Paradigmas de Linguagens de Programação

Sintaxe e semântica

Sintaxe e Semântica

- A **sintaxe** de uma linguagem de programação é a forma de suas expressões, sentenças e unidades de programa.

- A **semântica** de uma linguagem de programação é o significado dessas expressões, sentenças e unidades de programas

Sintaxe e Semântica

- Exemplo:
- a **sintaxe** de uma sentença **while** em Java:

while (expressão_booleana) sentença

- a **semântica** desse formato de setença é que quando o valor atual da expressão booleana for verdadeiro, a senteça dentro da estrutura será executada.

Sintaxe + Semântica: em conjunto definem uma linguagem

Sintaxe e Semântica em Português

O carteiro entregou a carta.

- **Sintaxe:**
  - O ----------(artigo definido)
  - carteiro ---(substantivo)
  - entregou ----(verbo)
  - a ----------(artigo definido)
  - carta -------(substantivo)

- **Semântica:**

- O profissional realizou a entrega de um texto impresso ao seu destinatário.

Descrever Sintaxe

- Uma linguagem (natural ou artificial) é um conjunto de **cadeias de caracteres** formadas a partir de um **alfabeto**.
- **Sentença**
  - Cadeia de caracteres de uma linguagem.
  - Unidade sintática completa.
- **Lexema**
  - Sequência minima de carcteres que formam partes reconhecíveis de uma sentença.
- **Token**
  - Categoria de lexemas
  - Tipo sintático que um lexema representa

Descrever Sintaxe

- Exemplo de **sentença** em Java:

index = 2 * count + 17

Lexemas	Tokens
index	identificador
=	sinal_de_igualdade
2	literal_inteiro
*	operador_de_multiplicacao
count	identificador
+	operador_de_adição
17	literal_inteiro
;	ponto_e_virgula

Reconhecedores e Geradores de Linguagens

- Dado uma linguagem $\boldsymbol{L}$ que usa um alfabeto $\boldsymbol{\Sigma}$ de caracteres
- Uma linguagem pode ser formalmente definida de duas formas:

- Reconhecedor
  - Um mecanismo $\boldsymbol{R}$ (dispositivo de reconhecimento) capaz de ler cadeias de caracteres do alfabeto $\boldsymbol{\Sigma}$
  - Decide se uma cadeia $\boldsymbol{w} \in \boldsymbol{\Sigma}^*$ percente ou não a $\boldsymbol{L}$
  - $\boldsymbol{R(w)} = \begin{cases} \boldsymbol{aceita, se\ w} \in \boldsymbol{L} \\ \boldsymbol{rejeita, se\ w} \notin \boldsymbol{L} \end{cases}$
  - Exemplo: analisador sintático de um compilador

Reconhecedores e Geradores de Linguagens

- Dado uma linguagem ***L*** que usa um alfabeto **∑** de caracteres
- Uma linguagem pode ser formalmente definida de duas formas:

- Gerador
  - Um dispositivo que produz cadeias válidas de ***L***
  - Define como as sentenças da linguagem poder ser formadas
  - **Gramática** = um modelo formal que define como as sentenças válidas de uma linguagem podem ser geradas.
  - Exemplo: gramática em BNF

Uma gramática para Português

- Exemplo de formação de sentenças (simples) em português
- Sentença:
  - Sintagma nominal + verbo + sintagma nominal
    - (grupo de palavras cujo núcleo é um **substantivo** (ex.: “o gato”, “uma carta”)
  - <*S*> → <*SN*><*V*><*SN*>
- Sintagma nominal: <*SN*> → <*A*><*Sb*>
  - Artigo + Substantivo
- Artigo: <*A*> → *o* | *a* | *um* | *uma*
- Verbo: <*V*> → *adora* | *odeia* | *come*
- Substantivo: <Sb> → *cachorro* | *gato* | *rato*

Uma gramática para Português

- “O cachorro adora um gato”

<S> → <SN><V><SN>
Sintagma nominal: <SN> → <A><Sb>
Artigo: <A> → *o* | *a* | *um* | *uma*
Verbo: <V> → *adora* | *odeia* | *come*
Substantivo: <Sb> → *cachorro* | *gato* | *rato*

Uma gramática para Português

- “O cachorro adora um gato”

<S> → <SN><V><SN>

<S> → <A><Sb> <V> <SN>

<S> → O cachorro <V> <SN>

<S> → O cachorro adora <SN>

<S> → O cachorro adora <A><Sb>

<S> → O cachorro adora um gato

<*S*> → <*SN*><*V*><*SN*>
Sintagma nominal: <*SN*> → <*A*><*Sb*>
Artigo: <*A*> → *o* | *a* | *um* | *uma*
Verbo: <*V*> → *adora* | *odeia* | *come*
Substantivo: <Sb> → *cachorro* | *gato* | *rato*

Uma gramática para Português

- “Um rato odeia o gato”

<S> → <SN><V><SN>
Sintagma nominal: <SN> → <A><Sb>
Artigo: <A> → *o* | *a* | *um* | *uma*
Verbo: <V> → *adora* | *odeia* | *come*
Substantivo: <Sb> → *cachorro* | *gato* | *rato*

Uma gramática para Português

- “Um rato odeia o gato”

<S> → <SN><V><SN>

<S> → <A><Sb> <V> <SN>

<S> → Um rato <V> <SN>

<S> → Um rato odeia <SN>

<S> → Um rato odeia <A><Sb>

<S> → Um rato odeia o gato

<*S*> → <*SN*><*V*><*SN*>

Sintagma nominal: <*SN*> → <*A*><*Sb*>

Artigo: <*A*> → *o* | *a* | *um* | *uma*

Verbo: <*V*> → *adora* | *odeia* | *come*

Substantivo: <Sb> → *cachorro* | *gato* | *rato*

Mais um exemplo

<*exp*> → <*exp*> **+** <*exp*>
<*exp*> → <*exp*> ***** <*exp*>
<*exp*> → **(** <*exp*> **)**
<*exp*> → **a**
<*exp*> → **b**
<*exp*> → **c**

