unesp

Linguagens de Programação

Aula 4

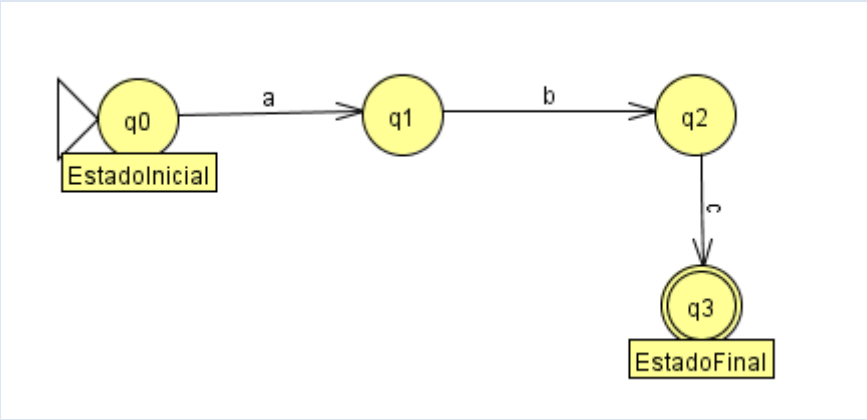
Celso Olivete Júnior

olivete@fct.unesp.br

Na aula passada...
Autômatos finitos

- **AF:** exemplos...

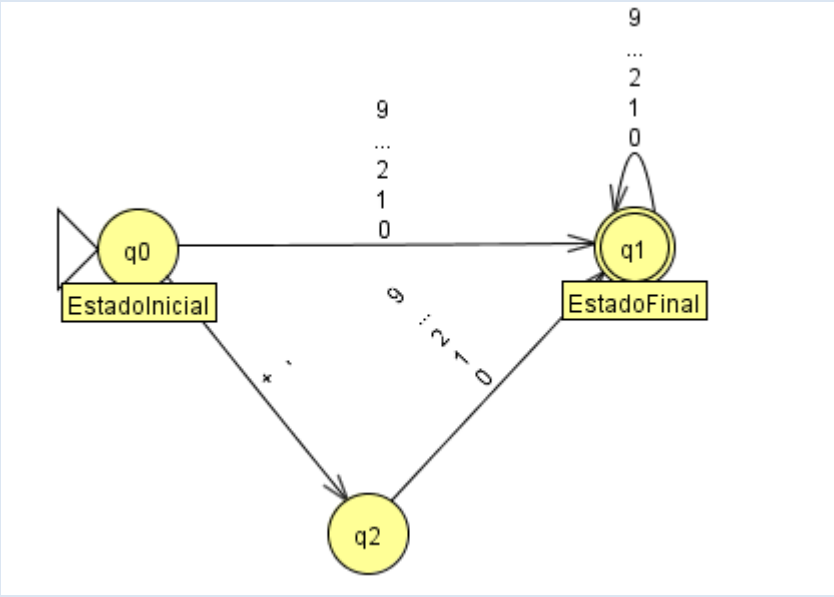
Cadeia de caracteres a,b,c



Na aula passada...
Autômatos finitos

- **AF:** exemplos...

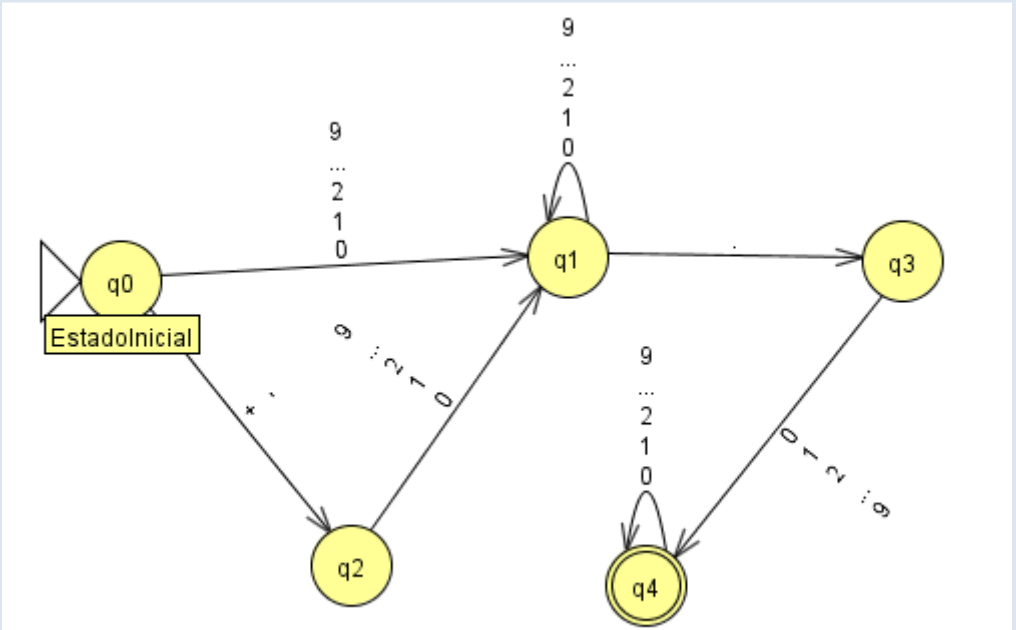
Números inteiros (com ou sem sinal)



Na aula passada...
Autômatos finitos

- **AF:** exemplos...

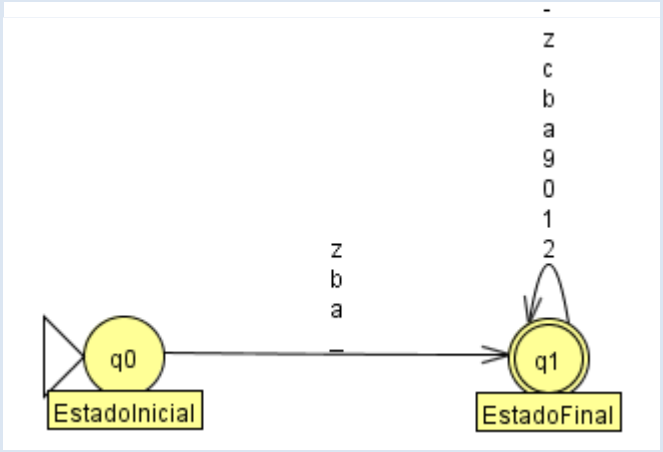
Números reais (com ou sem sinais)



Na aula passada...
Autômatos finitos

- **AF:** exemplos...

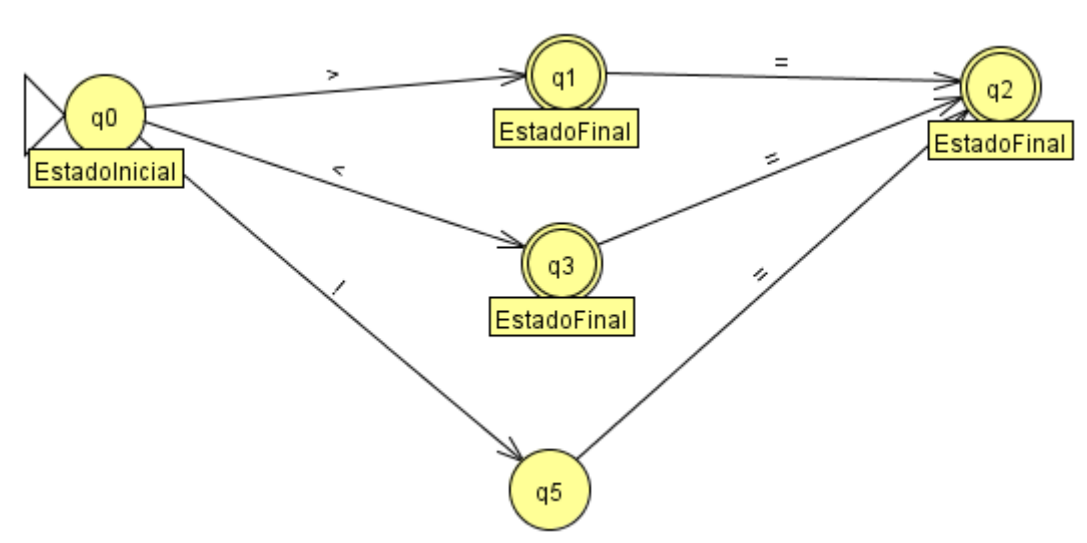
Identificador



Na aula passada...
Autômatos finitos

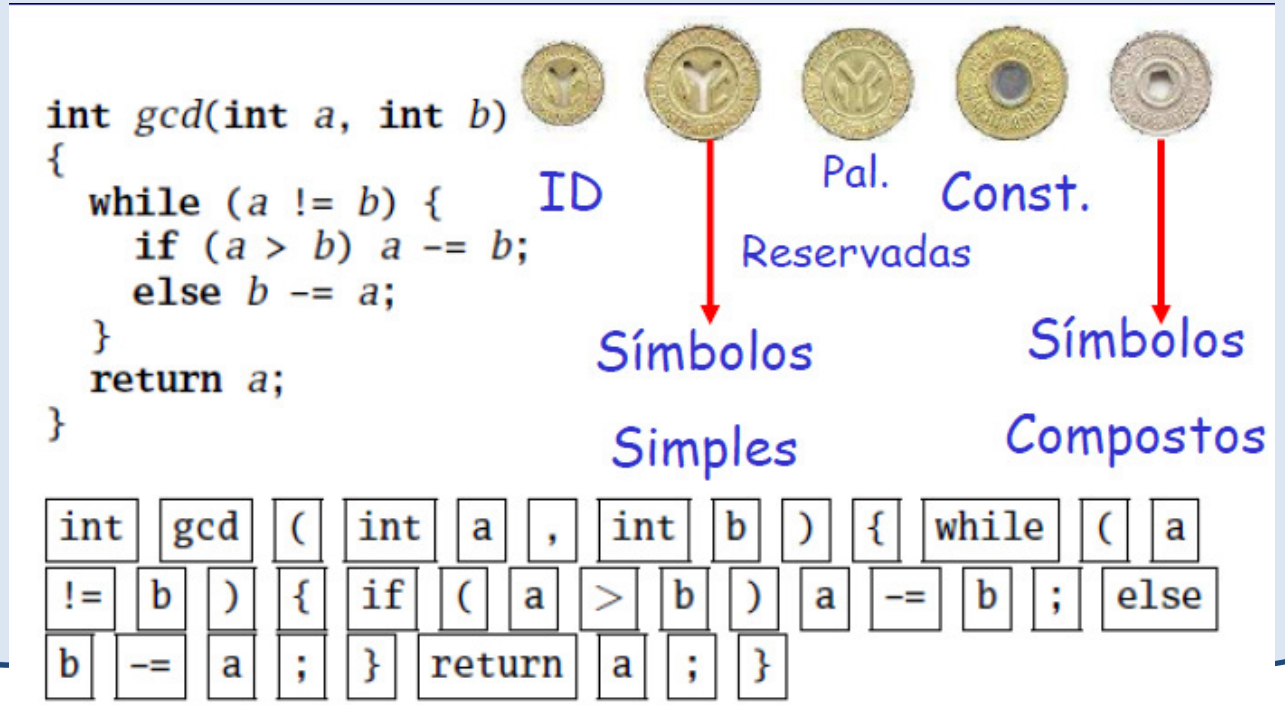
- **AF:** exemplos...

