgiusti@icomp.ufam.edu.br

# Leitura recomendada e suplementar

- **Sebesta**. Conceitos de Linguagens de Programação:
  - Capítulo 3: especificando sintaxe e semântica
  - Capítulo 4: análise léxica e sintática
- **Lewis e Papadimitriou.** *Elements of the Theory of Computation*:
  - Capítulo 1: *Sets, Relations, and Languages*
  - Em particular:
    - Seção 1.7: *Alphabets and Languages*

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Introdução

- O que é uma linguagem natural?
  - Português, inglês, espanhol
  - Ortografia? Gramática? Semântica?
    - Vovô viu a uva.
    - Vovô viram o uva.
  - Linguagens naturais são... naturais
    - Ortografia, gramática e semântica são observações sobre uma linguagem

# Introdução

- Linguagem formal
  - Possui um conjunto de regras sintáticas e semânticas usadas para definir uma forma de comunicação correta
- Na computação
  - Quais instruções o computador entende? Como serão armazenadas ou transmitidas?
  - Deve ser utilizada por várias pessoas, ou seja, inteligível

# Sintaxe e semântica

- Sintaxe
  - Forma como as instruções, expressões e unidades de programa de uma linguagem são escritas
- Semântica
  - Significado de suas expressões, instruções e unidades de programas
  - Ambas estão estreitamente relacionadas
  - A semântica deve seguir diretamente da sintaxe

# Sintaxe e semântica

Sintaxe:

```
if (expressão)
    comando;
```

Semântica:

Se o valor da *expressão* for verdadeiro, então execute o *comando* condicionado.

Sintaxe:

```
variável++;
```

Semântica:

A expressão resulta no valor da *variável*. Após a avaliação, o valor da *variável* é incrementado.

# Sintaxe e semântica

- Se a semântica não segue diretamente da sintaxe, a linguagem pode ser complexa e difícil de entender
- Exemplo em C:
  - A ordem de avaliação dos operandos não é especificada

```
int i = 0;
int j = i++ + i;
```

  - Se o valor de uma expressão depende da ordem de avaliação dos operandos, o comportamento do programa é indefinido

# Especificação de sintaxe

- Especificar a sintaxe da linguagem é mais simples do que a semântica
  - Sintaxe: notação concisa e universal
    - Gramáticas e expressões regulares
    - Notações BNF e EBNF
  - Semântica: não
    - Gramáticas de atributos fazem parte
    - Semântica pragmática (linguagem natural) é o jeito mais comum

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Gramáticas

- Conceito introduzido na linguística
  - O objetivo era desenvolver tradutores automáticos de idiomas
  - Impulsionou compiladores na década de 1960

Noam Chomsky

- Apesar não ser bem sucedido na tradução automática, a teoria de linguagens formais é de grande importância na computação

https://en.wikipedia.org/wiki/Noam_Chomsky#/media/File:Noam_Chomsky_portrait_2017.jpg

# Conceitos

- Um alfabeto ou vocabulário é um conjunto finito e não vazio de símbolos
  - $\Sigma = \{A, B, C, \ldots, Z\}$
    - Alfabeto romano
  - $\Sigma = \{0, 1\}$
    - Alfabeto binário
  - $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
    - Alfabeto de dígitos

# Conceitos

- Uma cadeia é uma sequência de símbolos
  - ABOBORA, BANANA
    - Cadeias do alfabeto romano
  - 1010100101, 111001101
    - Cadeias do alfabeto binário

- Uma cadeia que não contém nenhum símbolo é a cadeia vazia ou nula
  - ε ou λ – cadeia vazia

# Operações sobre alfabetos

- O comprimento de uma cadeia é o número de símbolos que ela contém
  - $|abc| = 3$
  - $|123456789| = 9$
  - $|\lambda| = 0$

# Operações sobre alfabetos

- A concatenação de duas cadeias é a concatenação dos símbolos das cadeias
  - Representada pelo operador $\circ$ ou pela simples justaposição das cadeias
    - $x = abc$ $\quad |x| = 3$
    - $y = def$ $\quad |y| = 3$
    - $x \circ y = xy = abcdef$ $\quad |xy| = |x| + |y| = 6$

Normalmente usamos as "primeiras" letras (*a*, *b*, *c*, ...) do alfabeto para símbolos e as "últimas" (*w*, *x*, *y*, *z*) para cadeias

# Operações sobre alfabetos

- A exponenciação de cadeias é equivalente à concatenação repetida
  - $w^0 = \lambda$
  - $w^i = ww^{i-1}$ $\quad (i > 0)$
- Exemplo
  - $(abc)^0 = \lambda$
  - $(abc)^1 = abc(abc)^0 = abc\lambda = abc$
  - $(abc)^2 = abc(abc)^1 = abcabc$

# Operações sobre alfabetos

- A exponenciação de alfabetos produz todas as cadeias de comprimento 0, 1, 2, ... que podem ser formadas com concatenação dos símbolos

- O fecho de um alfabeto é a união de todas as exponenciações de tamanhos 0, 1, 2, ...
  - $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \Sigma^4 \ldots$
- Exemplo
  - $\{0, 1\}^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$

# Linguagens

- Uma **linguagem** $L$ sobre um alfabeto $\Sigma$ é um conjunto $L \subseteq \Sigma^*$
  - É um conjunto de cadeias sobre o alfabeto
  - As cadeias que pertencem à linguagem são também chamadas sentenças da linguagem
  - Uma linguagem pode ser finita ou infinita
  - Normalmente descrita através de notação implícita de conjuntos
    - A = $\{x$ : condição envolvendo $x\}$

# Linguagens

- Exemplos:
  - A linguagem dos números binários pares
    - $\Sigma = \{0, 1\}$
    - $L = \{w \in \Sigma^* : \text{o último símbolo de } w \text{ é } 0\}$
  - A linguagem que contém as cadeias $a$, $ab$ e $bc$
    - $L = \{a, ab, bc\}$
  - A linguagem que contém as cadeias com zeros à esquerda e uns à direita
    - $L = \{0^n 1^m : n, m > 0\}$

