Linguagens Formais e Autômatos

Gramáticas Livre do Contexto (GLC)

Cristiano Lehrer, M.Sc.

ybadoo

Linguagens Livres do Contexto (1/2)

- Seu estudo é de fundamental importância na informática:
  - Compreende um universo mais amplo de linguagens (comparativamente com as regulares) tratando, adequadamente, questões como parênteses balanceados, construções de blocos estruturados, entre outras, típicas de linguagens de programação como C, Java, entre outras.
  - Os algoritmos reconhecedores e geradores que implementam as Linguagens Livres do Contexto são relativamente simples e possuem uma boa eficiência.
  - Exemplos típicos de aplicações dos conceitos e resultados referentes às Linguagens Livre do Contexto são analisadores sintáticos, tradutores de linguagens e processadores de texto em geral.

Linguagens Livres do Contexto (2/2)

- O estudo da Classe das Linguagens Livres do Contexto é desenvolvido a partir de um formalismo axiomático ou gerador (gramática) e um operacional ou reconhecedor (autômato) como segue:
  - Gramática Livre do Contexto:
    - Gramática onde as regras de produção são definidas de forma mais livre que na Gramática Regular.
  - Autômato com Pilha:
    - Autômato cuja estrutura é análoga à do Autômato Finito, adicionando uma memória auxiliar do tipo pilha (a qual pode ser lida ou gravada) e a facilidade de não determinismo.

Gramática Livre do Contexto

- Uma Gramática Livre do Contexto (G) é uma gramática:

$$G = (V, T, P, S)$$

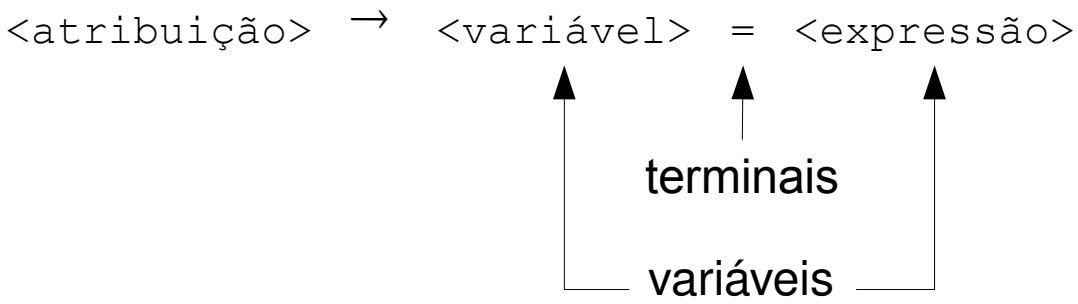
  - V → conjunto de variáveis.
  - T → conjunto de terminais.
  - P → regras de produção.
  - S → variável inicial.

Forma de Backus-Naur (1/6)

- Popularmente conhecida como BNF.
- É um método formal para descrição das regras de produção de uma Gramática Livre do Contexto.
- Uma metalinguagem é uma linguagem usada para descrever outra linguagem.
- BNF é uma metalinguagem para linguagens de programação.

Forma de Backus-Naur (2/6)

- BNF utiliza abstrações para representar estruturas sintáticas.
- As abstrações são geralmente chamadas de símbolos não terminais (variáveis).
- Os lexemas e *tokens* são chamados símbolos terminais.
- As definições são chamadas de regras:



ybadoo

Forma de Backus-Naur (3/6)

- Uma gramática é uma coleção de regras.
- Símbolos não terminais podem ter mais de um definição:

```
<if_stmt>   →   if <logic_expr> then <stmt>
            |   if <logic_expr> then <stmt> else <stmt>
```

- Listas de tamanhos variáveis:

```
<ident_list>   →   identifier
               |   identifier, <ident_list>
```

Forma de Backus-Naur (4/6)

- BNF é um dispositivo de geração de linguagens.
- Sentenças são geradas através da aplicação de regras.
- Inicia com um não terminal especial, chamado de Símbolo Inicial.
- Esta geração de sentença é chamada de derivação.
- O Símbolo Inicial representa um programa completo.