Comparação (>, >=, <, <=, !=)



Compilação passo-a-passo
Análise léxica

- **Análise léxica**: A análise léxica (AL) é responsável por ler o código fonte e separá-lo em partes significativas, denominadas *tokens,* instanciadas por átomos



Compilação passo-a-passo
Análise léxica

- **Análise léxica**: A análise léxica (AL) é responsável por ler o código fonte e separá-lo em partes significativas, denominadas *tokens,* instanciadas por átomos
  - *Esse reconhecido pode ser feito por autômatos finitos*

Compilação passo-a-passo
Análise léxica

- Classes de átomos mais comuns:
  - identificadores;
  - palavras reservadas;
  - números inteiros sem sinal;
  - números reais;
  - cadeias de caracteres;
  - sinais de pontuação e de operação;
  - caracteres especiais;
  - símbolos compostos de dois ou mais caracteres especiais;
  - comentários;
  - etc.

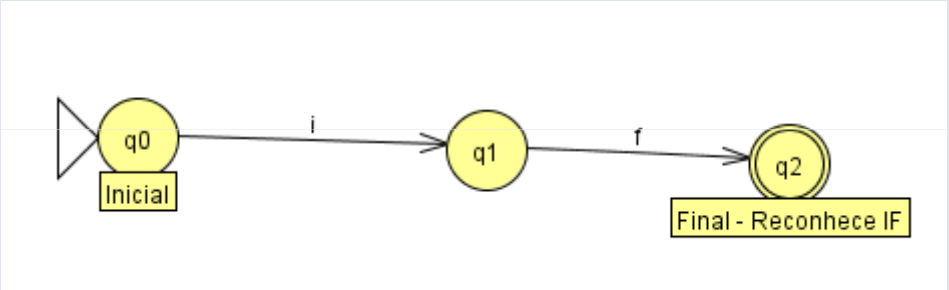
Compilação passo-a-passo
Análise léxica

- Exemplos de *tokens* que podem ser reconhecidos em uma linguagem de programação como C

palavras reservadas	if else while do
identificadores	
operadores relacionais	< > <= >= == !=
operadores aritméticos	+ * / -
operadores lógicos	&& || & | !
operador de atribuição	=
delimitadores	; ,
caracteres especiais	( ) [ ] { }

Compilação passo-a-passo
Análise léxica

- Reconhecimento da palavra reservada IF



- Façam os demais AF para reconhecimento dos *tokens*

palavras reservadas	if else while do
identificadores	
operadores relacionais	< > <= >= == !=
operadores aritméticos	+ * / -
operadores lógicos	&& || & | !
operador de atribuição	=
delimitadores	; ,
caracteres especiais	( ) [ ] { }

Compilação passo-a-passo

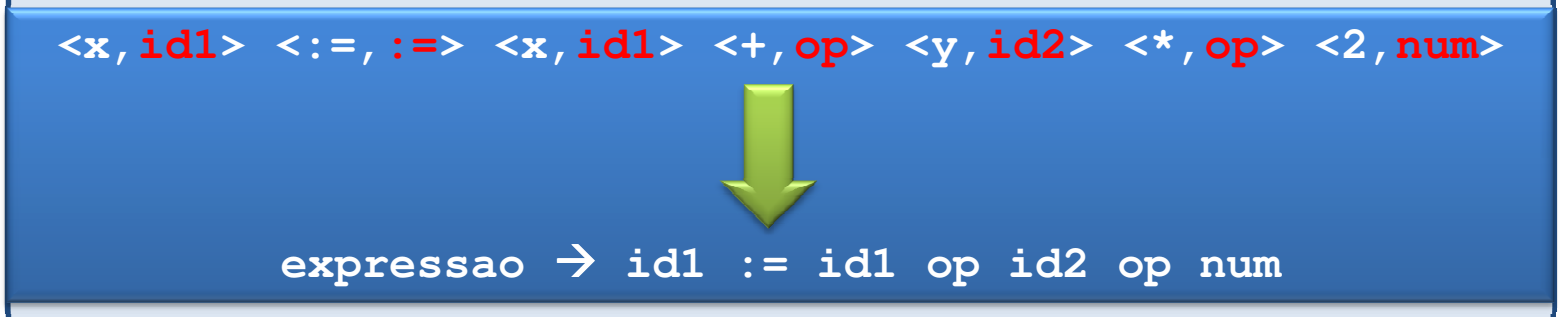
- Retomando o processo de compilação: Análise léxica

```
x:=x+y*2
```

```
<x,id1> <:=,:=> <x,id1> <+,op> <y,id2> <*,op> <2,num>
```

Compilação passo-a-passo

- **Analisador sintático**: Verificação da formação do programa
  - **Gramáticas livres de contexto**



x:=x+y*2

Compilação passo-a-passo

- **Analisador semântico**: verificação do uso adequado
  - A partir da **gramática**, verifica se os identificadores estão sendo usados de acordo com o tipo declarado

Compilação passo-a-passo

- Fases de **síntese**:
  - **Geração de código intermediário (3 endereços)**

```
id1 := id1 op id2 op num
```

```
temp1 := id2 * 2
temp2 := id1 + temp1
id1 := temp2
```

x:=x+y*2

Compilação passo-a-passo

- Fases de **síntese**:
  - **Otimização do código intermediário**

```
temp1 := id2 * 2
temp2 := id1 + temp1
id1 := temp2
```

```
temp1 := id2 * 2
id1 := id1 + temp1
```

x:=x+y*2

Compilação passo-a-passo

- Fases de **síntese**:
  - **Geração do código objeto**

```
temp1 := id2 * 2
id1 := id1 + temp1
```

```
MOV id2 R1
MULT 2 R1
MOV id1 R2
ADD R1 R2
MOV R2 id1
```

x:=x+y*2

Objetivo – Especificação de uma LP

- Foco na **forma de descrever formalmente uma LP**
  - Considerando os dois aspectos (da fase de **análise**)
    - **Sintaxe**: conjunto de regras que determinam quais construções são corretas
    - **Semântica**: descrição de como as construções da linguagem devem ser interpretadas e executadas

❖ Em Pascal: a:=b
Sintaxe: comando de atribuição correto
Semântica: substituir o valor de a pelo valor de b

Sintaxe

- A sintaxe de uma LP é descrita formalmente por uma **gramática**

Conceito de linguagem

- **Linguagem** é uma coleção de **cadeias** de símbolos, de comprimento finito. Estas cadeias são denominadas sentenças da linguagem, e são **formadas pela justaposição de elementos individuais, os símbolos ou átomos da linguagem**.

Linguagem

- Gramática: conceitos preliminares
  - **Alfabeto ou vocabulário:** Conjunto finito não vazio de símbolos. Símbolo é um elemento qualquer de um alfabeto.
    - Ex:
      - ✓{a,b}
      - ✓{0,1,2,3,4,5,6,7,8,9}

Linguagem

- Gramática: conceitos preliminares
  - **Cadeia:** Concatenação de símbolos de um alfabeto. Define-se como cadeia vazia ou nula uma cadeia que não contém nenhum símbolo.
  - Ex:
    - aab
    - 123094
    - λ – cadeia nula

Linguagem

- Gramática: conceitos preliminares
  - **Comprimento de cadeia:** Número de símbolos de uma cadeia. Ex:
    - |aab| = 3
    - |123094|=6
    - |λ|=0

Linguagem

- Gramática: conceitos preliminares
  - **Concatenação de cadeias:** Define-se a concatenação z de uma cadeia x com uma cadeia y, como sendo a concatenação dos símbolos de ambas as cadeias, formando a cadeia xy. |z| = |x| + |y|. Ex:
    - x = abaa; y = ba → z = abaaba
    - x = ba; y = λ → z = ba

Linguagem

- Gramática: conceitos preliminares
- **Produto de alfabetos:** É o produto cartesiano de alfabetos. Ex:
  - V1 = {a,b} V2 = {1, 2, 3} →
  - V1.V2 = V1xV2 = {a1, a2, a3, b1, b2, b3}
  - **Observe que V1.V2 ≠ V2.V1**

Linguagem

- Gramática: conceitos preliminares
- **Fechamento (Clausura) de um Alfabeto:** Seja A um alfabeto, então o fechamento de A é definido como:

$$A^* = A^0 \cup A^1 \cup A^2 \cup \ldots \cup A^n \cup \ldots$$

- Portanto A* = conjunto das cadeias de qualquer comprimento sobre o alfabeto a. Ex:
  - A = {1}
  - A* = {λ, 1, 11, 111, ...}

Linguagem

- Gramática: conceitos preliminares
- **Fechamento positivo de um Alfabeto:** Seja A um alfabeto, então o fechamento positivo de A é definido como:

$$A^+ = A^1 \cup A^2 \cup \ldots \cup A^n \cup \ldots$$

- Portanto A^+ = conjunto das cadeias de comprimento um ou maior sobre o alfabeto a. Ex:
  - A = {1}
  - A^+ = {1, 11, 111, ...}

Gramática

- Gramática: utilizada para descrever a **sintaxe** de uma gramática. Formada por:

G = (Vn, Vt, P, S)

  - onde:
    - ✓ **Vn representa o vocabulário não terminal (variáveis)** da gramática. Este vocabulário corresponde ao conjunto de todos os símbolos dos quais a gramática se vale para definir as leis de formação das sentenças da linguagem.

Gramática

- Gramática

G = (Vn, Vt, P, S)

- onde:
  - ✓ **Vt** é o **vocabulário terminal**, contendo os símbolos que constituem as sentenças da linguagem. Dá-se o nome de terminais aos elementos de **Vt**.