# Linguagens

- Exemplos:
  - A linguagem vazia
    - $L = \varnothing$
  - A linguagem que contém todas as cadeias que contém o símbolo $a$ à esquerda e o símbolo $b$ à direita e, além disso, apenas um dos símbolos pode ocorrer múltiplas vezes
    - $L = \{ab^n \cup a^nb : n > 0\}$

# Operações com linguagens

- A concatenação de duas linguagens resulta na concatenação de todas as suas cadeias
  - Exemplo:
    - $L_1 = \{aba\}, L_2 = \{cate, caxi\}$
    - $L_1 \circ L_2 = \{abacate, abacaxi\}$
  - Exemplo:
    - $L_1 = \{w \in \{0,1\}^* : w \text{ tem número par de 0's}\}$
    - $L_2 = \{01^n : n \geqslant 0\}$
    - $L_1 L_2 = \{w : w \text{ tem um número ímpar de 0's}\}$

# Operações com linguagens

- O fecho de uma linguagem $L$ é a união de todas as exponenciações $L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$
  - Exemplo:
    - $L = \{ab, c\}$
    - $L^* = \{\varepsilon, ab, c, abc, cab, ababc, abcab, abcc, cabab, cabc, ccab, \ldots\}$

# Operações com linguagens

- O fecho sob concatenação de L é a concatenação da linguagem com seu fecho
  - $L^+ = LL^*$
- Exemplo:
  - $L = \{ab, c\}$
  - $L^+ = \{ab, c, abc, cab, ababc, abcab, abcc, cabab, cabc, ccab, \ldots\}$

Se *L* não for a linguagem vazia,
então *L*+ = *L* – {ε}

# Linguagens em compiladores

- A primeira parte da compilação ou interpretação de um programa é sua análise léxica
  - Nessa etapa, o programa é "quebrado" em elementos fundamentais de acordo com os *tokens* da linguagem
    - Identificadores, operadores, literais etc.

Podemos utilizar **expressões regulares** para especificar o **léxico** da linguagem

# Análise léxica

- O léxico consiste de lexemas e *tokens*
  - Lexema:
    - Menor unidade sintática do programa
  - *Token*:
    - Uma classe de lexemas

```
int num = 42;
```

| Lexemas | Tokens |
|---|---|
| `int` | tipo_int |
| num | identificador |
| = | op_atrib |
| 42 | literal_int |
| ; | op_term |

# Expressões regulares

- Uma expressão regular (ER) é uma linguagem que pode ser descrita por meio de concatenação, união e fecho de outras ER
  - $\varnothing$ é uma ER – linguagem vazia
  - Qualquer símbolo $x$ é uma ER – linguagem $\{x\}$
  - Se $r$ e $s$ são ER para as linguagens $R$ e $S$...
    - $rs$ é uma ER – linguagem $R \circ S$
    - $(r+s)$ ou $(r|s)$ é uma ER – linguagem $R \cup S$
    - $r^*$ é uma ER – linguagem $R^*$

# Expressões regulares

- Exemplos:
  - `00` denota a linguagem {00}
  - `(0+1)*` é a linguagem dos dígitos binários e inclui a cadeia vazia
  - `0+1(0+1)*` é a linguagem dos dígitos binários, excluindo zeros à esquerda
  - `(0+1)*00(0+1)*` é a linguagem de todas as cadeias de 0s e 1s que contém ao menos duas ocorrências consecutivas do 0

# Análise léxica

- O léxico de uma linguagem de programação pode ser especificado por meio de expressões regulares
- Durante a etapa de análise léxica, o compilador ou interpretador "consome" a entrada, verificando se as cadeias fazem parte da linguagem léxica

```
int num = 42;
```

# Especificando o léxico

- Identificadores e palavras reservadas
  - $r$ = (A+B+C+...+Z+a+b+c+...+z+_)
  - $s$ = (0+1+2+...+9)
  - $t = r(r+s)^*$
    - Uma única expressão regular é utilizada para identificadores e palavras reservadas
    - Ao verificar que uma cadeia pertence a essa linguagem, o compilador pesquisa o lexema em uma tabela para verificar se é uma palavra reservada

# Especificando o léxico

- Literais de inteiros em Python 3
  - $b$ = `0b(0+1)(0+1)*`
  - $o$ = `0o(0+...+7)(0+...+7)*`
  - $h$ = `0x(0+...+F)(0+...+F)*`
  - $d$ = `(1+...+9)(0+...+9)`
  - $s = b+o+h+d$