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

(a + b)

((a + b)*c)

Mais um exemplo

(a + b)

<*exp*> → <*exp*> **+** <*exp*>
<*exp*> → <*exp*> ***** <*exp*>
<*exp*> → **(** <*exp*> **)**
<*exp*> → **a**
<*exp*> → **b**
<*exp*> → **c**

Mais um exemplo

(a + b)

<exp> → (<exp>)

→ (<exp> + <exp>)

→ (a + <exp>)

→ (a + b)

<*exp*> → <*exp*> **+** <*exp*>
<*exp*> → <*exp*> ***** <*exp*>
<*exp*> → **(** <*exp*> **)**
<*exp*> → **a**
<*exp*> → **b**
<*exp*> → **c**

Mais um exemplo

((a + b)*c)

<*exp*> → <*exp*> **+** <*exp*>
<*exp*> → <*exp*> ***** <*exp*>
<*exp*> → **(** <*exp*> **)**
<*exp*> → **a**
<*exp*> → **b**
<*exp*> → **c**

Mais um exemplo

((a + b)*c)

<exp> → (<exp>)

→ (<exp> * <exp>)

→ ((<exp>) * <exp>)

→ ((<exp> + <exp>) * <exp>)

→ ((a + <exp>) * <exp>)

→ ((a + b) * <exp>)

→ ((a + b) * c)

<*exp*> → <*exp*> **+** <*exp*>
<*exp*> → <*exp*> ***** <*exp*>
<*exp*> → **(** <*exp*> **)**
<*exp*> → **a**
<*exp*> → **b**
<*exp*> → **c**

Hierarquia de Chomsky

- O linguista **Noam Chomsky (1956, 1959)**, definiu as quatro classes de gramáticas formais
- Possui 4 níveis, sendo que os **dois últimos níveis** são amplamente utilizados na descrição de linguagens de programação e na implementação de interpretadores e compiladores.
  - Tipo 0 – Gramáticas com estrutura de frase ou irrestrita.
  - Tipo 1 – Gramáticas sensíveis ao contexto.
  - Tipo 2 – **Gramáticas livres de contexto**.
  - Tipo 3 – **Gramáticas regulares**.

Gramática BNF

- Criada por **John Backus** (ALGOL 58, 1959).
- Modificada por **Peter Naur** no **ALGOL 60**.
- Tornou-se a notação mais popular para descrever sintaxe de linguagens de programação.
- Hoje, **BNF ≈ gramática livre de contexto** (os termos são usados como sinônimos).
- BNF é uma metalinguagem para linguagens de programação

Gramática BNF

- Uma descrição BNF, ou gramática, é formada por:
  - Elementos:
    - Um conjunto de **símbolos terminais** (tokens)
    - Um conjunto de **símbolos não-terminais** (abstrações)
    - Um **símbolo inicial não-terminal** (raiz da árvore de derivação)
  - **Regra ou produção**:
    - <S> → *w*
    - w é uma cadeia de caracteres contendo símbolos terminais e não-terminais

<assign> **→** ***<var>*** **=** ***<expression>***

Lado esquerdo
(LHS – left-hand side)

Lado direito
(RHS – right-hand side)

Gramática BNF

- Quando se usa mais de uma produção com o mesmo *left-hand side*, pode-se usar uma forma abreviada com a barra em pé **|**

<if_stmt> → if (<logic_expr>) <stmt>

<if_stmt> → if (<logic_expr>) <stmt> else <stmt>

Pode ser descrito:

<if_stmt> → if (<logic_expr>) <stmt> **|** *if (<logic_expr>) <stmt> else <stmt>*

Gramática BNF

- Descrição de Listas em BNF
  - Regra/Produção **recursiva**

<ident_list> → identifier | identifier, <ident_list>

- <ident_list> pode ser:
  - Um **identificador único** (x)
  - Um identificador seguido de vírgula e **outra lista:** x, y, z

Gramática BNF

- BNF prevê um símbolo não-terminal especial <empty> ou ε que deve ser usado quando se deseja gerar nada
- Por exemplo, a gramática a seguir define um comando **if-then** típico com a parte **else** opcional

<if_stmt> → if (<logic_expr>) then <stmt> <else-stmt>

<else-stmt> → else <stmt> | <empty>

Gramática BNF

Exemplo: Uma gramática para uma pequena linguagem.

<program> → begin <stmt_list> end

<stmt_list> → <stmt> | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var> | <var> – <var> | <var>

Gramática BNF

Exemplo: begin A = B + C ; B = C end

<program> → begin <stmt_list> end

<stmt_list> → <stmt> | <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var> | <var> – <var> | <var>

Gramática BNF

<program> → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var> | <var> – <var> | <var>

Exemplo: begin A = B + C ; B = C end

<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 + ; <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

Gramática BNF

<program> → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var> | <var> – <var> | <var>

Exemplo: begin A = B + C ; B = C end

<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 ← **Forma sentencial**
→ begin A = B + <var> ; <stmt_list> end
→ begin A = B + C ; <stmt_list> end
→ begin A = B + ; <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 ← **Sentença gerada**

Linguagem

- Uma linguagem definida por uma gramática BNF é o conjunto de todas as strings terminais que podem ser derivadas a partir de um símbolo inicial
- Exemplo: a gramática a seguir pode derivar todas as strings formadas por sequências de dígitos, formando uma linguagem para inteiros positivos (incluindo o zero).