Gramática

❑ Gramática

G = (Vn, Vt, P, S)

- onde:
  - ✓ **P** são as **regras de produção**, que definem o conjunto de todas as leis de formação utilizadas pela gramática para definir a linguagem. Para tanto, cada construção parcial, representada por um não-terminal, é definida como um conjunto de regras de formação relativas à definição do não-terminal a ela referente. A cada uma destas regras de formação que compõem o conjunto P dá-se o nome de produção da gramática

Gramática

❑ Gramática

G = (Vn, Vt, P, S)

- onde:
  - ✓ **S** é um elemento de Vn, raiz ou símbolo inicial de G

Gramática

- A sintaxe de uma linguagem é descrita por uma gramática com os seguintes elementos
  - Símbolos terminais: cadeias que estão no programa
    - while, do, for, id
  - Símbolos não-terminais: não aparecem no programa
    - <cmd_while>, <programa>
  - Produções: como produzir cadeias que formam o programa
    - <cmd_while> ::= while (<expressão>) <comandos>
  - Símbolo inicial: não-terminal a partir do qual se inicia a produção do programa
    - <programa>

Gramática: exemplo i

```
<program> ::= begin <lista_cmd> end
<lista_cmd> ::= <cmd> | <cmd>; <lista_cmd>
<cmd> ::= <var> = <expr>
<var> ::= A|B|C
<expr> ::= <var> + <var> | <var> * <var> | <var>
```

Qual expressão pode ser reconhecida pela gramática???
Ex: A=B+C; B=C

- Símbolos terminais: cadeias que estão no programa
  - begin, end, A, B, C, +, *
- Símbolos não-terminais: não aparecem no programa
  - <program>, <lista_cmd>, <expr>, <cmd>, <var>
- Produções: como produzir cadeias que formam o programa
  - <program>, <lista_cmd>, <expr>, <cmd>, <var>
- Símbolo inicial: não-terminal a partir do qual se inicia a produção do programa
  - <program>

Métodos formais de descrever a sintaxe

- Sintaxe
  - definida formalmente através de uma gramática.
- Gramática
  - conjunto de definições que especificam uma sequência válida de caracteres.

- Duas classes de gramáticas são úteis na definição formal das gramáticas:
  - Gramáticas livres de contexto;
  - Gramáticas regulares.

Forma de Backus-Naur

- A notação inicialmente criada por John Backus e Noam Chomsky, foi modificada posteriormente por Peter Naur dando origem à forma de Backus-Naur ou BNF.

Forma de Backus-Naur

- Metalinguagem
  - A BNF é uma metalinguagem para descrever as LP
- Abstrações
  - Símbolos Não-terminais
- Lexemas ou Tokens
  - Símbolos Terminais
- Produção
  - É uma definição de uma abstração;
  - O lado esquerdo corresponde à abstração;
  - O lado direito pode ser uma definição ou um conjunto de definições.

BNF - definições

- < > indica um *não-terminal* – termo que precisa ser expandido, por exemplo <variavel>
- Símbolos não cercados por < > são *terminais;*
  - Eles são representativos por si
    - ✓ Exemplo: if, while, (, =
- Os símbolos ::= significam *é definido como*
- Os símbolos cercados por {} indica que o termo pode ser repetido n vezes (inclusive nenhuma)
- O símbolo | significa *or*
  - Usado para separar alternativas,
    - ✓ Exemplo: <sinal> ::= + | -

BNF

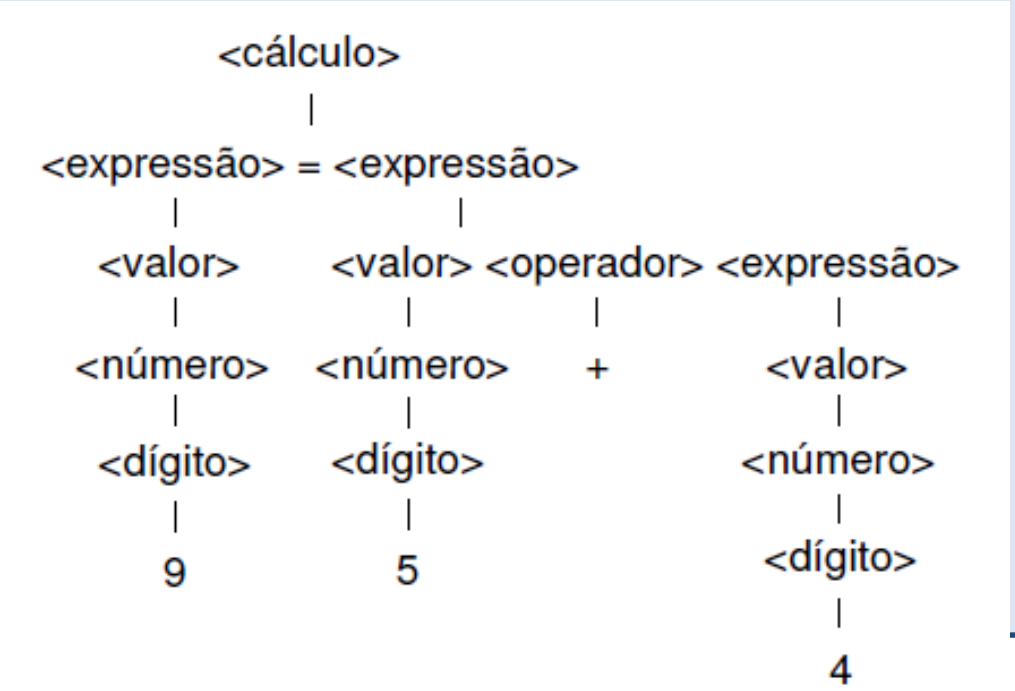
- Exemplo

<cálculo> ::= <expressão> = <expressão>
<expressão> ::= <valor> | <valor><operador><expressão>
<valor> ::= <número> | <sinal><número>
<número> ::= <dígito> | <dígito><número>
<operador> ::= + | - | * | /
<sinal> ::= + | -
<dígito> ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A gramática pode ser usada para produzir ou reconhecer programas sintaticamente corretos
Exemplo: **9=5+4**

<cálculo> ::= <expressão> = <expressão>
<expressão> ::= <valor> | <valor><operador><expressão>
<valor> ::= <número> | <sinal><número>
<número> ::= <dígito> | <dígito><número>
<operador> ::= + | - | * | /
<sinal> ::= + | -
<dígito> ::= 0 |1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Exemplo: **9 = 5+4**



BNF

- BNF é formal e precisa
- BNF é essencial para construção de compiladores

BNF usa recursão

Exemplo: definindo uma BNF para um número inteiro

<integer> ::= <digit> | <integer> <digit>

ou

<integer> ::= <digit> | <digit> <integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Gramática

- Exercícios
  - Faça as BNFs que representam:
  1. letras
  2. Números inteiros (com ou sem sinal)
  3. Números reais (com ou sem sinais)
  4. Identificadores
  5. Comparação (>, >=, <, <=, !=) entre dois identificadores
  6. Atribuição (=) entre dois identificadores
  7. Comando IF → `if(condicao){ comandos;} else { comandos;}`

     `lembre-se de derivar` **`condicao`** `e` **`comandos`**
  8. Comando While

BNF

- BNF para o comando IF

<if statement> ::=if(<condition>) <statement> |

if (<condition>) <statement> else <statement>

BNF - Exemplos

<identifier> ::= <letter> |
<identifier> <letter> |
<identifier> <digit>

<block> ::= { <statement list> }

<statement list> ::= <statement> |
<statement list> <statement>

BNF - Exemplos

```
<statement> ::=
        <block>
    |   <assignment statement>
    |   <break statement>
    |   <continue statement>
    |   <do statement>
    |   <for loop>
    |   <goto statement>
    |   <if statement>
    |   . . .
```

Limitações da BNF

- Não é fácil impor limite de tamanho – por exemplo, o tamanho máximo para nome de variável
- Não há como impor restrição de distribuição no código fonte – por exemplo, uma variável deve ser declarada antes de ser usada
- Descreve apenas a sintaxe, não descreve a semântica
- Nada melhor foi proposto

Aplicando uma BNF

- Uma gramática simplificada para uma atribuição com operações aritméticas – ex: **y = (2*x + 5)*x - 7;**

```
<assignment> ::= ID = <expression> ;
<expression> ::= <expression> + <term>
                 | <expression> - <term>
                 | <term>
<term>   ::=  <term> * <factor>
                 | <term> / <factor>
                 | <factor>
<factor> ::=     ( <expression> )
                 | ID
                 | NUMBER
<ID> ::= x|y
<NUMBER> ::= 0|1|2|…|9
```

O que deve ser produzido?

- O analisador sintático (*parser*) deve identificar uma entrada válida.
- Isso é feito com a definição de mais alto nível (top level production), chamada de símbolo inicial.
  - Geralmente o símbolo inicial é a primeira produção.
- O *parser* tenta "reduzir" a entrada ao símbolo inicial.

O que deve ser produzido?

- O processo de tentar "reduzir" as regras de acordo com a entrada e a partir do **símbolo inicial** é conhecido como derivação.
  - Derivações podem ocorrer mais à esquerda ou mais à direita

Derivação mais à esquerda

- Derivação mais à esquerda: ocorre quando o não-terminal substituído é sempre mais à esquerda.

Exemplo de derivação mais à esquerda

```
<program> ::= begin <lista_cmd> end
<lista_cmd> ::= <cmd> | <cmd>; <lista_cmd>
<cmd> ::= <var> = <expr>
<var> ::= A|B|C
<expr> ::= <var> + <var> | <var> * <var> | <var>
```

- Entrada: A=B
- Derivação

```
<programa> ➔ begin <lista_cmd> end
           ➔ begin <cmd> end
           ➔ begin <var> = <expr> end
           ➔ begin A= <expr> end
           ➔ begin A= <var> end
           ➔ begin A= B end
```

Simule para a entrada: A=B+C; B=C

Derivação mais à direita

- Derivação mais à direita: ocorre quando o não-terminal substituído é sempre mais à direita.