Forma de Backus-Naur (5/6)

G = ({program, stm_list, stmt, var, expression},
{begin, end, ;, =, +, -, A, B, C}, P, program)

P = { <program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>

| <stmt> ; <stmt_list>

<stmt> → <var> **=** <expression>

<var> → **A** | **B** | **C**

<expression> → <var> **+** <var>

| <var> **-** <var>

| <var> }

ybadoo

Forma de Backus-Naur (6/6)

- Uma derivação de um programa nesta linguagem:

```
<program>
begin <stmt_list> end
begin <stmt> ; <stmt_list> end
begin <var> = <expression> ; <stmt_list> end
begin A = <expression> ; <stmt_list> end
begin A = <var> + <var> ; <stmt_list> end
begin A = B + <var> ; <stmt_list> end
begin A = B + C ; <stmt_list> end
begin A = B + C ; <stmt> end
begin A = B + C ; <var> = <expression> end
begin A = B + C ; B = <expression> end
begin A = B + C ; B = <var> end
begin A = B + C ; B = C end
```


ybadoo

Derivação (1/2)

- Um outro exemplo de gramática:

```
G = ({assign, id, expr},
     {A, B, C, =, +, *, (, )}, P, assign)
P = { <assign> →  <id> = <expr>
      <id>     →  A   |   B   |   C
      <expr>   →  <id> + <expr>
               |  <id> * <expr>
               |  ( <expr> )
               |  <id>   }
```

Derivação (2/2)

- Derivação à Extrema Esquerda (DEE):

<assign>
<id> **=** <expr>
A = <expr>
A = <id> ***** <expr>
A = B * <expr>
A = B * (<expr> **)**
A = B * (<id> **+** <expr> **)**
A = B * (A + <expr> **)**
A = B * (A + <id> **)**
A = B * (A + C)

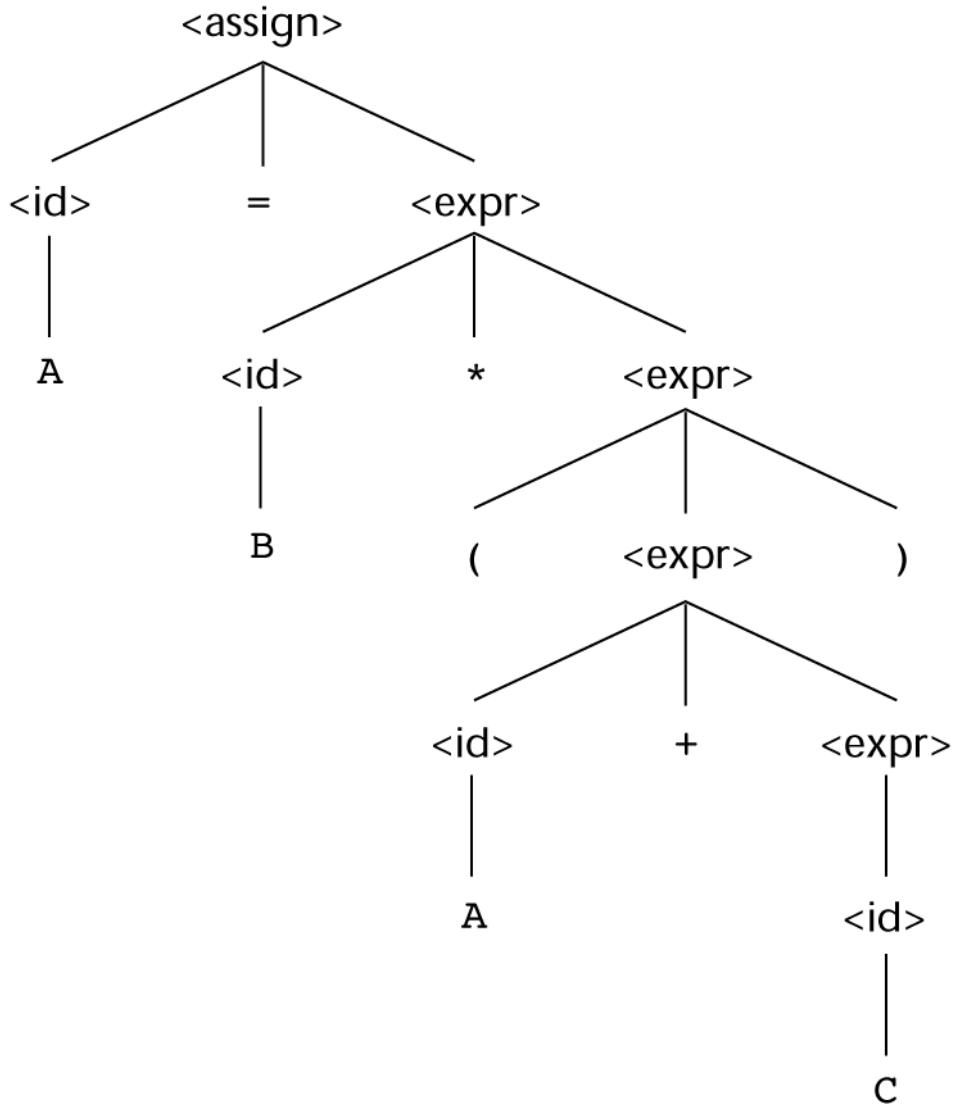
- Derivação à Extrema Direita (DED):