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Linguagem

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

352

<inteiro> → <inteiro><digito>

→ <inteiro><digito><digito>

→ <digito><digito><digito>

→ 3<digito><digito>

→ 35<digito>

→ 352

Linguagem

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

352

<inteiro> → <inteiro><digito>

→ <inteiro><digito><digito>

→ <digito><digito><digito>

→ 3<digito><digito>

→ 35<digito>

→ 352

Aumentando a Linguagem

352

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Aumentando a Linguagem

352 **3.14**

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Aumentando a Linguagem

352 **3.14**

<real> → <inteiro>.<inteiro>

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Aumentando a Linguagem

352 **3.14**

<numero> → <inteiro> | <real>

<real> → <inteiro>.<inteiro>

<inteiro> → <inteiro><digito> | <digito>

<digito> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Árvores de Derivação (*parse trees*)

- Representam a **estrutura hierárquica** de uma sentença.
- Sempre começam com o **símbolo inicial não terminal** da gramática.
- Cada **nó interno** = símbolo **não terminal**.
- Cada **folha** = símbolo **terminal** (palavra final da sentença).

Árvores de Derivação (*parse trees*)

- Exemplo: Uma gramática para sentenças de atribuição simples.

<assign> → <id> = <expr>

<id> → A| B | C

<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

Árvores de Derivação (*parse trees*)

- Exemplo: Uma gramática para sentenças de atribuição simples.

A = B * (A + C)

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

Árvores de Derivação (*parse trees*)

- Exemplo: Uma gramática para sentenças de atribuição simples.

A = B * (A + C)

<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)

<assign> → <id> = <expr>

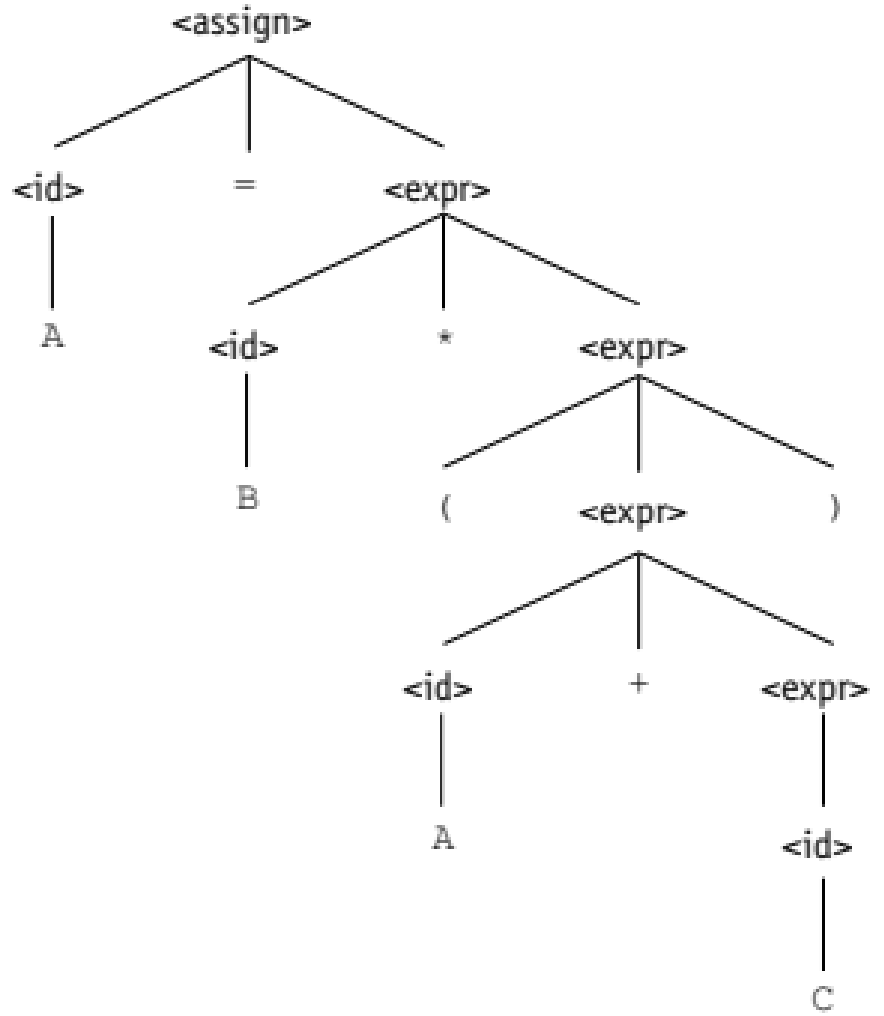
<id> → A| B | C

<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

Árvores de Derivação (*parse trees*)

- Exemplo: Uma gramática para sentenças de atribuição simples.

A = B * (A + C)



Ambiguidade

- Uma gramática que gera uma sentença para a qual existem duas ou mais árvores de análise sintática é dita **ambígua**.

- Exemplo: Uma gramática ambígua para sentenças de atribuição simples.