Exemplo de derivação à direita

```
<program> ::= begin <lista_cmd> end
<lista_cmd> ::= <cmd> | <cmd>; <lista_cmd>
<cmd> ::= <var> = <expr>
<var> ::= A|B|C
<expr> ::= <var> + <var> | <var> * <var> | <var>
```

- Entrada: A=B;
- Faça a derivação à direita para a entrada: A=B;

Simule para a entrada: A=B+C; B=C

Exemplo (1)

- Considerando a gramática abaixo, faça as substituições para a entrada **z = (2*x);**

```
<assignment> ::= ID = <expression> ;
<expression> ::= <expression> + <term>
                | <expression> - <term>
                | <term>
<term>    ::=  <term> * <factor>
                | <term> / <factor>
                | <factor>
<factor> ::=      ( <expression> )
                | ID
                | NUMBER
<ID> ::= x|y
<NUMBER> ::= 0|1|2|…|9
```

Exemplo (2)

- Considerando a gramática anterior, faça as substituições para a entrada

Entrada: **z = (2*x + 5);**

1. Realize o processo de derivação mais à esquerda e depois mais à direita
2. Verifique se existe mais de uma derivação possível (à direita ou à esquerda)

Aplicando as regras da gramática (1)

Entrada: **z = (2*x + 5)*y – 7;**

Fonte: **z = (2*x + 5)*y – 7;**

Scanner

```
tokens: ID ASSIGNOP GROUP NUMBER OP ID  OP NUMBER GROUP OP ID OP NUMBER DELIM
valores: z   =       (     2    *   x  +   5       )   *   y  –   7     ;
```

Parser

Aplicando as regras da gramática (2)

```
tokens: ID = ( NUMBER * ID + NUMBER ) * ID - NUMBER ;
```

```
parser:   ID              ...   read (shift) first token
          factor          ...                          reduce
          factor =        ...                          shift
FAIL: Can't match any rules (reduce)
Backtrack and try again
          ID = ( NUMBER ...                            shift
          ID = ( factor ...                            reduce
          ID = ( term * ...                            sh/reduce
          ID = ( term * ID ...                         shift
          ID = ( term * factor ...                     reduce
          ID = ( term ...                              reduce
          ID = ( term + ...                            shift
          ID = ( expression + NUMBER ...               reduce/sh
          ID = ( expression + factor ...               reduce
          ID = ( expression + term ...                 reduce
```

Ação

Aplicando as regras da gramática (3)

tokens: ID = (NUMBER * ID + NUMBER) * ID –NUMBER;

input:	ID = (*expression* ...	reduce
	ID = (*expression*) ...	shift
	ID = *factor* ...	reduce
	ID = *factor* * ...	shift
	ID = *term* * ID ...	reduce/sh
	ID = *term* * *factor* ...	reduce
	ID = *term* ...	reduce
	ID = *term* – ...	shift
	ID = *expression* – ...	reduce
	ID = *expression* – NUMBER ...	shift
	ID = *expression* – *factor* ...	reduce
	ID = *expression* – *term* ...	reduce
	ID = expression ;	shift
	assignment	reduce

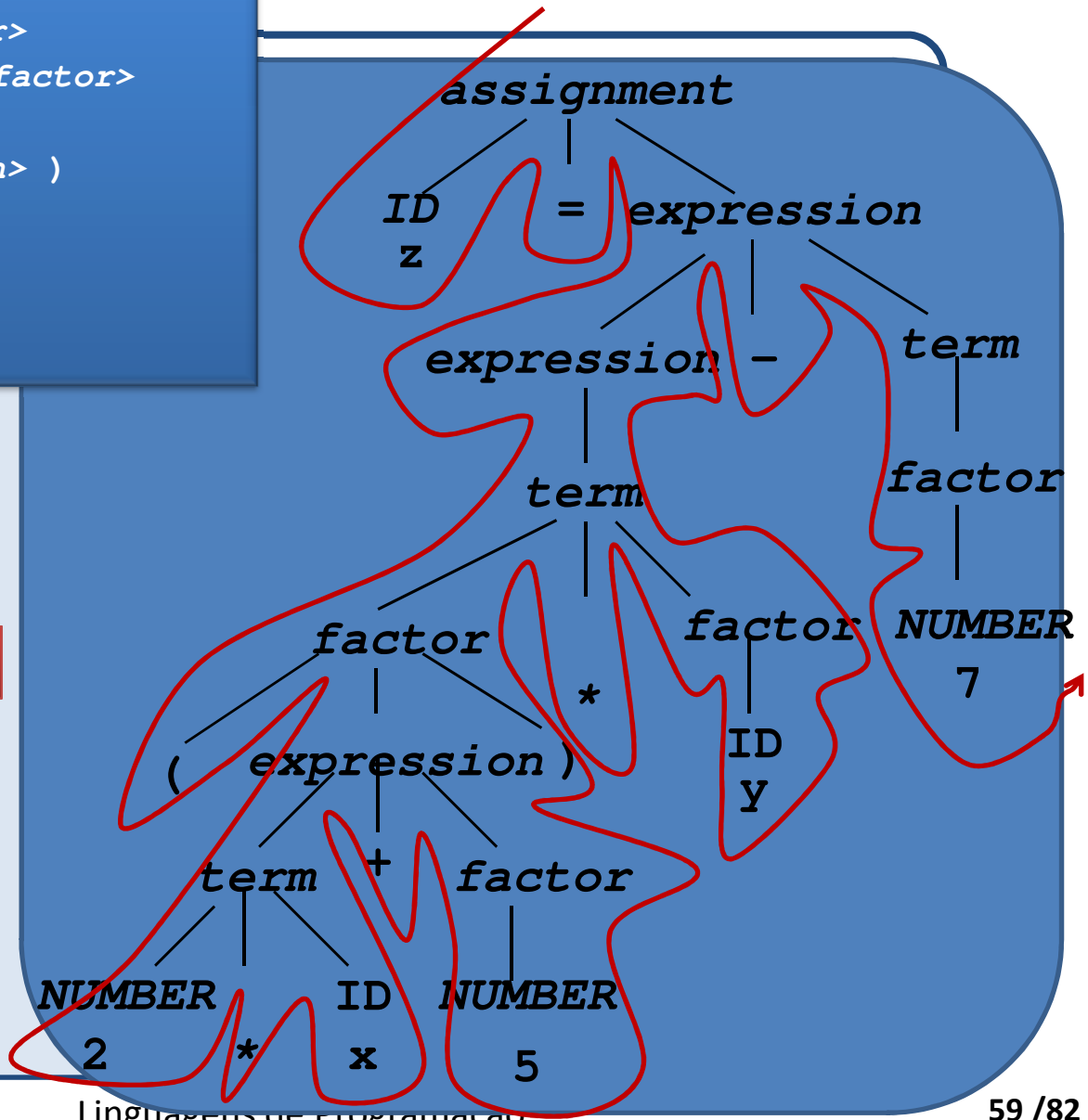
SUCCESS!!

```
<assignment> ::= ID = <expression> ;
<expression> ::= <expression> + <term>
                 | <expression> - <term>
                 | <term>
<term>   ::=  <term> * <factor>
                 | <term> / <factor>
                 | <factor>
<factor> ::=     ( <expression> )
                 | ID
                 | NUMBER
<ID> ::= x|y
<NUMBER> ::= 0|1|2|…|9
```

árvore a partir da entrada:

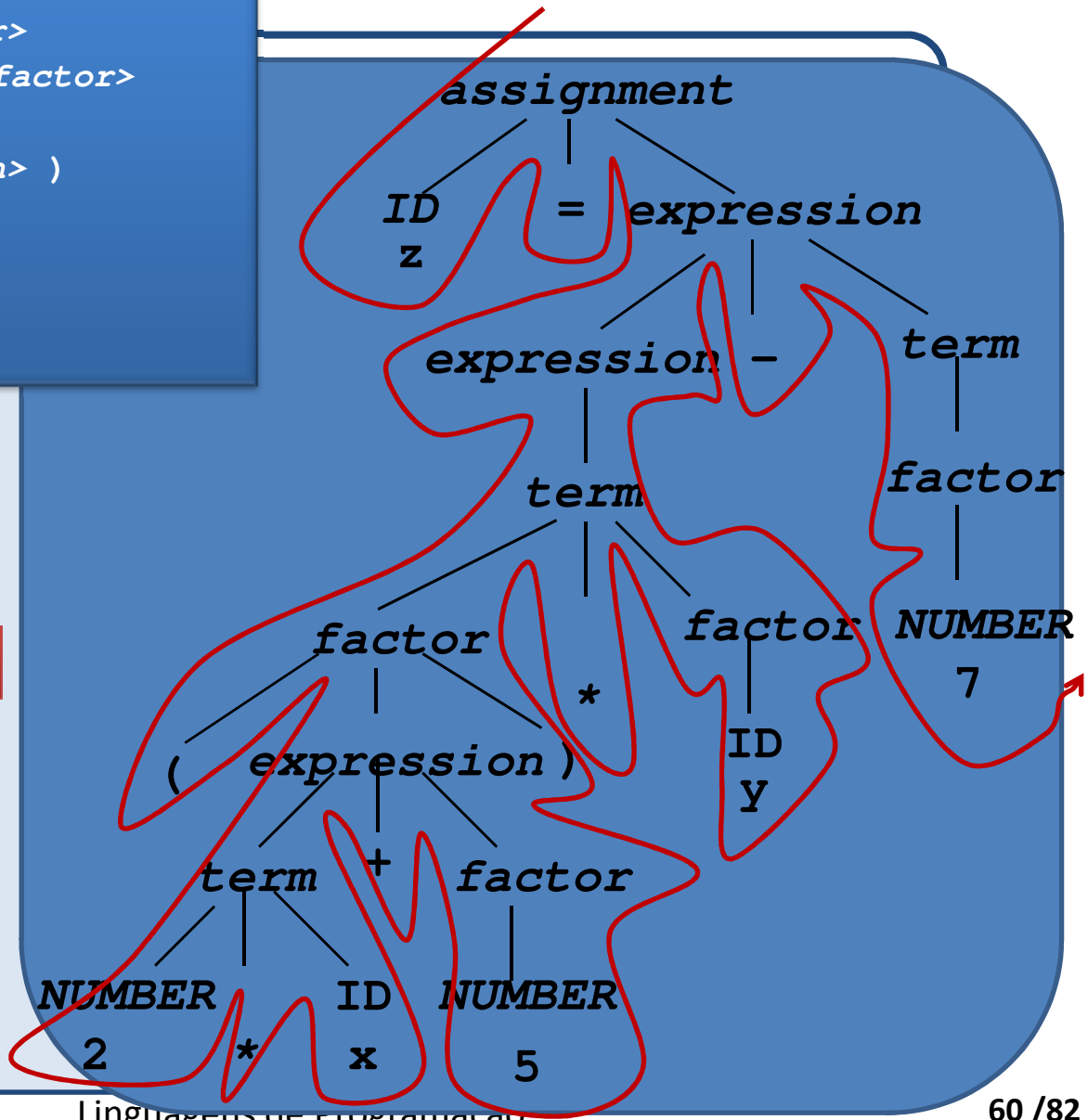
```
z = (2*x + 5)*y - 7;
```

Algumas produções foram omitidas para reduzir o espaço



```
<assignment> ::= ID = <expression> ;
<expression> ::= <expression> + <term>
                 | <expression> - <term>
                 | <term>
<term>   ::=  <term> * <factor>
                 | <term> / <factor>
                 | <factor>
<factor> ::=     ( <expression> )
                 | ID
                 | NUMBER
<ID> ::= x|y
<NUMBER> ::= 0|1|2|…|9
```