<assign>
<id> **=** <expr>
<id> **=** <id> ***** <expr>
<id> **=** <id> *** (** <expr> **)**
<id> **=** <id> *** (** <id> **+** <expr> **)**
<id> **=** <id> *** (** <id> **+** <id> **)**
<id> **=** <id> *** (** <id> **+ C)**
<id> **=** <id> *** (A + C)**
<id> **= B * (A + C)**
A = B * (A + C)

G = ({assign, id, expr},
{A, B, C, =, +, *, (,)}, P, assign)
P = { <assign> → <id> **=** <expr>
<id> → **A** | **B** | **C**
<expr> → <id> **+** <expr>
| <id> ***** <expr>
| **(** <expr> **)**
| <id> }

ybadoo

Árvore de Derivação



- Gramáticas descrevem uma estrutura hierárquica das sentenças.
- Essa estrutura hierárquica é chamada de *parse tree*.

ybadoo

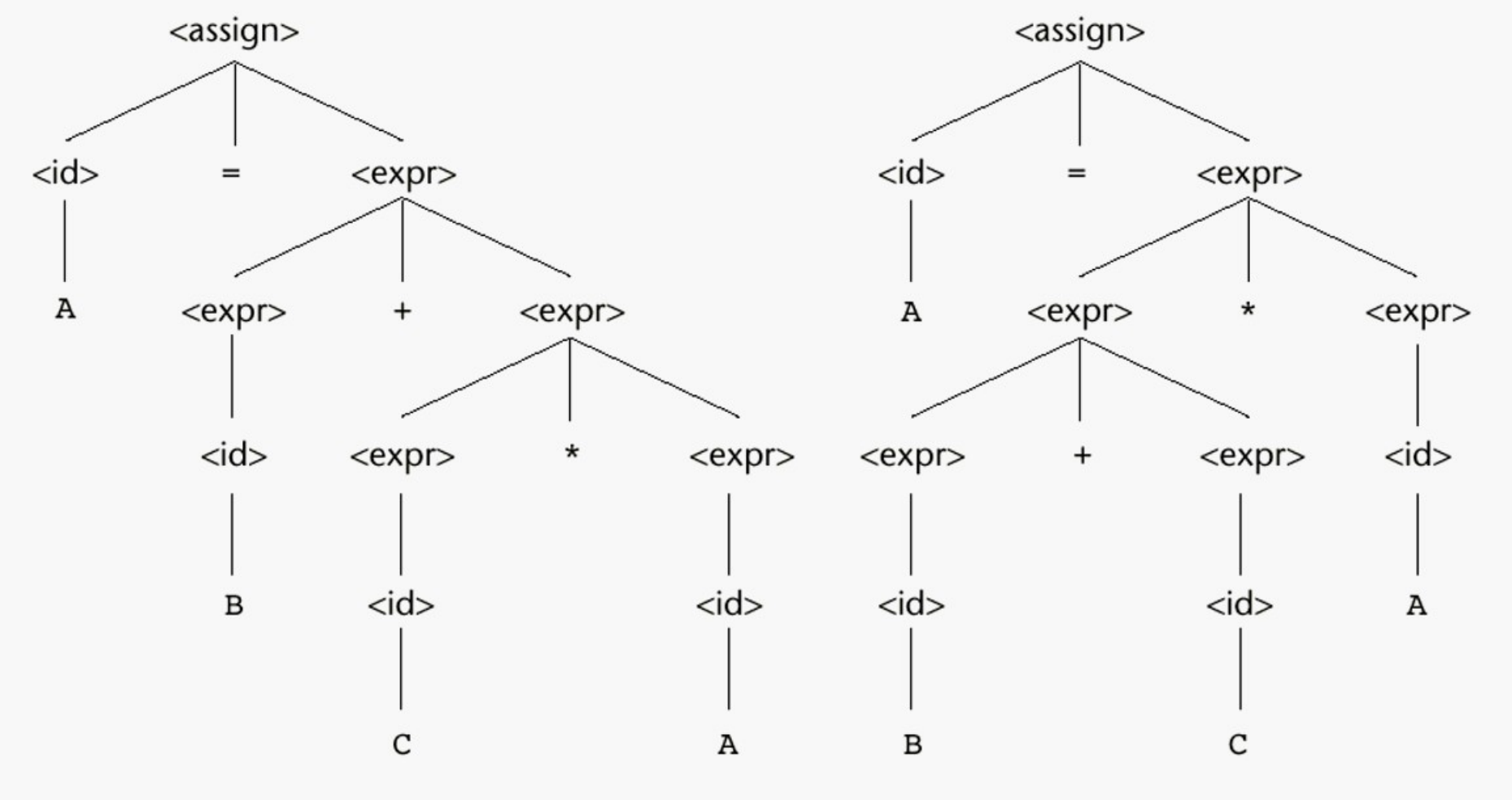
Ambiguidade (1/2)

- Uma gramática que gera uma sentença para o qual existem duas ou mais árvores de derivação.

```
G = ({assign, id, expr},
     {A, B, C, =, +, *, (, )}, P, assign)
P = { <assign>    →  <id> = <expr>
      <id>        →  A   |   B   |   C
      <expr>      →  <expr> + <expr>
                  |  <expr> * <expr>
                  |  ( <expr> )
                  |  <id>   }
```

Ambiguidade (2/2)

```
A := B + C * A
```

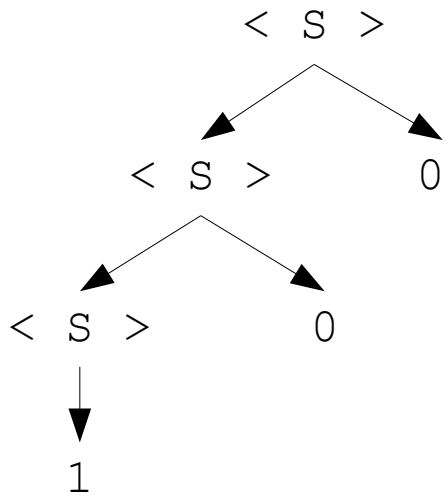


Recursividade (1/2)

Desenvolva uma Gramática Livre do Contexto sobre o alfabeto Σ = {0, 1}, que reconheça a linguagem L = {w | w é um número binário}.