<assign> → <id> = <expr>

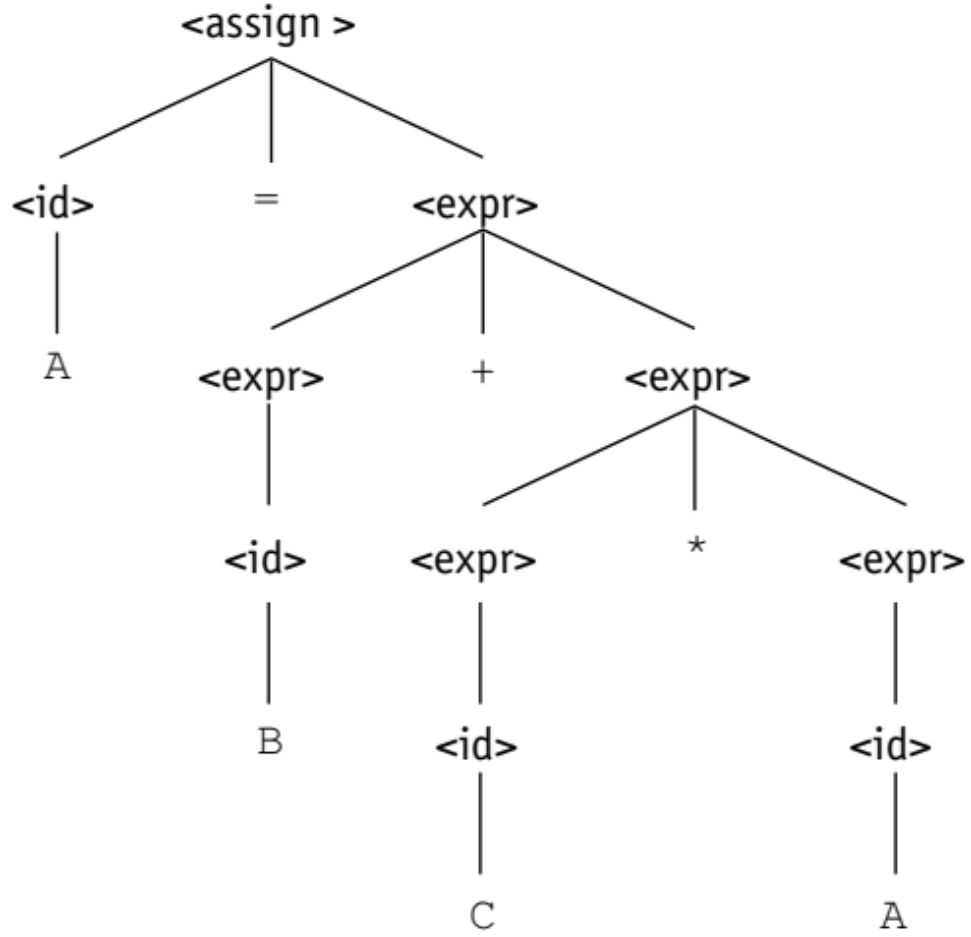
<id> → A | B | C

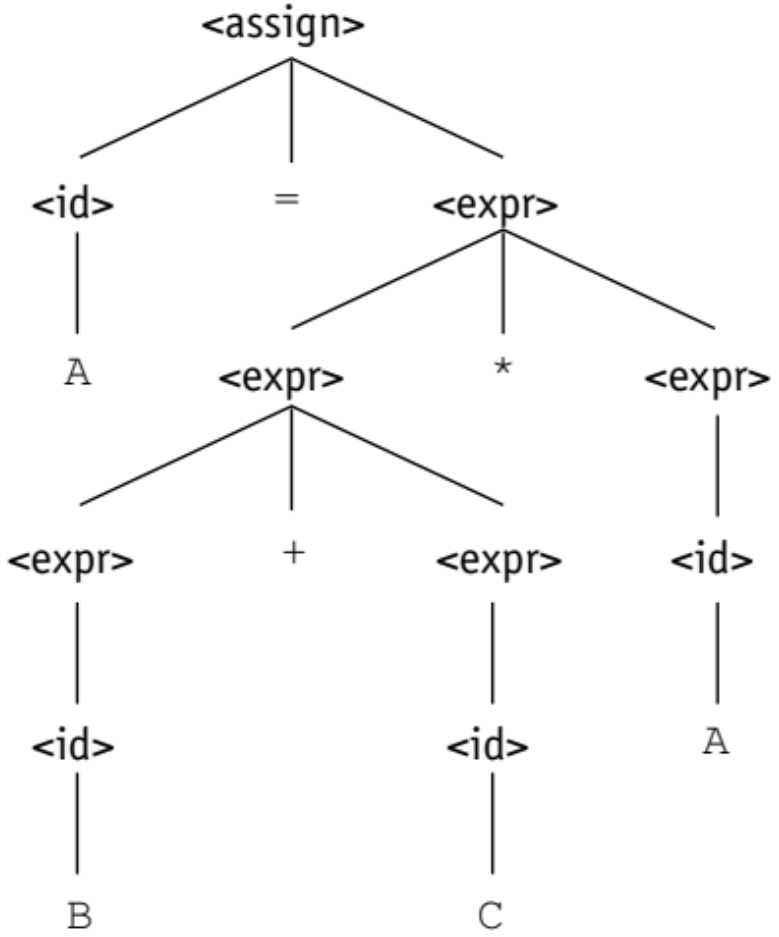
<expr> → <expr> + <expr> | <expr> * <expr> | (<expr>) | <id>