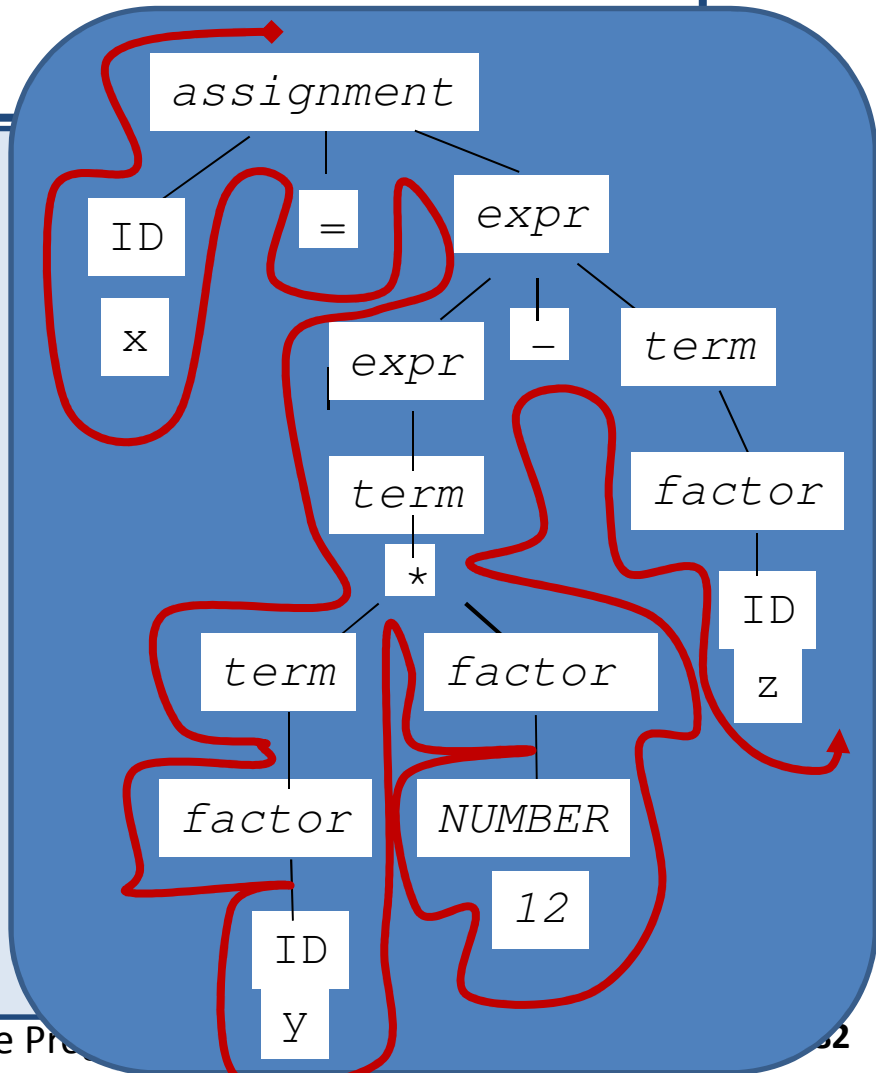
```
z = (2*x + 5)*y - 7;
```



Árvore de análise

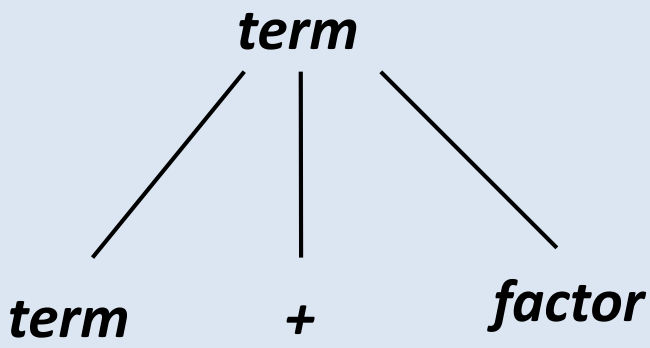
- O *parser* cria uma estrutura de dados representando como uma entrada "encaixa" nas regras da gramática.
- Geralmente como uma árvore.

- Exemplo:
- **x = y * 12 - z**



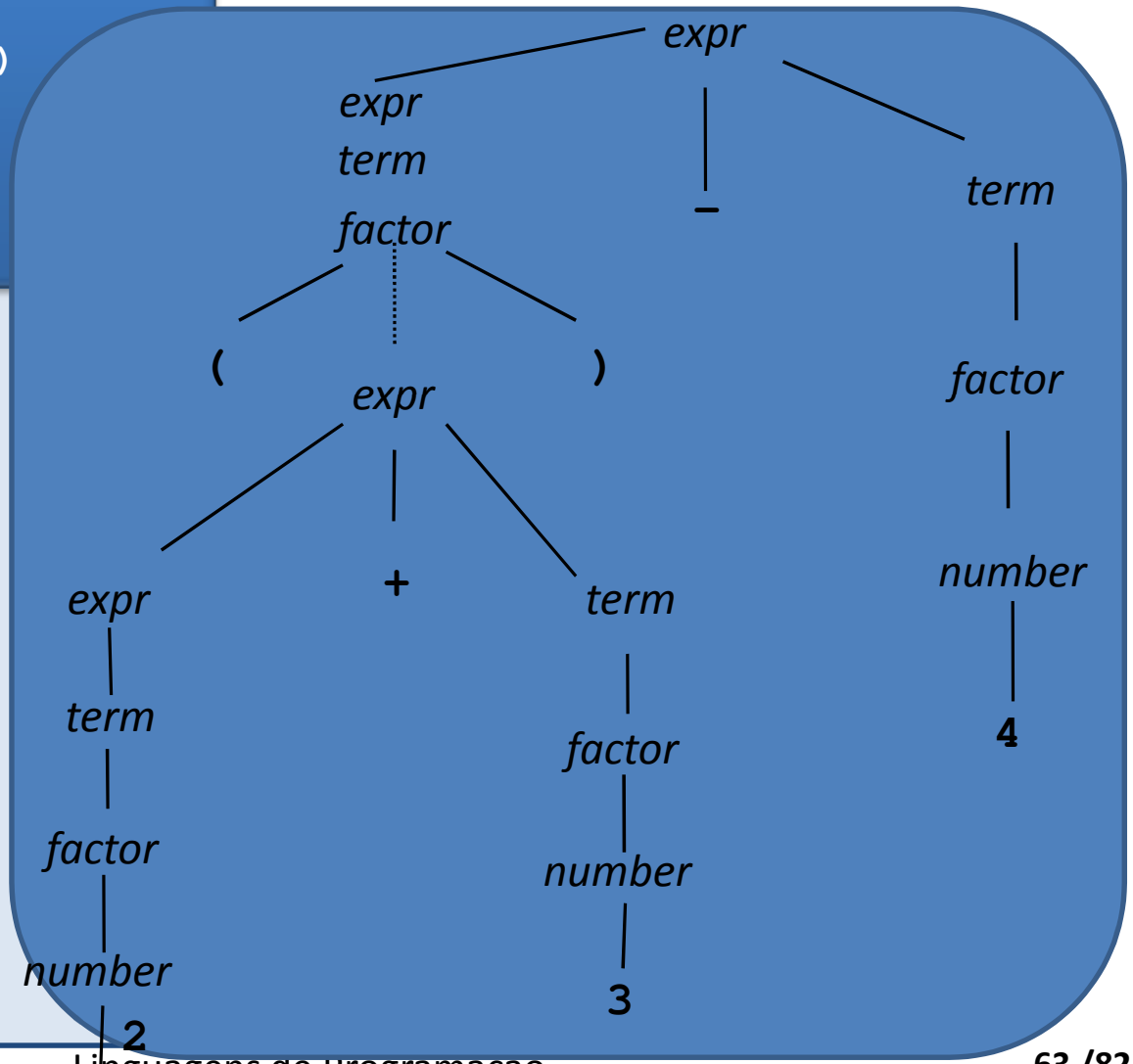
Estrutura da árvore

- O *símbolo inicial* é o nó raiz da árvore.
  - Isso representa a entrada sendo analisada.
- Cada troca é uma derivação (parse) usando uma regra correspondente a um nó e seus filhos.
- Exemplo: ***<term> ::= <term> + <factor>***



```
<assignment> ::= ID = <expression> ;
<expression> ::= <expression> + <term>
                 | <expression> - <term>
                 | <term>
<term>   ::=  <term> * <factor>
                 | <term> / <factor>
                 | <factor>
<factor> ::=     ( <expression> )
                 | ID
                 | NUMBER
<ID> ::= x|y
<NUMBER> ::= 0|1|2|…|9
```