* Em Python 2 os octais seguem a forma *o* = `0(0+...+7)(0+...+7)*`

# Especificando o léxico

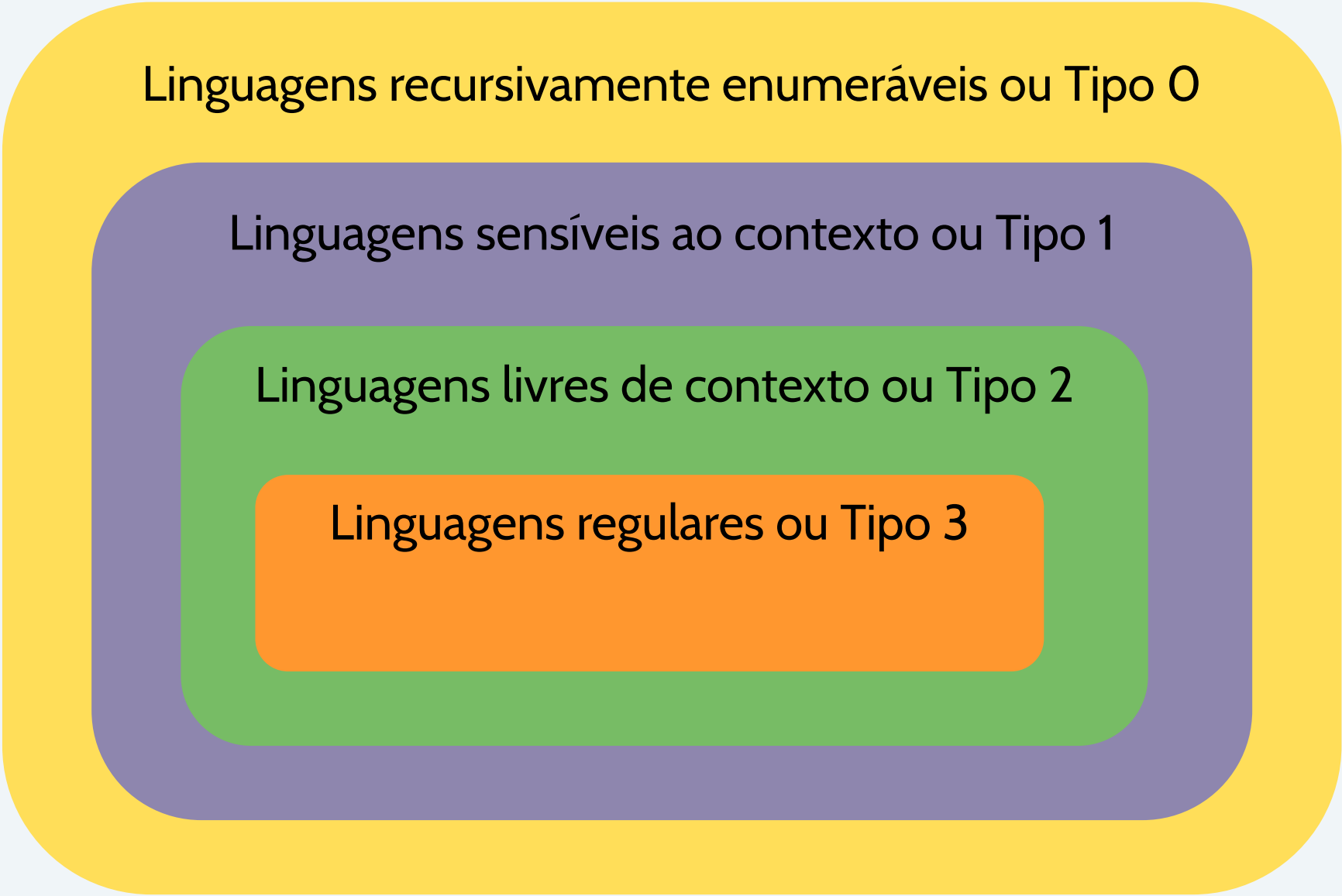
- Operadores em C (parcial)
  - $a$ = "+" | - | "++" | --
  - $m$ = "*" | / | %
  - $l$ = ~ + & + ^ + "|" + "&&" + "||"
  - $r$ = < + "<=" + > + ">=" + "=="

A cadeia "++" será reconhecida como duas ocorrências do símbolo "+" ou uma ocorrência do símbolo "++"?

# Tipos de linguagens

- Nem todas as linguagens são iguais
  - As linguagens das expressões regulares parecem "limitadas"
    - Contêm apenas cadeias que são concatenação e união de expressões regulares
    - Não podem representar, por exemplo, a linguagem dos parênteses balanceados
    - Ou a linguagem na qual um rótulo <a> precisa ser fechado por um rótulo </a> equivalente

# Hierarquia de Chomsky



# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Gramáticas

- As linguagens regulares podem ser descritas por meio de expressões regulares
  - Podem representar o léxico das LP
- Linguagens livres de contexto podem ser descritas por meio de notação BNF ou EBNF
  - Podem representar a sintaxe das LP
- Mas, de maneira geral, todas as linguagens podem ser descritas por meio de gramáticas

# Gramáticas

- Todas as gramáticas possuem
  - Um alfabeto ou conjunto de símbolos terminais
  - Um conjunto de símbolos não terminais
  - Um conjunto de regras de produção
  - Um símbolo inicial
- A gramática é um mecanismo gerador
  - Ela gera uma linguagem aplicando transformações sobre o símbolo inicial

# Gramáticas

- Formalmente, uma gramática é uma quádrupla
  - $G = (N, \Sigma, P, S)$
    - $N$ é o conjunto de símbolos não terminais
    - $\Sigma$ é o alfabeto (conjunto de símbolos terminais)
    - $P$ é o conjunto de regras
    - $S$ é o símbolo inicial
  - A gramática gera uma linguagem derivando cadeias a partir do símbolo inicial
    - Ao final, temos uma sentença

# Gramática

- Exemplo:
  - $G = (N, \Sigma, P, S)$
    - $N = \{S, A, B\}$
    - $\Sigma = \{a, b\}$
    - $P = \{S \rightarrow Ab \mid aB,$
      $A \rightarrow aA \mid \varepsilon,$
      $B \rightarrow bB \mid \varepsilon \}$
  - Produz a linguagem $L(G) = \{a^n b^m : (n = 1 \text{ e } m > 0) \text{ ou } (n > 0 \text{ e } m = 1) \}$

# Derivações

- Para sabermos o que uma gramática é capaz de produzir, podemos realizar uma derivação
  - Uma derivação é a aplicação repetida das regras de produções
  - Começamos com uma forma sentencial
    - Pode ser um não terminal ou uma combinação de terminais e não terminais
  - Usando as regras, "refinamos" a forma sentencial até que sobrem apenas terminais
    - O resultado é uma sentença

# Derivações

- Gramáticas são mecanismos geradores, mas podem ser utilizadas para reconhecer sentenças
  - Para saber se uma cadeia $w \in \Sigma^*$ é uma sentença da linguagem, verificamos se existe uma derivação que gera $w$
  - Exemplo (gramática do slide 41):
    - aab é uma sentença, pois existe a derivação
      - S => Ab => aAb => aaAb => aab
    - ba não é uma sentença, pois não há nenhuma derivação que produza um b antes de um a

# Gramáticas e hierarquia de Chomsky

- As gramáticas possuem tipos equivalentes às linguagens
  - Gramáticas regulares
  - Gramáticas livres de contexto
  - Gramáticas dependentes de contexto
  - Gramáticas irrestritas
    - Produzem linguagens recursivamente enumeráveis