Ambiguidade

- Exemplo: A = B + C * A

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



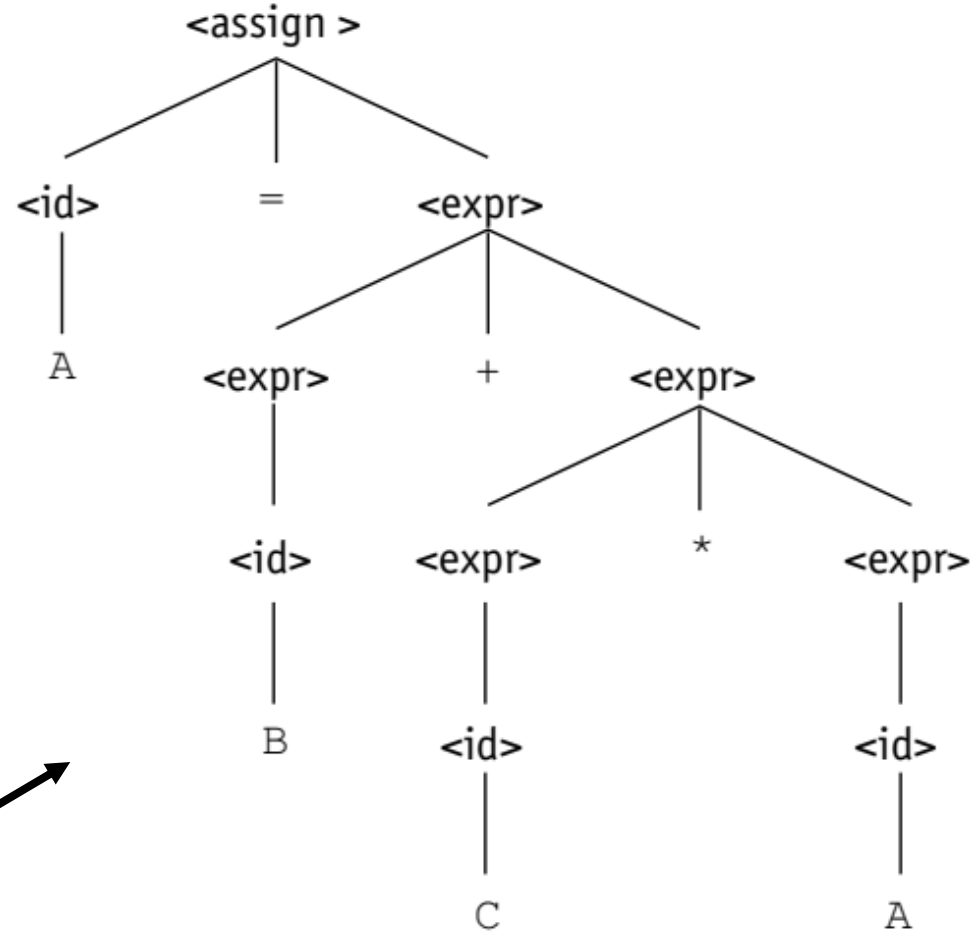


Ambiguidade

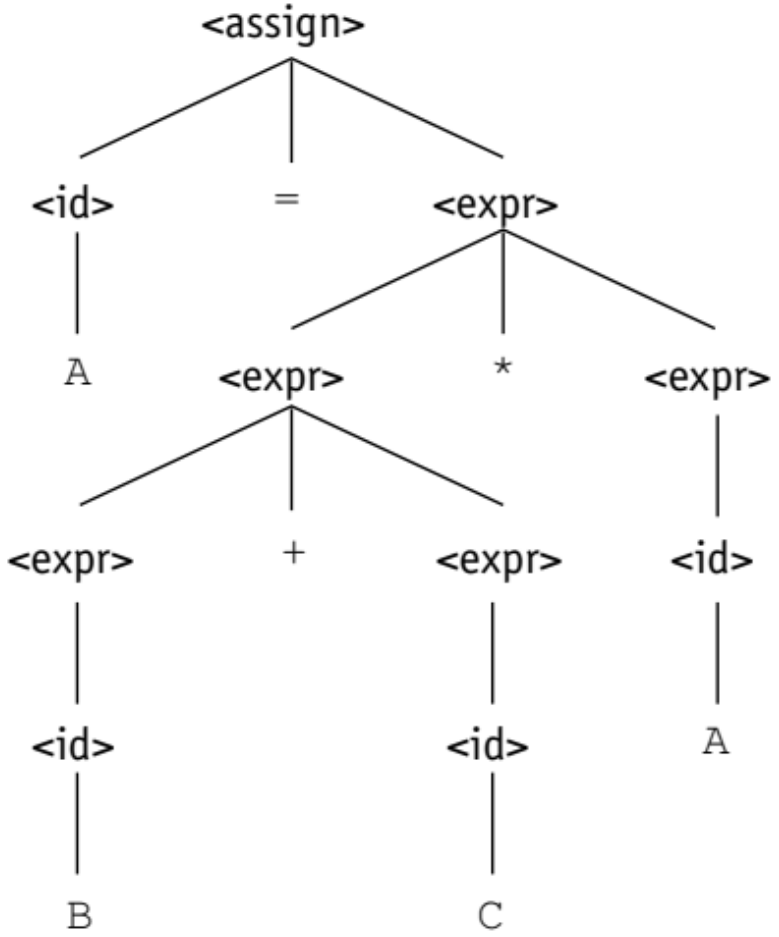
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr> | <expr> * <expr> | (<expr>) | <id>

- Exemplo: A = B + C * A

B = 2
C = 3
A = 4



A = 14



A = 20

Operadores

- **Operador**
  - Símbolo que representa uma operação.
  - Ex.: +, -, *, /, =
- **Operando**
  - Entrada sobre a qual o operador atua.
  - Pode ser identificador, literal ou expressão.
  - Ex.: em A + 3, os operandos são A e 3.
- **Operadores** → símbolos terminais (+, *, etc.)
- **Operandos** → derivados de não-terminais (<id>, <expr>)