Recursividade à esquerda

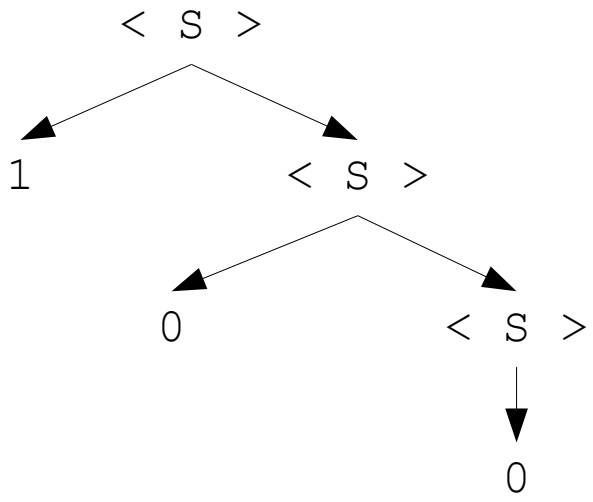
```
G = ({S}, {0, 1}, P , S)
P = {< S > -> < S > 0
            |  < S > 1
            |  0
            |  1 }
```



100

Recursividade à direita

```
G = ({S}, {0, 1}, P , S)
P = {< S > -> 0 < S >
            |  1 < S >
            |  0
            |  1 }
```



Recursividade (2/2)

Desenvolva uma Gramática Livre do Contexto sobre o alfabeto Σ = {0, 1}, que reconheça a linguagem L = {w | w é um número binário}.

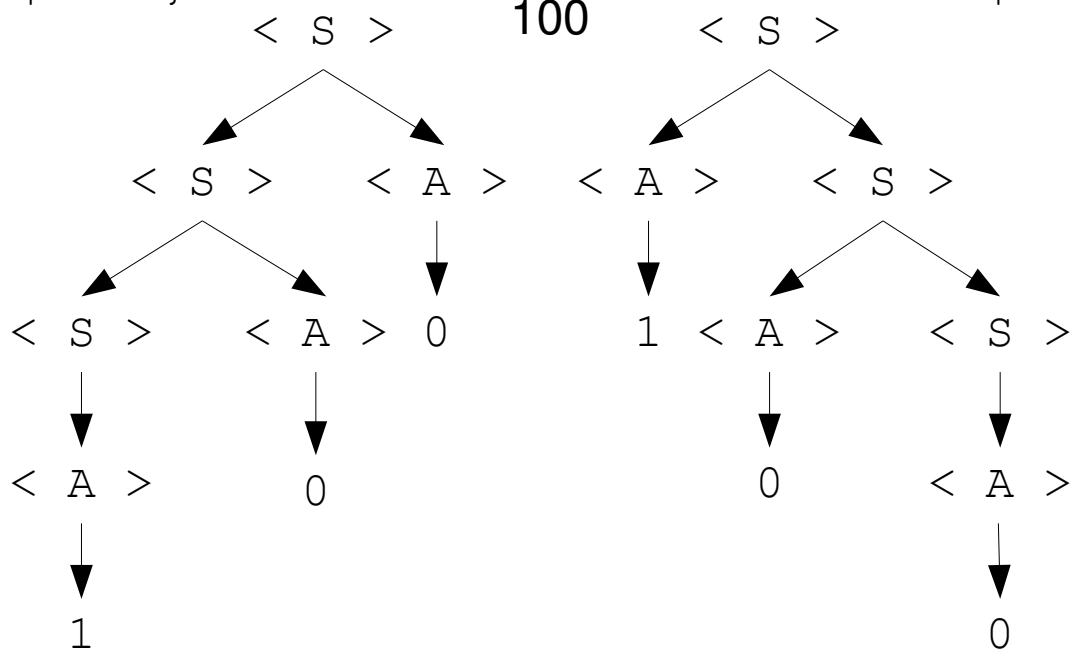
Recursividade à esquerda

```
G = ({S, A}, {0, 1}, P , S)
P = {< S > -> < S > < A >
           |  < A >
     < A > -> 0
           |  1 }
```