# Gramáticas e hierarquia de Chomsky

- Nas gramáticas regulares (GR), as produções são limitadas àquelas equivalentes a expressões regulares
  - Do lado esquerdo só pode aparecer um símbolo não terminal
  - Do lado direito, só pode haver
    - Um único símbolo não terminal; ou
    - Um símbolo não terminal à direita de um símbolo terminal*

*Ou um símblo não terminal à esquerda

# Gramática

- Exemplo:
  - $G = (N, \Sigma, P, S)$
    - $N = \{S, M, N, E\}$
    - $\Sigma = \{0, 1\}$
    - $\mathrm{P} = \{S \rightarrow 0S \mid 1S \mid M,$
      $M \rightarrow 0N,$
      $N \rightarrow 0E \mid 0,$
      $E \rightarrow 0E \mid 1E \mid 0 \mid 1\ \}$
  - Produz a linguagem $L(G) =$ `(0+1)*00(0+1)*`

# Gramáticas e hierarquia de Chomsky

- Nas gramáticas livres de contexto (GLC), as produções são um pouco menos limitadas
  - Do lado esquerdo só pode aparecer um símbolo não terminal
  - Do lado direito pode haver qualquer combinação de símbolos terminais e não terminais

# Gramáticas e hierarquia de Chomsky

- Exemplo:
  - $G = (N, \Sigma, P, S)$
    - $N = \{S\}$
    - $\Sigma = \{\mathrm{a}, \mathrm{b}\}$
    - $\mathrm{P} = \{S \rightarrow \mathrm{a} \mid \mathrm{b} \mid \mathrm{a}S\mathrm{a} \mid \mathrm{b}S\mathrm{b} \}$
  - Produz:
    - $L(G) = \{x \in \Sigma^* \mid x \text{ é um palíndromo} \}$

# Gramáticas e hierarquia de Chomsky

- Exemplo:
  - $G = (N, \Sigma, P, S)$
    - $N = \{S\}$
    - $\Sigma = \{\mathrm{a}, \mathrm{b}\}$
    - $\mathrm{P} = \{S \rightarrow \mathrm{ab} \mid \mathrm{aSb}\ \}$
  - Produz:
    - $L(G) = \{\ a^n b^n : n > 0\ \}$

# Exercício 2

A) Escreva uma ER para as cadeias sobre $\{a, b, c\}$ que contêm pelo menos um $a$ e um $b$

B) Escreva uma ER para as cadeias sobre $\{a, b\}$ tais que a sequência $aa$ nunca apareça à direita de uma sequência $bb$

C) Escreva uma gramática livre de contexto para as sequências balanceadas de parênteses

- Exemplo: `()`, `()()`, `(())`, mas não `())`, `)(` etc.

D) Escreva uma gramática regular equivalente a alguma das duas expressões regulares acima

# Diferenças entre gramáticas

- Gramáticas livres de contexto são mais poderosas que gramáticas regulares
  - A linguagem $a^n b^n$, por exemplo, requer uma gramática livre de contexto (para $n$ qualquer)
    - Não há mecanismo em ER que permita gerar duas cadeias de tamanhos sempre iguais
  - Porém a GLC não consegue representar uma linguagem como $a^n b^n c^n$
    - Ela não consegue representar o contexto de que $b^n$ está "cercada" por duas cadeias de tamanhos iguais

# Diferenças entre gramáticas

- As gramáticas dependentes de contexto são mais poderosas que GR e GLC
  - Elas permitem que símbolos não terminais apareçam do lado esquerdo das produções
    - P = { S → abc | A,
      A → aABc | abc,
      cB → Bc,
      bB → bb }
    - S => A => aABc => aabcBc => aabBcc
      => aabbcc
  - Pode descrever uma linguagem como $a^n b^n c^n$

# Diferenças entre gramáticas

- Gramáticas irrestritas podem descrever quaisquer linguagens enumeráveis
  - Qualquer linguagem infinita, mas contável, pode ser gerada por uma gramática irrestrita
    - Em termos práticos, qualquer instância de um problema que pode ser resolvido por um computador hipotético
  - Qualquer cominação de símbolos terminais e não terminais pode aparecer à esquerda e à direita das regras

# Gramáticas estudadas

- As gramáticas são importantes no estudo formal de computabilidade
- Problemas computacionais podem ser descritos como problemas de linguagem
  - Um grafo G contém um ciclo?
  - Podemos representar o grafo G como uma sentença da linguagem dos grafos que possuem um ciclo?
- Linguagens aparecem no estudo *do que* os computadores são capazes de calcular e também do *custo computacional* necessário

# Gramáticas estudadas

- No contexto das linguagens de programação, gramáticas e linguagens são importantes para descrever os programas
  - Não apenas se um programa pertence à linguagem de programação
  - Mas principalmente identificar a estrutura desse programa

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Notação BNF

- A BNF (*Backus-Naur Form*) é derivada de uma notação criada por John Backus para descrever o ALGOL 58
  - Tem poder descritivo equivalente ao de gramáticas livres de contexto
  - Mas possui maior legibilidade
  - Utilizada para descrever tanto o léxico quanto a sintaxe das linguagens
  - Utilizada também para descrever comandos em manuais técnicos

# Notação BNF

- Não é necessário descrever a quádrupla, nem o alfabeto, apenas as produções (chamadas regras)
  - Os conjuntos de terminais e não terminais ficam implícitos na lista de regras
- Apenas **um** não terminal pode aparecer do lado esquerdo (mesma limitação de GLC)
- Os não terminais são descritos entre colchetes angulares
- Disjunções podem ser descritas com barra ou repetindo o não terminal à esquerda

# Notação BNF

- Exemplo: (comando if em Pascal)

```
<if_stmt> ::= if <expr_lógica> then <stmt>
<if_stmt> ::= if <expr_lógica> then <stmt>
                 else <stmt>
```

```
<if_stmt> ::= if <expr_lógica> then <stmt>
            | if <expr_lógica> then <stmt>
                 else <stmt>
```

# Notação BNF

- Exemplo: (fórmulas bem-formadas)

```
<fbf> ::= <var> | <const> | <compost>
<var> ::= P | Q | R | S | T | U
<const> ::= V | F
<compost> ::= (<fbf>) | ¬<fbf>
            | <fbf> <conect> <fbf>
<conect> ::= ∧ | ∨ | → | ↔ | <=>
```