Precedência de operadores

- Exemplo: Gerar A + B * C e A * B + C

<assign> → <id> = <expr>

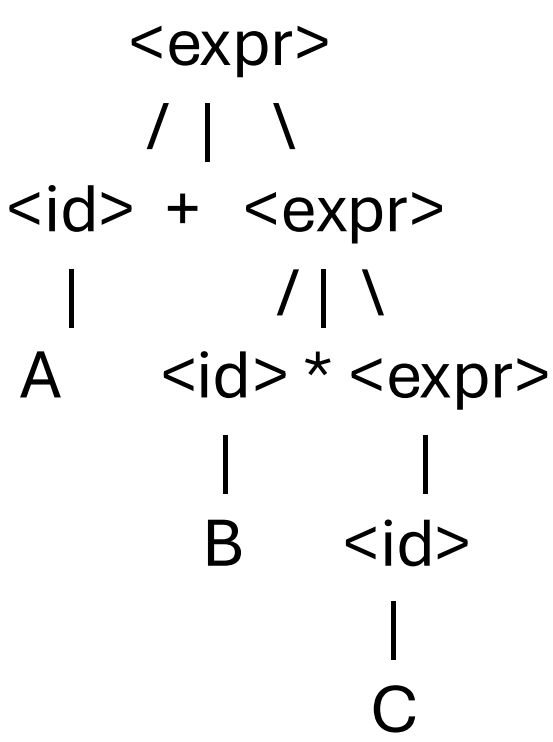
<id> → A| B | C

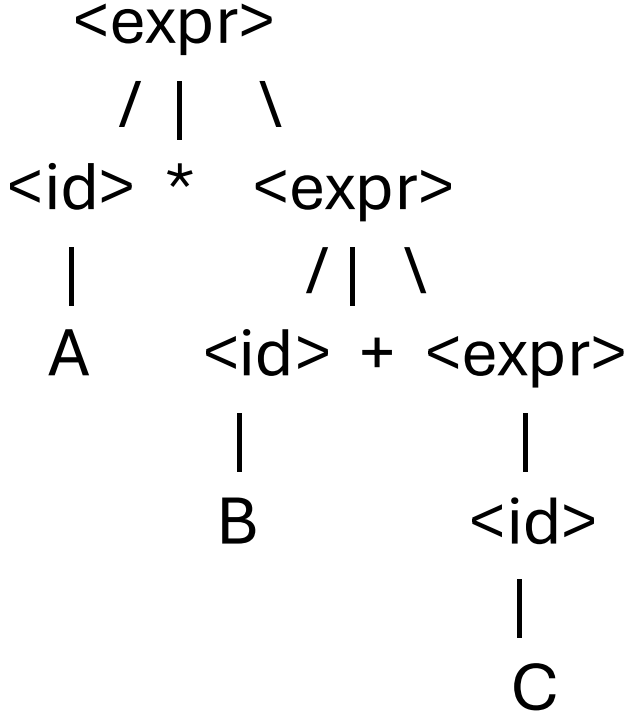
<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

Precedência de operadores

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

- Exemplo: Gerar **A + B * C** e **A * B + C**

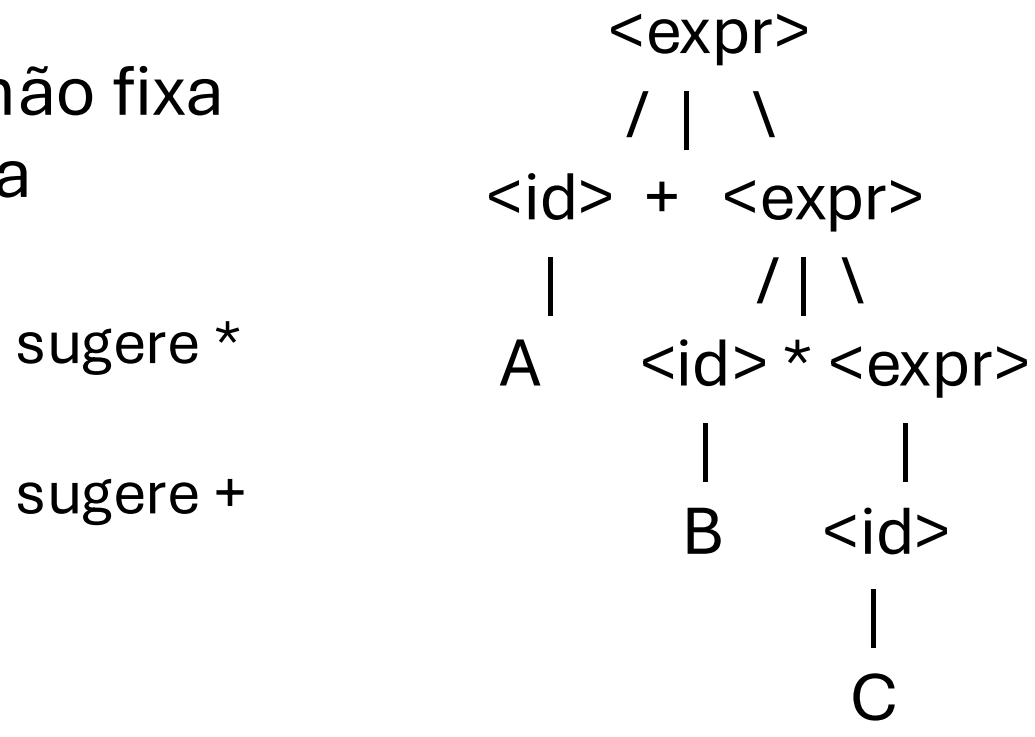


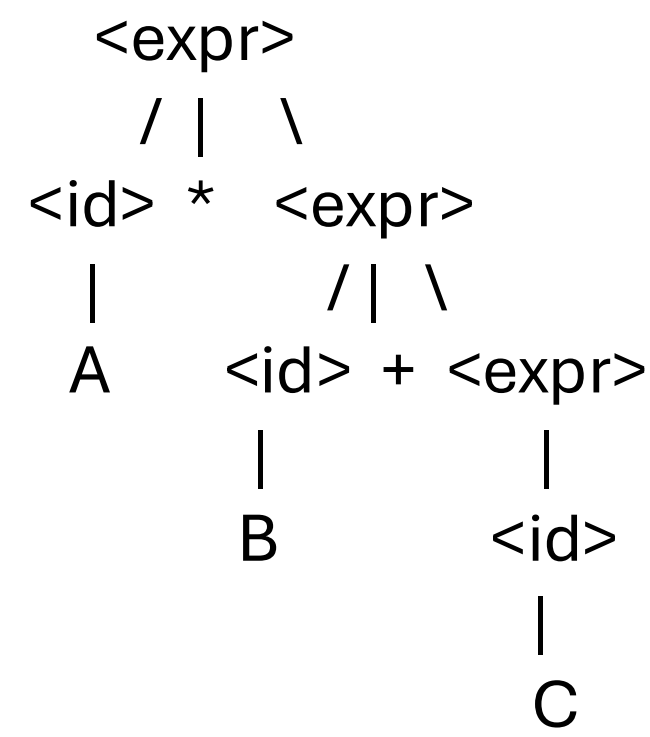


Precedência de operadores

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

- Exemplo: Gerar **A + B * C** e **A * B + C**
- Essa gramática não fixa uma precedência consistente:
  - Em A + B * C ela sugere * antes de +.
  - Em A * B + C ela sugere + antes de *.