Recursividade à direita

```
G = ({S, A}, {0, 1}, P , S)
P = {< S > -> < A > < S >
           |  < A >
     < A > -> 0
           |  1 }
```

100



Linguagem Regular (1/2)

Desenvolva uma Gramática Livre do Contexto sobre o alfabeto Σ = {x, y, z, w}, que reconheça a linguagem L = {w | w possui *xyw* ou *wzz* como prefixo, *wzx* ou *wyx* como subpalavra e *xyy* ou *xwz* como sufixo}.

Com recursividade à esquerda

```
G = ({exp, pre, presub, sub, subsuf, suf, alf}, {x, y, z, w}, P, exp)
P = {< exp >     ->  < pre > < alf > < sub > < alf > < suf >
                 |   < presub > < alf > < suf >
                 |   < pre > < alf > < subsuf >
                 |   xywzxyy | xywzxwz | xywyxyy | xywyxwz
     < pre >     ->  xyw | wzz
     < presub >  ->  xywzx | xywyx
     < sub >     ->  wzx | wyx
     < subsuf >  ->  wzxyy | wzxwz | wyxyy | wyxwz
     < suf >     ->  xyy | xwz
     < alf >     ->  < alf > x | < alf > y | < alf > z | < alf > w | ε  }
```

Linguagem Regular (2/2)

Desenvolva uma Gramática Livre do Contexto sobre o alfabeto Σ = {x, y, z, w}, que reconheça a linguagem L = {w | w possui *xyw* ou *wzz* como prefixo, *wzx* ou *wyx* como subpalavra e *xyy* ou *xwz* como sufixo}.

Com recursividade à direita

```
G = ({exp, pre, presub, sub, subsuf, suf, alf}, {x, y, z, w}, P, exp)
P = {< exp >     ->  < pre > < alf > < sub > < alf > < suf >
                 |   < presub > < alf > < suf >
                 |   < pre > < alf > < subsuf >
                 |   xywzxyy | xywzxwz | xywyxyy | xywyxwz
     < pre >     ->  xyw | wzz
     < presub >  ->  xywzx | xywyx
     < sub >     ->  wzx | wyx
     < subsuf >  ->  wzxyy | wzxwz | wyxyy | wyxwz
     < suf >     ->  xyy | xwz
     < alf >     ->  x < alf > | y < alf > | z < alf > | w < alf > | ε  }
```