# Derivações à esquerda e à direita

- Durante a derivação, os não terminais podem ser substituídos em qualquer ordem
- Porém, podemos realizar uma substituição sistemática
  - Na derivação mais à esquerda (*leftmost derivation*), sempre substituímos o não terminal que se encontra mais à esquerda
  - Equivalentemente, temos também a derivação mais à direita (*rightmost derivation*)

# Derivações à esquerda e à direita

- Exemplo de derivação mais à esquerda

```
<fbf> <conect> <fbf> => <var> <conect> <fbf>
        => P <conect> <fbf>
        => P ∧ <fbf>
        => P ∧ <const>
        => P ∧ V
```

- Exemplo de derivação mais à direita

```
<fbf> <conect> <fbf> => <var> <conect> <fbf>
        => <var> <conect> <const>
        => <var> <conect> V
        => <var> ∧ V
        => P ∧ V
```

# Exercício resolvido em aula

- Dada a gramática

```
<atrib> ::= <id> := <expr>

<id> ::= A | B | C

<expr> ::= <id> + <expr> | <id> * <expr>
          | ( <expr> ) | <id>
```

- Mostre a derivação mais à esquerda que produz a sentença
  - A := B + C

# Exercício 3

- Dada a gramática

```
<atrib> ::= <id> := <expr>

<id> ::= A | B | C

<expr> ::= <id> + <expr> | <id> * <expr>
          | ( <expr> ) | <id>
```

- Mostre a derivação mais à esquerda que produz a sentença
  - A := B * (A + C)

# Agenda

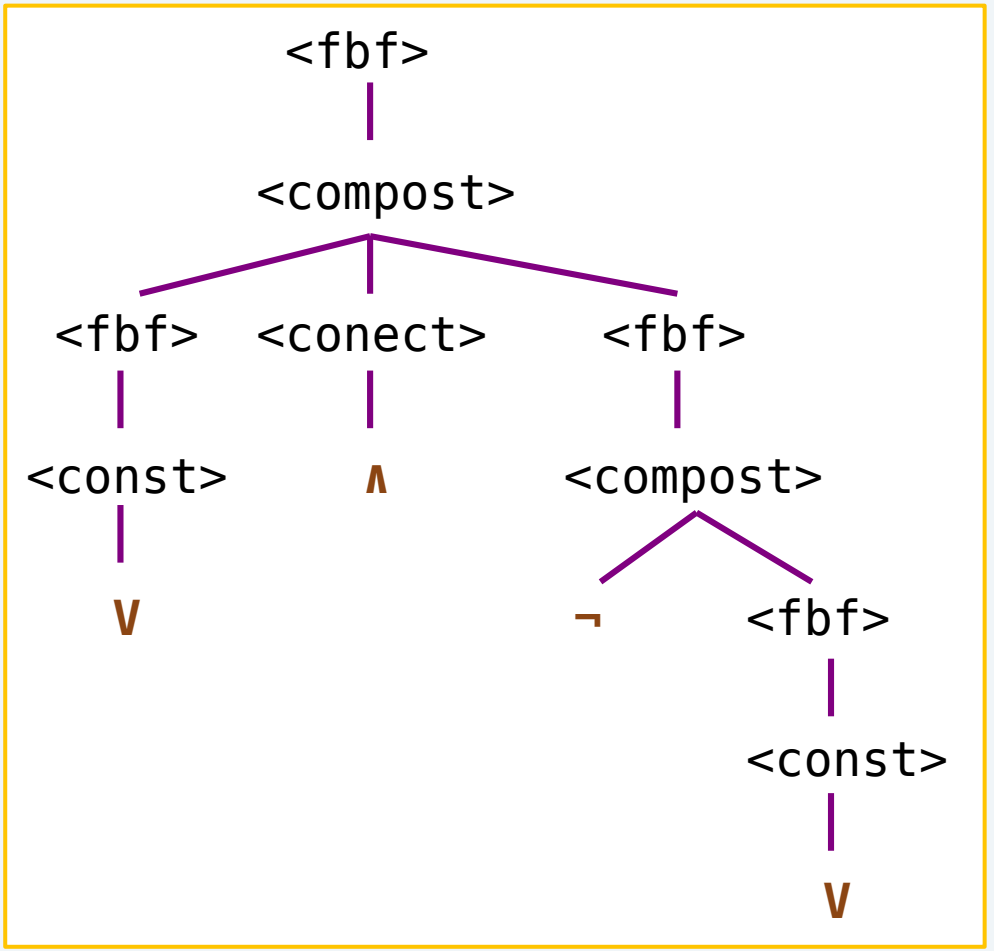
- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Árvores de derivação

- Uma derivação pode ser re-escrita na forma de uma árvore
  - A forma sentencial da derivação é colocada na raiz da árvore
  - Cada substituição de um não terminal produz uma sub-árvore
  - Tanto a derivação mais à esquerda quanto a mais à direita produzem a mesma árvore
    - A única diferença é a ordem em que desenhamos as sub-árvores

```
<fbf> ::= <var> | <const> | <compost>
<var> ::= P | Q | R | S | T | U
<const> ::= V | F
<compost> ::= (<fbf>) | ¬<fbf>
            | <fbf> <conect> <fbf>
<conect> ::= ∧ | ∨ | → | ↔ | <=>
```

```
<fbf> => <compost>
   => <fbf> <conect> <fbf>
   => <const> <conect> <fbf>
   => V <conect> <fbf>
   => V ∧ <fbf>
   => V ∧ <compost>
   => V ∧ ¬<fbf>
   => V ∧ ¬<const>
   => V ∧ ¬V
```



# Árvores de derivação

- A árvore de derivação também é conhecida como árvore de análise sintática
  - Descreve a estrutura hierárquica das sentenças (programas) de uma linguagem de programação
- O compilador/interpretador constrói uma árvore sintática
  - Essa geração pode ser implícita ou explícita
  - Mas deve ser única
    - Deseja-se que a gramática não seja ambígua

# Ambiguidade

- Uma gramática é ambígua se contém uma sentença que
  - Pode ser gerada por duas derivações mais à esquerda (ou mais à direita)
  - Ou que admite duas árvores de derivação distintas
- A existência de ambiguidade dificulta a análise do programa
  - Qual é a estrutura hierárquica se a gramática admite duas árvores diferentes?