Precedência de operadores

- Exemplo: Uma gramática não ambígua para expressões

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | <term>

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>

Precedência de operadores

- Exemplo: A = B + C * A

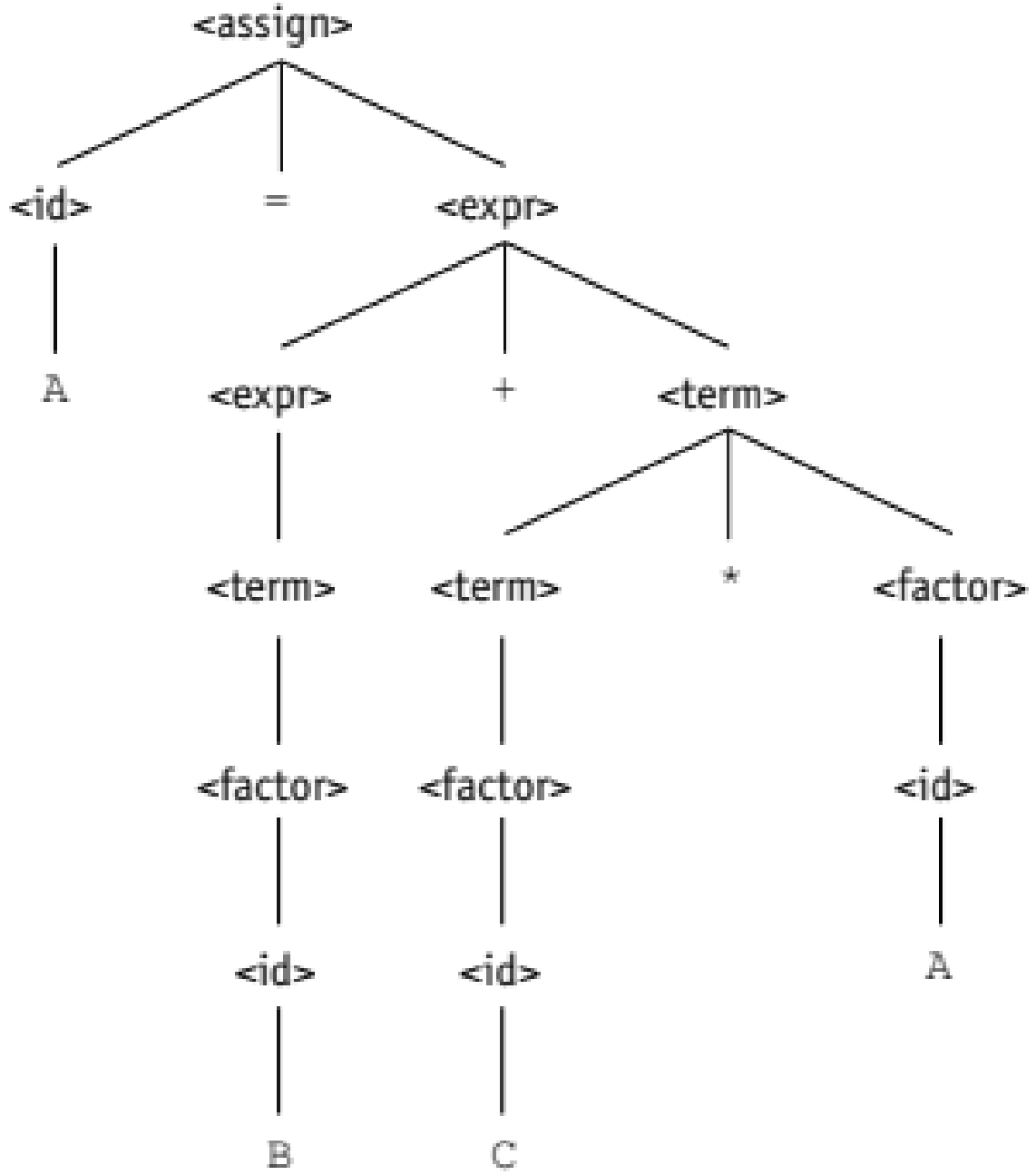
<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term> | <term>

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>



Precedência de operadores

- Exemplo: A = B + C * A
- Cada derivação com uma gramática não ambígua tem uma **única** árvore de análise sintática, apesar de ela poder ser representada por derivações diferentes.