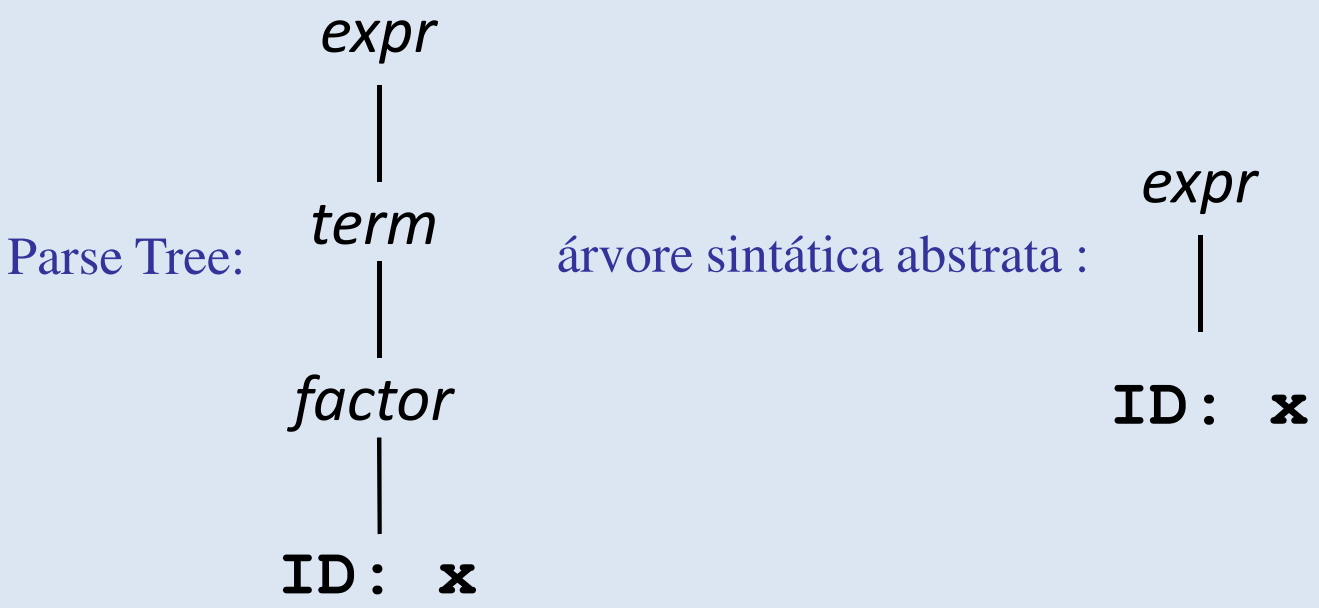
Árvore para (2+3)-4



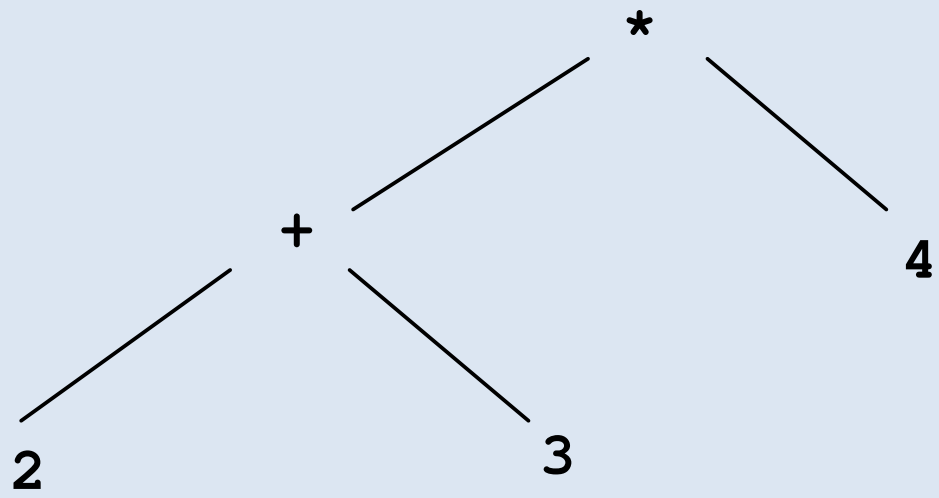
Árvore de sintaxe

- Árvores de Análise são detalhadas: cada passo em uma derivação é um nó.
- Após a análise, os detalhes de derivação não são necessários para fases subsequentes.
- O Analisador Semântico remove as produções intermediárias para criar uma árvore sintática abstrata – (abstract) syntax tree.

Árvore de sintaxe



Exemplo: árvore sintática abstrata para (2+3)*4

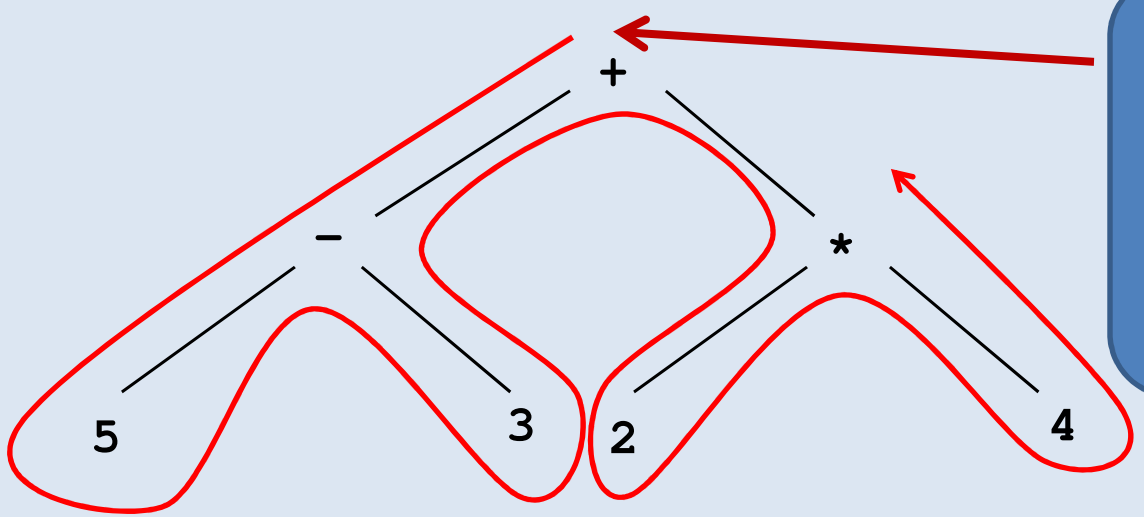


Semântica dirigida pela sintaxe

- A árvore sintática abstrata corresponde à computação realizada.
- Para executar a computação, basta percorrer a árvore in order (in order traversal).
- **Q: o que signifia "in order traversal"?**

Árvore de sintaxe

- Q: o que signifia "in order traversal"?



Sentido da leitura: In Order → Computação ocorre quando visita o nó pela segunda vez

Produz: 5-3+2*4

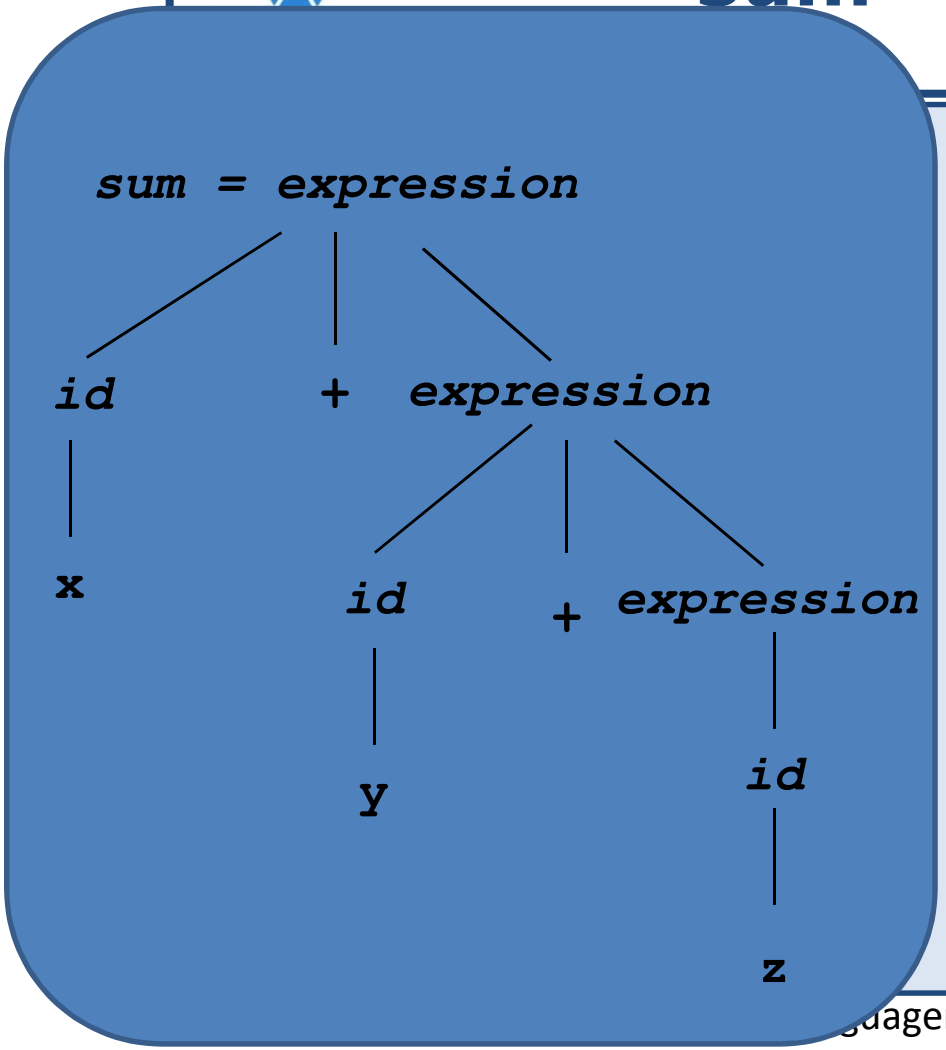
BNF e precedência de operadores

- A ordem de produção afeta a ordem de computação.
- Considere a BNF:

```
<assignment> => id = <expression> ;
<expression> => id + <expression>
              | id - <expression>
              | id * <expression>
              | id / <expression>
              | id
              | number
```

- Essa BNF descreve a aritmética padrão?