# Ambiguidade do if..else

- O problema do "senão pendente" (*dangling else*)

```
<stmt> ::= if (<expr>) <stmt>
         | if (<expr>) <stmt> else <stmt>
```

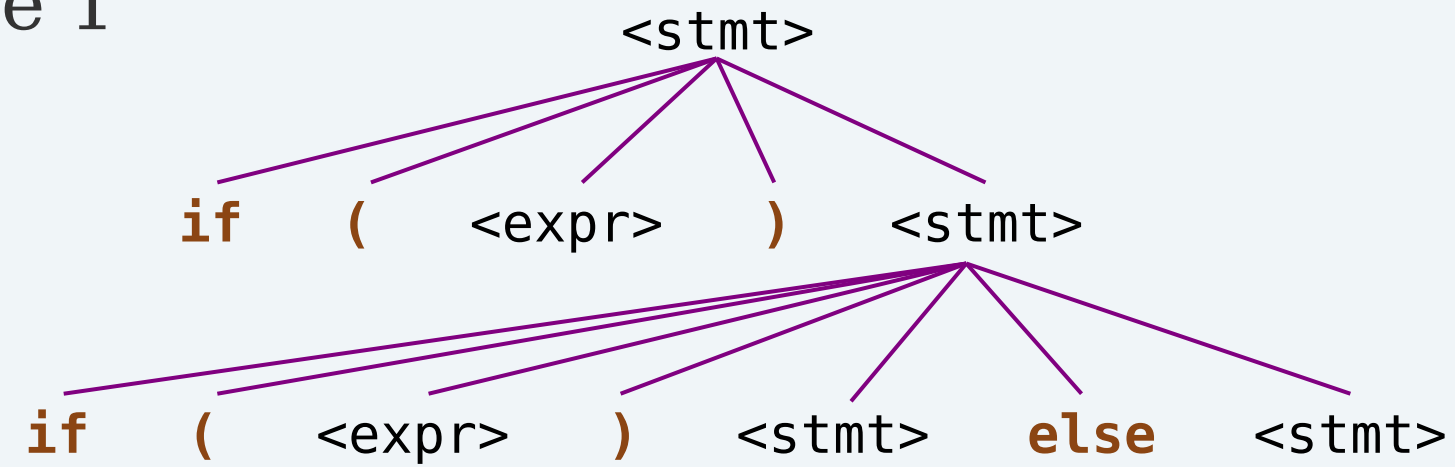
- Podemos mostrar a ambiguidade dessa gramática com a seguinte forma sentencial

```
if (<expr>) if (<expr>) <stmt> else <stmt>
```

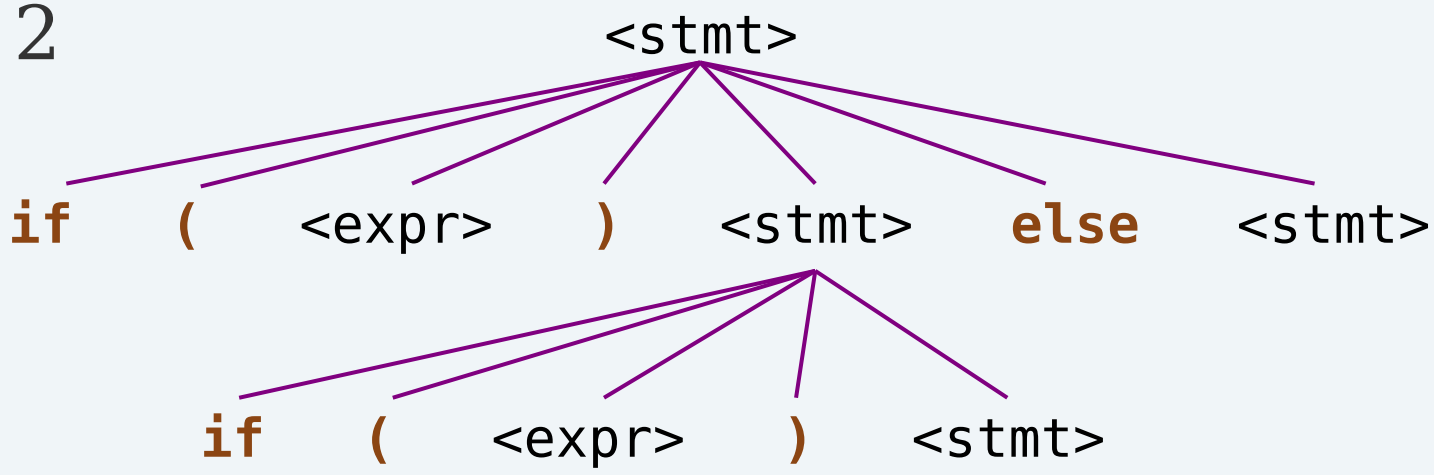
```
<stmt> ::= if (<expr>) <stmt>
         | if (<expr>) <stmt> else <stmt>
```

```
if (<expr>) if (<expr>) <stmt> else <stmt>
```

- Árvore 1




- Árvore 2



# Ambiguidade do if..else

- O "senão pendente" não deixa claro qual é a relação "se..senão" a ser seguida

```
if (done)
   if (denom == 0)
      result = 0;
   else
      result = num / denom;
```

  - Qual é a condição associada ao else?