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <term>
         => A = <term> + <term>
         => A = <factor> + <term>
         => A = <id> + <term>
         => A = B + <term>
         => A = B + <term> * <factor>
         => A = B + <factor> * <factor>
         => A = B + <id> * <factor>
         => A = B + C * <factor>
         => A = B + C * <id>
         => A = B + C * A
```

```
<assign> => <id> = <expr>
         => <id> = <expr> + <term>
         => <id> = <expr> + <term> * <factor>
         => <id> = <expr> + <term> * <id>
         => <id> = <expr> + <term> * A
         => <id> = <expr> + <factor> * A
         => <id> = <expr> + <id> * A
         => <id> = <expr> + C * A
         => <id> = <term> + C * A
         => <id> = <factor> + C * A
         => <id> = <id> + C * A
         => <id> = B + C * A
         => A = B + C * A
```

Precedência de operadores

- Gramática pode ser escrita **não ambígua**, respeitando precedência de + e *.
- Usa **não terminais diferentes** para cada nível: <expr>, <term>, <factor>.
- <expr> lida com +
- <term> lida com *
- <factor> lida com parênteses/ids.
- Na árvore:
  - + fica **mais perto da raiz** → menor precedência.
  - * fica **mais abaixo** → maior precedência.

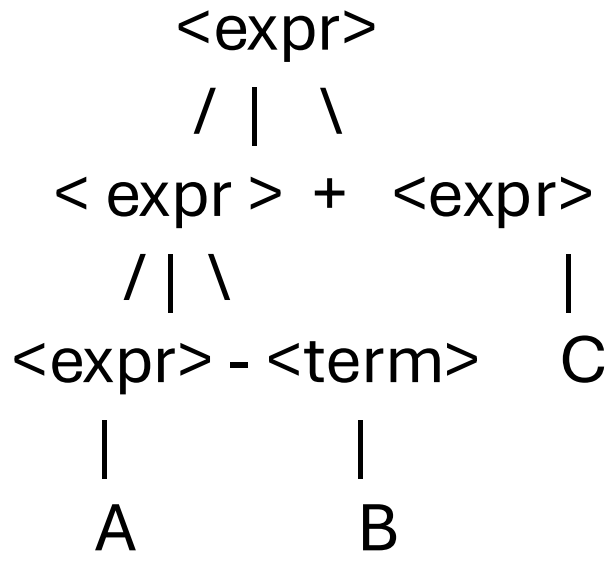
Associatividade de operadores

- Define a ordem de avaliação quando dois operadores têm a mesma precedência.
- Exemplo: A - B - C pode ser:
  - (A - B) - C → associatividade **à esquerda**
  - A - (B - C) → associatividade **à direita**

Associatividade de operadores

- Recursão à esquerda
- O **símbolo não-terminal** aparece no início da regra/produção.
- Garante associatividade à esquerda.
- Exemplo: A – B - C

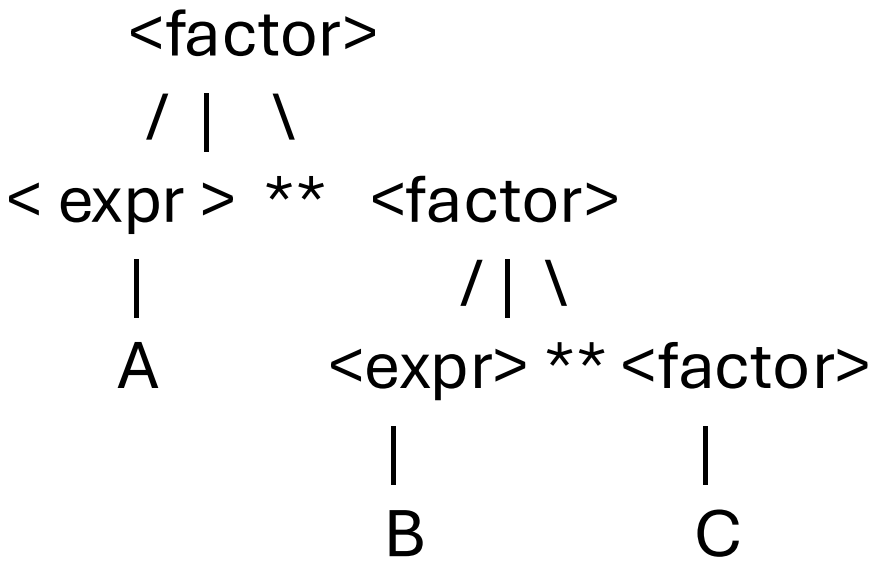
<expr> → <expr> - <term> | <term>



Associatividade de operadores

- Recursão à direita
- O símbolo não-terminal aparece no final da regra/produção.
- Garante associatividade à direita.
- Exemplo: A ** B ** C (exponenciação)

<factor> → <expr> ** <factor> | <id>



BNF Estendida - EBNF

- Versões com o intuito de melhorarem a BNF
- Não aumenta o poder descritivo, apenas a clareza e facilidade de escrita.
- Usa **metassímbolos** (colchetes, chaves, parênteses) para simplificar regras.
- Muito usada em documentação de linguagens de programação modernas.

BNF Estendida - EBNF

- **Parte opcional**
- Parte opcional do lado direito → **colchetes [ ]**.
- Exemplo em C: if-else

EBNF: <if_stmt> → if (<expression>) <statement> [else <statement>]

BNF: <if_stmt> → if (<expression>) <statement> | if (<expression>) <statement> else <statement>

BNF Estendida - EBNF

- **Repetição**
- Parte pode se repetir indefinidamente → **chaves { }**.
- Evita escrever regras recursivas adicionais.

EBNF: <ident_list> → <identifier> { , <identifier> }

BNF: <ident_list> → <identifier> | <identifier> , <ident_list>

BNF Estendida - EBNF

- **Escolha múltipla**
- Várias opções → **parênteses ( )** com **“|”**.
- Substitui múltiplas regras alternativas.

EBNF: <term> → <term> (* | / | %) <factor>

BNF: <term> → <term> * <factor> | <term> / <factor> | <term> % <factor>

BNF Estendida - EBNF

- **Exemplo**

BNF:

<expr> → <expr> + <term>
| <expr> – <term>
| <term>
<term> → <term> * <factor>
| <term> / <factor>
| <factor>
<factor> → <exp> ** <factor>
<exp>
<exp> → (<expr>)
| id

EBNF:

<expr> → <term> { (+ | –) <term>}
<term> → <factor> { (* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

Paradigmas de Linguagens de Programação

Sintaxe e semântica