Ordem de operações. Exemplo: sum = x + y + z;



Aplicação da regra:

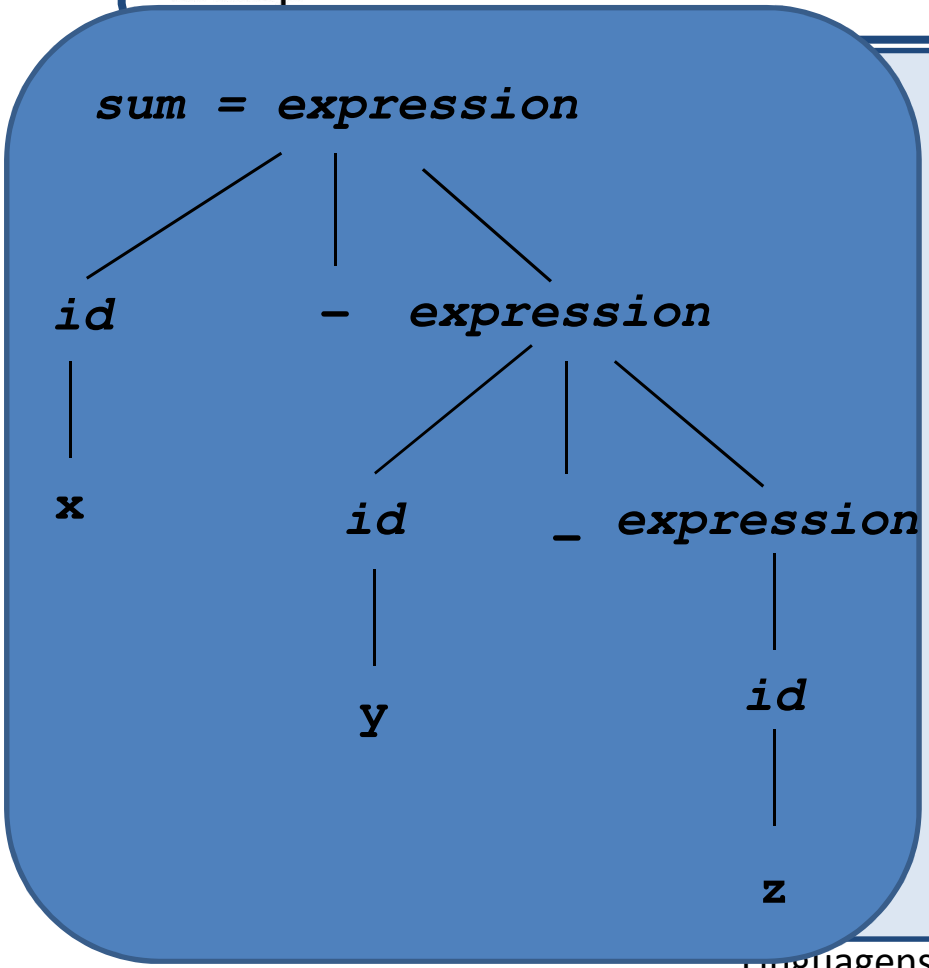
```
assignment
id  = expression
id  = id + expression
id  = id + id + expression
id  = id + id + id
sum =  x +  y +  z
```

Resultado:

```
sum = x + (y + z)
```

Não exatamente correto
(associativo a direita)

Ordem de operações. Exemplo: sum = x - y - z;



Aplicação da regra :

```
assignment
id  = expression
id  = id - expression
id  = id - id - expression id
= id - id - id
sum =  x -  y -  z
```

Resultado:

```
sum = x - (y - z)
```

Errado! *Subtração não é associativo a direita*

O Problema da associatividade a direita

- O Problema é que as regras apresentadas são recursivas à direita. Isso produz *parse tree* que é associativa a direita.

```
<expression> ::= id + <expression>
               | id - <expression>
               | id * <expression>
```

- Solução: definir a regra de derivação recursiva à esquerda. Isso Produz uma parse associativa a esquerda.

```
expression => expression + id
            | expression - id
            | ...
```

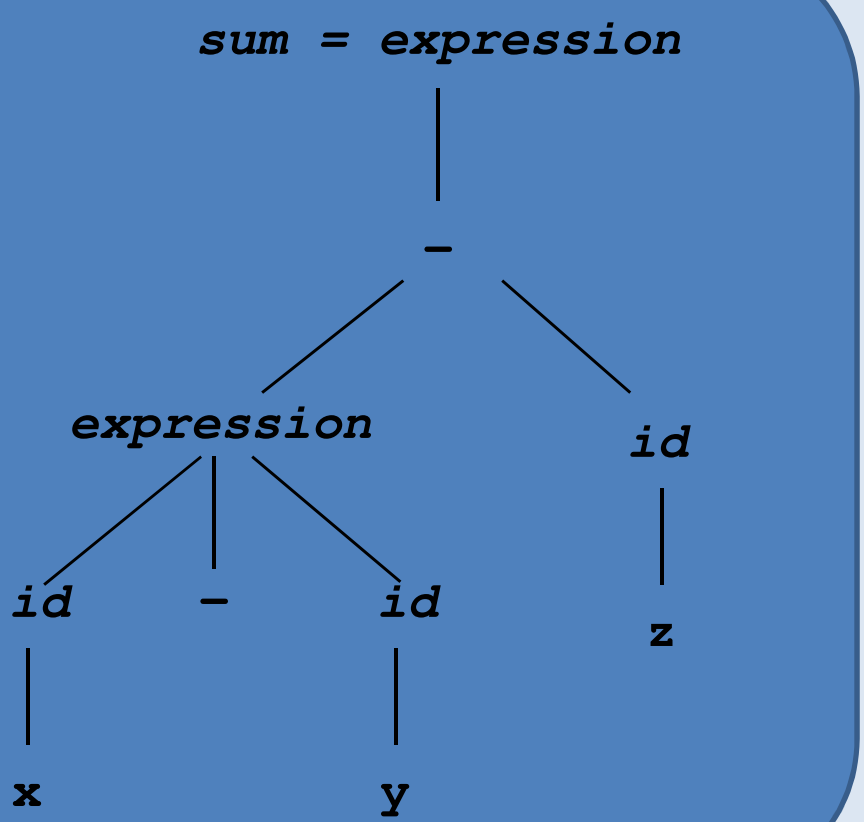
Gramática revisada (1)

- A regra da gramática deveria usar *recursão à esquerda* para gerar *associação a esquerda* de operadores

```
assignment =>  id = expression ;
expression => expression + id
            | expression - id
            | expression * id
            | expression / id
            | id
            | number
```

- Isso funciona?

Ordem de operações. Exemplo: sum = x - y - z;



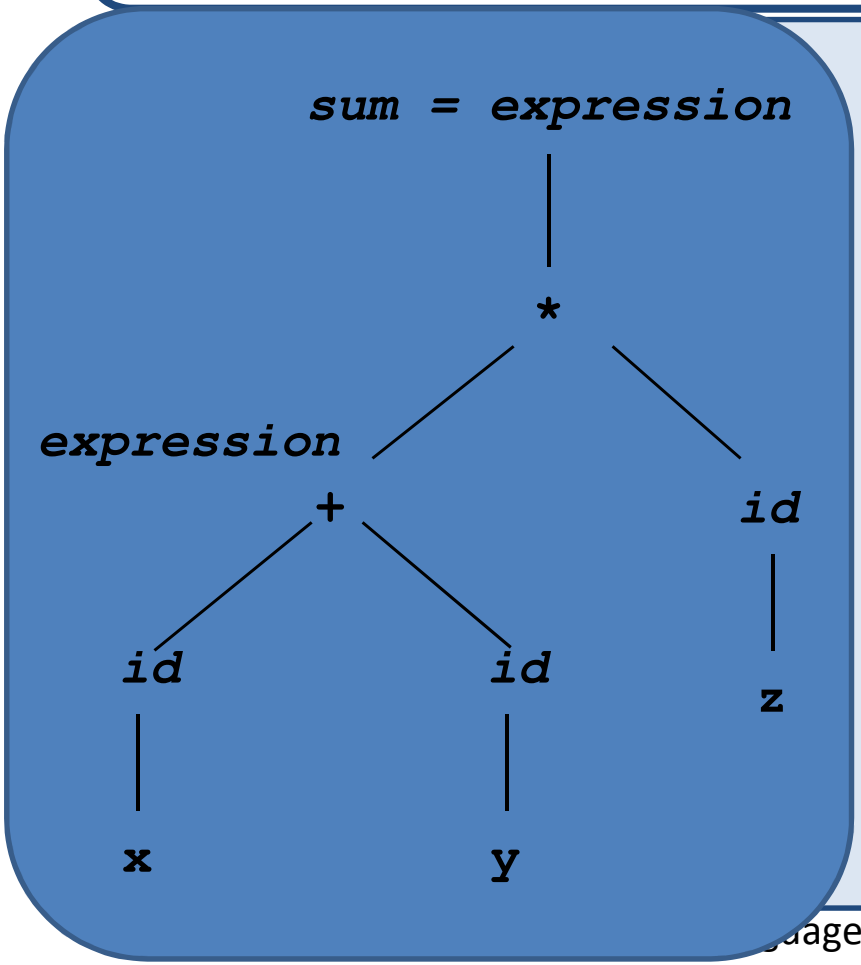
Aplicando a Regra:

```
sum = expression
sum = expression - id
sum = expression - z
sum = expression - id - z
sum = expression - y  - z
sum = id - y          - z
sum = x - y           - z
```

Resultado:

```
sum = (x - y) - z
```

Ordem de operações. Exemplo: sum = x + y * z;



Aplicando a Regra:

```
sum = expression
sum = expression * id
sum = expression * z
sum = expression + id * z
sum = expression + y  * z
sum = id + y          * z
sum = x - y           - z
```

Result:

```
sum = (x + y) * z
```

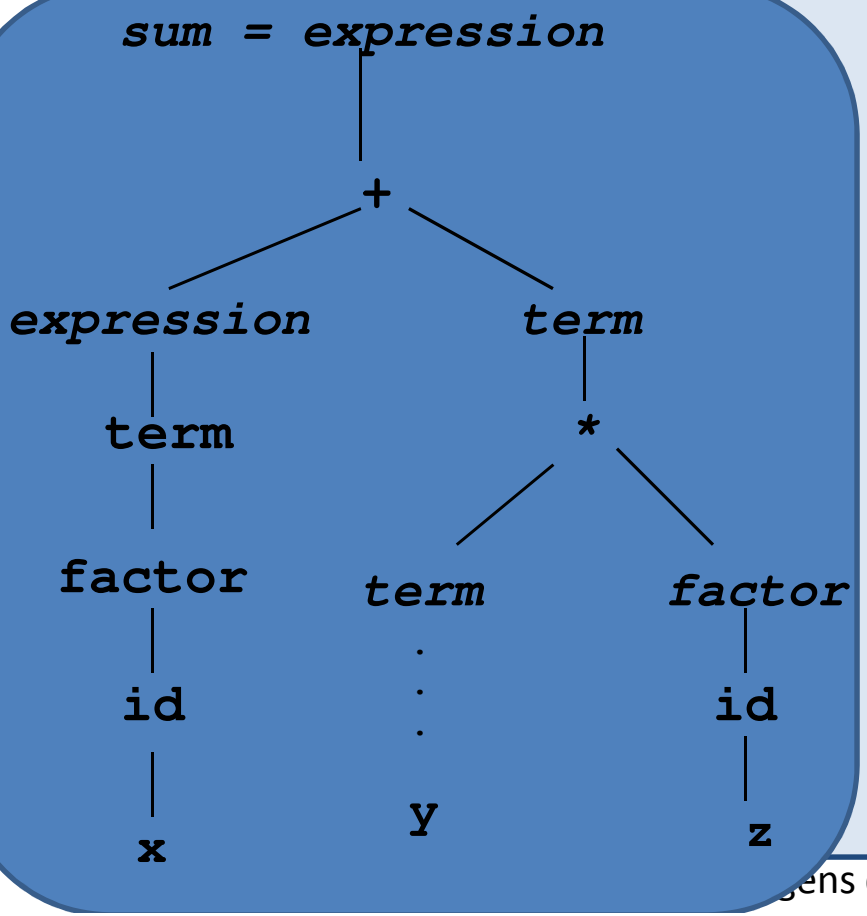
Errado!