# Ambiguidade do if..else

- Linguagens de programação não podem ser ambíguas
- Podemos resolver a ambiguidade de diversas formas
  - **Python** utiliza indentação para deixar a estrutura hierárquica do programa clara

```
if done:
   if denom == 0:
      result = 0
   else:
      result = num / denom;
```

# Ambiguidade do if..else

- Linguagens de programação não podem ser ambíguas
- Podemos resolver a ambiguidade de diversas formas
  - **COBOL** sempre associa um IF a um END-IF correspondente

```
IF Done
   IF Denom = 0
      SET Result TO 0
   ELSE
      DIVIDE Num INTO Denom GIVING Result
   END-IF
END-IF
```

# Ambiguidade do if..else

- Mas a ambiguidade do **if**..**else** pode ser resolvida sem a necessidade de modificar a sintaxe da linguagem
  - Em Pascal, o comando abaixo não é ambíguo
  - O **else** pendente sempre se liga com o **if** mais próximo

```
if Done then
   if Denom = 0 then
      Result := 0
   else
      Result := Num div Denom
```

# Resolvendo a ambiguidade do if..else

- A derivação do comando `if` é separada em dois não terminais
  - Um deles sempre produz um `if` que está casado
    - Sempre que derivarmos através desta regra, iremos produzir uma forma sentencial que possui um `if` associado a seu `else`
  - O outro sempre produz um `if` não casado
    - Por essa regra, temos uma forma sentencial na qual o `if` não está associado a nenhum `else`

# Resolvendo a ambiguidade do if..else

```
<stmt> ::= <matched> | <unmatched>

<matched> ::= if ( <expr> ) <matched> else <matched>
            | Qualquer comando, exceto outro if

<unmatched> ::= if ( <expr> ) <stmt>
              | if ( <expr> ) <matched> else <unmatched>
```

# Demonstração de ambiguidade

- De maneira geral, *não existe* um algoritmo capaz de identificar se uma gramática é ambígua
  - Determinar se uma gramática é ambígua ou não é um problema indecidível
- Em alguns casos, podemos mostrar que uma gramática específica não é ambígua
  - Normalmente apenas com gramáticas "bem--comportadas" e conhecendo sua estrutura

# Demonstração de ambiguidade

- Por outro lado, mostrar que uma gramática *é ambígua* é muito mais simples
  - Basta mostrar que existe uma forma sentencial para a qual temos
    - Duas derivações mais à esquerda
    - Duas derivações mais à direita
    - Ou duas árvores de derivação

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Gramáticas de expressões

- As gramáticas de expressões são um caso recorrente de gramáticas
  - Quase todas as linguagens de programação possuem regras que especificam expressões
    - Atribuições
      - `i := 0;`
    - Operações algébricas
      - `j + 1 => i`
    - Operadores associativos
      - `i = j++ + i;`

# Gramáticas de expressões

- Gramáticas livres de contexto são capazes de representar
  - Precedência de operadores
    - Os operadores que aparecem em "níveis mais profundos" da derivação têm maior precedência
  - Associatividade
    - Os operadores associam à esquerda ou à direita dependendo de como os não terminais são substituídos recursivamente

# Precedência de operadores

- Na gramática abaixo, o operador de multiplicação aparece "depois" do operador de soma
- Ele ficará "mais profundo" na derivação
  - Verifique isso construindo a árvore de derivação para a expressão `A + B * C`

```
<expr> ::= <expr> + <term> | <term>
<term> ::= <term> * <fator> | <fator>
<fator> ::= A | B | C
```

# Associatividade à esquerda

- Na gramática abaixo, o operador de divisão se associa da esquerda para a direita
  - Motivo: a recursão ocorre à esquerda
    - Construa a árvore de derivação para a expressão `A / B / C`
    - Veja que ela equivale a (A / B) / C

```
<term> ::= <term> * <fator>
         | <term> / <fator>
         | <fator>

<fator> ::= A | B | C
```

# Associatividade à direita

- Na gramática abaixo, o operador de potenciação se associa da direita para a esquerda
  - Motivo: a recursão ocorre à direita
    - Construa a árvore de derivação para a expressão `A ** B ** C`
    - Veja que ela equivale a A ** (B ** C)

```
<fator> ::= <atom> ** <fator>
          | <atom>

<atom> ::= A | B | C
```

# Uma gramática de expressões

```
<atrib> ::= <id> = <expr>

<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>

<term> ::= <term> * <fator>
         | <term> / <fator>
         | <term> % <fator>
         | <fator>

<fator> ::= <atom> ** <fator>
          | <atom>

<atom> ::= ( <expr> )
         | <id>
         | <int>
         | <flt>
```

‹id›, ‹int› e ‹flt› são *tokens* definidos na gramática léxica

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Notação EBNF

- A notação EBNF estende a BFN, incluindo
  - Operador de opcional
  - Operador de repetição
  - Operador de escolha múltipla

- A EBNF aumenta a legibilidade, mas ***não*** o poder de representação da BNF
  - Qualquer gramática que pode ser escrita com BNF pode ser escrita com EBNF e vice-versa

# EBNF: opcional

- Um trecho da regra escrito entre colchetes equivale a uma derivação opcional
  - Use quando duas regras derivarem formas sentenciais muito parecidas

```
<if_stmt> ::= if <expr_lógica> then <stmt>
            | if <expr_lógica> then <stmt>
                 else <stmt>
```

```
<if_stmt> ::= if <expr_lógica> then <stmt>
            [ else <stmt> ]
```

# EBNF: repetições

- Um trecho entre colchetes é equivalente ao operador de fecho
  - Use para substituir recursões

```
<lista_var> ::= <var>
              | <var>, <lista_var>
```

```
<lista_var> ::= <var> {, <lista_var>}
```

# EBNF: múltipla escolha

- Use quando uma regra gera várias formas sentenciais semelhantes, que diferem por um elemento que deve ser escolhido

```
<mult> ::= <mult> * <soma>
         | <mult> / <soma>
         | <mult> % <soma>
```

```
<mult> ::= <mult> ( * | / | % ) <soma>
```

# Gramática de expressões BNF vs. EBNF

```
<expr> ::= <expr> + <term>
         | <expr> - <term>
         | <term>

<term> ::= <term> * <fator>
         | <term> / <fator>
         | <fator>

<fator> ::= <atom> ** <fator>
          | <atom>

<atom> ::= ( <expr> )
         | <id>
```

A repetição substitui a recursão.

```
<expr> ::= <term> {(+ | -) <term>}

<expr> ::= <fator> {(* | /) <term>}

<fator> ::= <exp> {** <exp>}

<exp> ::= ( <expr> )
        | <id>
```

# EBNF e Associatividade

- Note que, sem recursão, não é possível representar associatividade
  - As duas regras
    - <term> ::= <fator> {* <fator>}
    - <term> ::= {<fator> *} <fator>
  - representam a mesma coisa
  - Não há recursão à esquerda ou à direita com o mecanismo de repetiação da EBNF
    - É uma iteração!