ybadoo

Precedência de Operadores

Operadores: multiplicação com maior precedência do que a adição

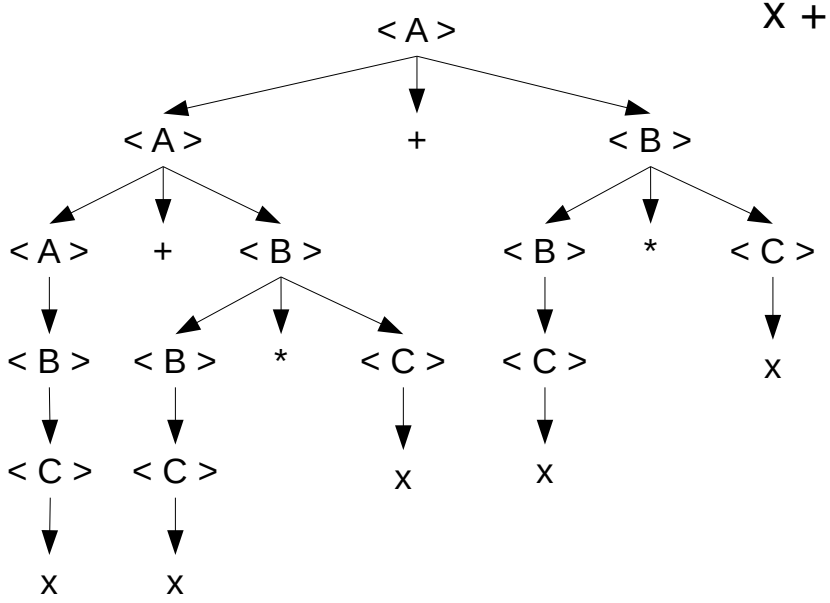
Recursividade à esquerda

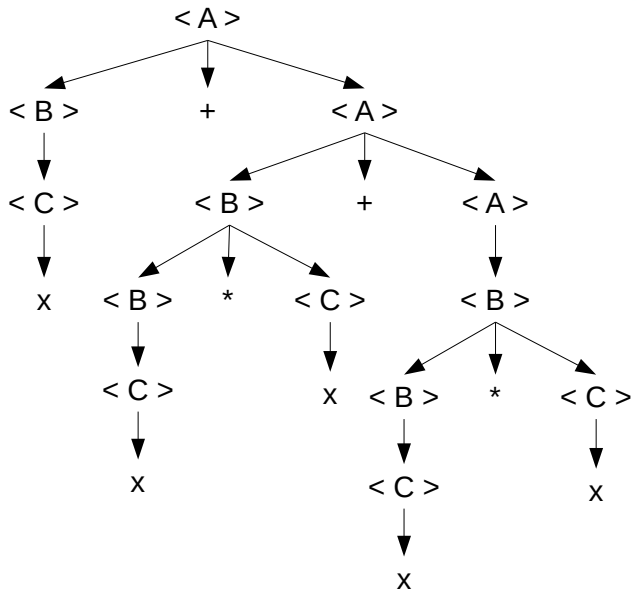
```
G = ({A, B, C}, {x, +, *}, P, A)
P = {< A > -> < A > + < B >
           |  < B >
     < B > -> < B > * < C >
           |  < C >
     < C > -> x }
```

Recursividade à direita

```
G = ({A, B, C}, {x, +, *}, P, A)
P = {< A > -> < B > + < A >
           |  < B >
     < B > -> < C > * < B >
           |  < C >
     < C > -> x }
```

x + x * x + x * x





Exemplos (1/2)

Reconhecimento de números inteiros e reais

```
G = ({REAL, INTEIRO, SINAL, NUMERO, DIGITO}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .},
      P, REAL)
P = {< REAL >        ->  < SINAL > . < NUMERO >
                     |   < INTEIRO > . < NUMERO >
                     |   < INTEIRO >
     < INTEIRO >     ->  < SINAL > < NUMERO >
     < SINAL >       ->  + | - | ε
     < NUMERO >      ->  < NUMERO > < DIGITO >
                     |   < DIGITO >
     < DIGITO >      ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  }
```

Exemplos (2/2)

Nome de variáveis na linguagem de programação Java

```
G = ({NOME, TEXTO, LETRA, SIMBOLO, DIGITO}, {a, b, c, d, e, f, g, h, i, j, k, l, m,
      n, o, p, q, r, s, t, u, v, w, x, y, z, $, _, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9},
      P, NOME)
P = {< NOME >      ->  < LETRA > < TEXTO >
     < TEXTO >     ->  < TEXTO > < LETRA >
                   |   < TEXTO > < SIMBOLO >
                   |   < TEXTO > < DIGITO >
                   |   ε
     < LETRA >     ->  a | b | c | d | e | f | g | h | i | j | k | l | m | n | o
                   |   p | q | r | s | t | u | v | w | x | y | z
     < SIMBOLO >   ->  $ | _
     < DIGITO >    ->  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  }
```