Gramática revisada (2)

- Para obter a *precedência* de operadores, é preciso definir outras regras

```
assignment =>  id = expression ;
expression => expression + term
            | expression - term
            | term
term   =>  term * factor
            | term / factor
            | factor
factor =>   ( expression )
            | id
            | number
```

```
assignment =>  id = expression ;
               expression => expression + term
                           | expression - term
                           | term
               term   =>  term * factor
                           | term / factor
                           | factor
               factor =>          ( expression )
                           | id
                           | number
```



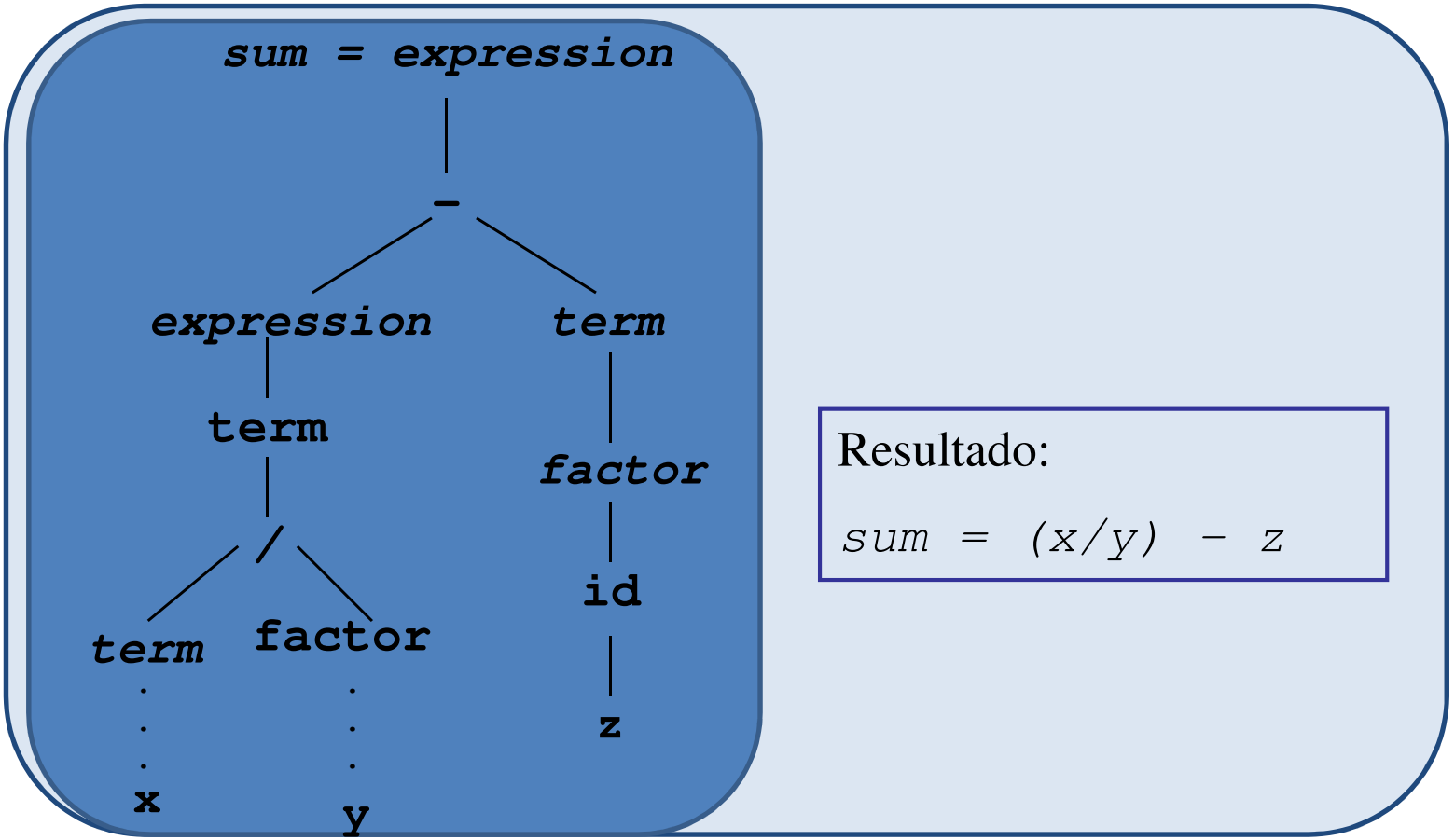
Aplicando a regra:

```
sum = expression
sum = expression + term
sum = term + term
sum = factor + term
sum = id + term
sum = x  + term
sum = x  + term * factor
   ...
```

Resultado:

```
sum = x + (y * z)
```

Ordem de operações. Exemplo: sum = x / y - z;



Resultado:

`sum = (x/y) - z`

Ambiguidade

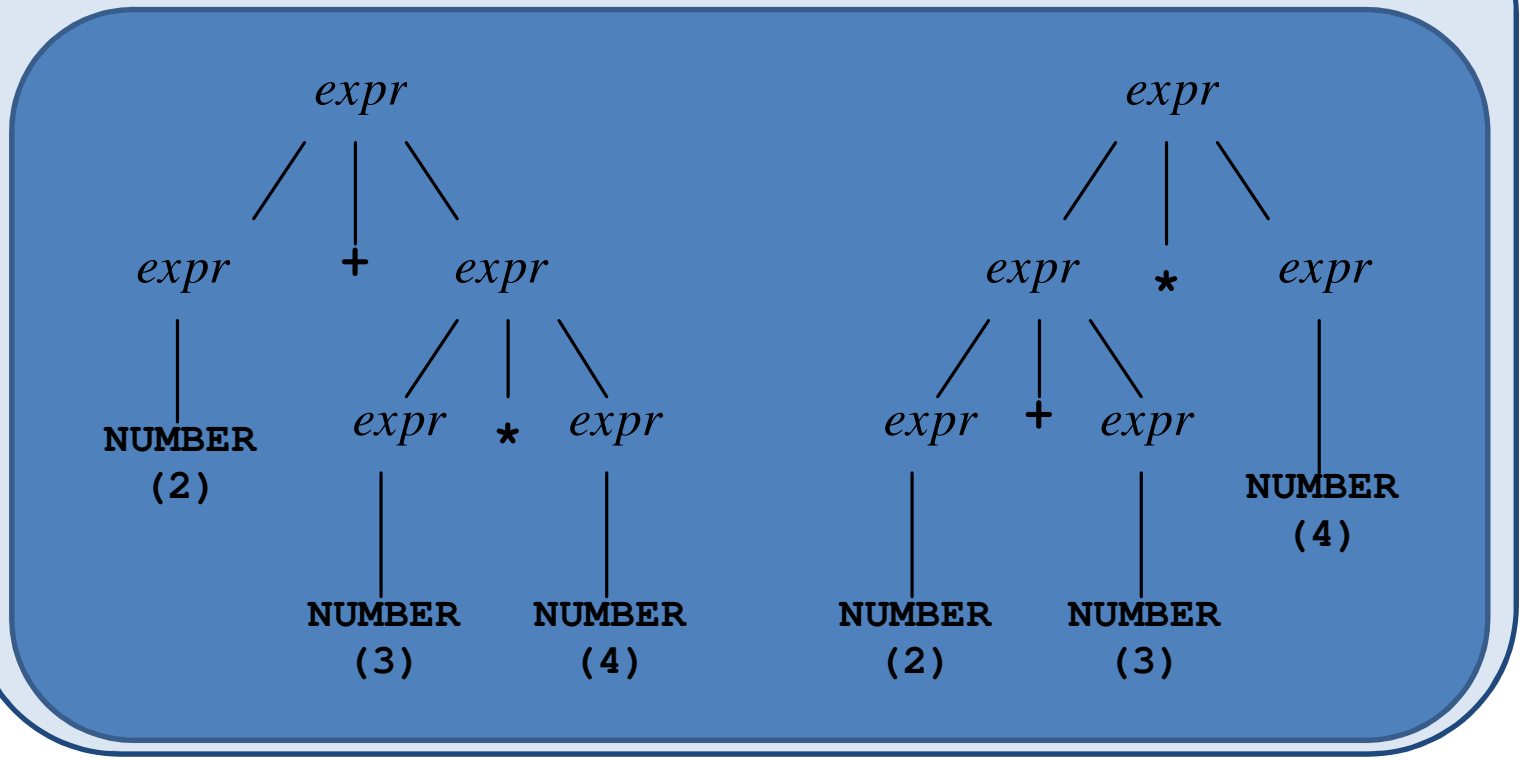
- Uma gramática é dita ambígua se houver mais de uma árvore gramatical ou mais de uma derivação para um sentença válida.
- Exemplo:

```
<expr> ::= <expr> + <expr>
         |<expr> * <expr>
         | id
         | number
```

- Como é a árvore (e a derivação) para **2 + 3 * 4** nessa regra?

Ambiguidade

- Duas árvores possíveis



Ambiguidade

- Para resolver a ambiguidade:
  - Reescreva as regras removendo a ambiguidade
  - *EBNF* (mais adiante) ajuda a remover ambiguidade

Projeto

- Façam os exercícios que encontram-se nos slides anteriores
- A definição do projeto encontra-se na área de projetos da disciplina
  - Pode ser feito em grupos de até 3 pessoas
  - Projeto está dividido em fase 1 e fase 2.
  - A apresentação das fases (1 e 2) será em data posterior a prova – a definir.