# EBNF e Associatividade

- Normalmente, a notação EBNF é utilizada para descrever as formas permitidas da linguagem
- Associatividade e precedência de operadores são listadas em uma tabela
  - Tabela de precedência
  - Normalmente contida nos documentos que descrevem as linguagens

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | ++ --<br>()<br>[]<br>.<br>-><br>(*type*){*list*} | Suffix/postfix increment and decrement<br>Function call<br>Array subscripting<br>Structure and union member access<br>Structure and union member access through pointer<br>Compound literal(C99) | Left-to-right |
| 2 | ++ --<br>+ -<br>! ~<br>(*type*)<br>*<br>&<br>sizeof<br>_Alignof | Prefix increment and decrement<br>Unary plus and minus<br>Logical NOT and bitwise NOT<br>Type cast<br>Indirection (dereference)<br>Address-of<br>Size-of[note 1]<br>Alignment requirement(C11) | Right-to-left |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right |
| 4 | + - | Addition and subtraction | |
| 5 | << >> | Bitwise left shift and right shift | |
| 6 | < <=<br>> >= | For relational operators < and ≤ respectively<br>For relational operators > and ≥ respectively | |
| 7 | == != | For relational = and ≠ respectively | |
| 8 | & | Bitwise AND | |
| 9 | ^ | Bitwise XOR (exclusive or) | |
| 10 | \| | Bitwise OR (inclusive or) | |
| 11 | && | Logical AND | |
| 12 | \|\| | Logical OR | |
| 13[note 2] | ?: | Ternary conditional[note 3] | Right-to-Left |
| 14 | =<br>+= -=<br>*= /= %=<br><<= >>=<br>&= ^= \|= | Simple assignment<br>Assignment by sum and difference<br>Assignment by product, quotient, and remainder<br>Assignment by bitwise left shift and right shift<br>Assignment by bitwise AND, XOR, and OR | |
| 15 | , | Comma | Left-to-right |

Exemplo: tabela de precedência e associatividade de C e C++

Fonte e tabela completa, incluindo notas de rodapé:

https://en.cppreference.com/w/c/language/operator_precedence

# Descrições de linguagens na prática

- Linguagens normalmente são descritas em notação EBNF modificada
  - Por exemplo, o padrão da linguagem JavaScript ("formalmente" ECMAScript) usa
    - *Itálicos* para representar símbolos não terminais e **negrito** para terminais
    - Recursão explícita no lugar de repetições
    - O sufixo opt para designar opcionais e o prefixo one of para designar escolhas
    - Regras múltiplas em várias linhas sem explicitar o operador |

# Descrições de linguagens na prática

- Amostra do léxico de JavaScript:

*DecimalIntegerLiteral* ::
    **0**
    *NonZeroDigit* *DecimalDigits*$_{opt}$

*DecimalDigits* ::
    *DecimalDigit*
    *DecimalDigits* *DecimalDigit*

*DecimalDigit* :: one of
    **0 1 2 3 4 5 6 7 8 9**

*NonZeroDigit* :: one of
    **1 2 3 4 5 6 7 8 9**

Gramática completa: https://tc39.es/ecma262/#sec-ecmascript-language-lexical-grammar

# Descrições de linguagens na prática

- O padrão da linguagem Go utiliza a notação EBNF de forma bastante "relaxada[1]"
  - Não é uma gramática
  - As regras especificam o formato das notações
  - As regras de precedência e de associatividade **não** aparecem nas regras, mas em uma tabela
  - Além disso, o documento mescla a descrição da sintaxe com a semântica

[1] O termo "relaxado" **não** é empregado aqui com sentido pejorativo.

# Descrições de linguagens na prática

- Amostra da sintaxe de expressões de Go:

```
Expression = UnaryExpr | Expression   binary_op   Expression.
UnaryExpr  = PrimaryExpr | unary_op  UnaryExpr.

binary_op  = "||" | "&&" | rel_op | add_op | mul_op.
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" | "|" | "^" .
mul_op     = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .

unary_op   = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

Especificação completa: https://golang.org/ref/spec

# Agenda

- Introdução
- Linguagens e expressões regulares
- Gramáticas
- Notação BNF
- Árvores de derivação e ambiguidade
- Gramáticas de expressões
- Notação EBNF
- Limitações da análise sintática

# Limitações

- Quais aspectos de linguagens de programação **não** podemos representar com BNF ou EBNF?

```
int main()
{
    int i = 0;
    i = i + j;
}
```

Variável não declarada

É preciso conhecer o contexto das variáveis utilizadas na expressão para garantir que elas foram declaradas

BNF representa apenas gramáticas livres de contexto

# Limitações

- Quais aspectos de linguagens de programação são **difíceis** de representar com BNF ou EBNF?

```
float f = 0;
  int i = 0.0;
```

Conversões de tipos de dados

Em Java:

- Inteiro para ponto flutuante ok
- Ponto flutuante para inteiro inválido

# Limitações

- É *possível* representar conversões de tipos através de BNF ou EBNF
  - Mas seria necessário incluir *muitas* regras

```
<assign_int>   ::= <var_int> = <expr_int>
<assign_float> ::= <var_float> = <expr_float_or_int>

<expr_int>     ::= ...

<expr_float_or_int> ::= <expr_int> | <exp_float>
<expr_float>        ::= ...
```

# Semântica estática

- Alguns aspectos que parecem mais pertinentes à sintaxe da linguagens são deixados para a semântica
- Semântica estática
  - Diz respeito às formas válidas do programa...
  - Mas está apenas indiretamente relacionada ao significado dos programas
  - O objetivo é verificar ***se*** uma forma sintática pode ter um significado válido, em vez ***qual é*** o significado do programa