Descrevendo Sintaxe e Semântica

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Introdução

- Descrição, compreensível, de uma linguagem de programação é difícil e essencial
- Capacidade de determinar como as expressões, instruções e unidades são formadas e suas intenções de efeito quando executadas
- Pode ser dividida em:
  - Sintaxe - forma
  - Semântica - significado
- Sintaxe e semântica estão bastante relacionados
- Descrever sintaxe é mais fácil do que a semântica

Sintaxe (1/2)

- Linguagem
  - Conjunto de cadeias de caracteres de algum alfabeto
- As cadeias são chamadas sentenças ou instruções
- Regras sintáticas de uma linguagem especificam quais cadeias do alfabeto pertencem à linguagem
- Lexema
  - Identificadores, constantes, operadores e palavras especiais
- Token
  - Categoria de lexemas
    - Identificador, operação de adição, ...

Sintaxe (2/2)

```
index = 2 * count + 17;
```

Lexemas	Tokens
index	identificador
=	operador de atribuição
2	constante numérica inteira
*	operador de multiplicação
count	identificador
+	operador de adição
17	constante numérica inteira
;	ponto e vírgula

Reconhecedores

- Suponha
  - Linguagem L sobre um alfabeto $\sum$
- Para definir L usando o método de reconhecimento é preciso construir um mecanismo R de tal forma que quando uma cadeia for fornecida para este mecanismo, este diga se esta cadeia pertence ou não a L
- E como um filtro que separa as sentenças corretas das erradas
- Reconhecimento não é usado para enumerar todas as sentenças de uma linguagem
- A análise sintática de um compilador é um reconhecedor, determinando apenas se um programa está na linguagem

Geradores

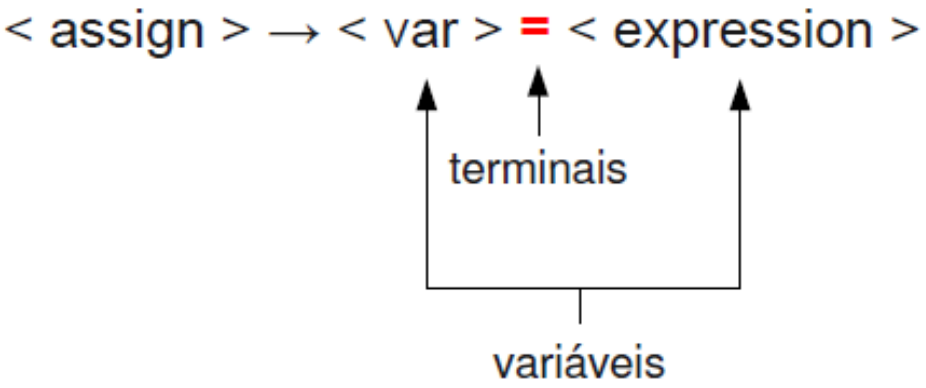
- Dispositivo usado para gerar sentenças de uma linguagem
- É como um botão que, quando pressionado, produz uma sentença da linguagem
- Não é claro que sentença será produzida:
  - Parece ser pouca utilizada como descritor de linguagem
- Para um programador, um reconhecedor (análise sintática) não é tão útil:
  - Só pode ser usado na tentativa e erro
- Geração é preferido a reconhecimento porque é mais fácil de ler e entender

Forma de Backus-Naur (1/6)

- Popularmente conhecida como BNF
- E um método formal para descrição da sintaxe
- Origem
  - ALGOL 58 e ALGOL 60
- E uma notação natural de descrever a sintaxe
- 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



Forma de Backus-Naur (3/6)

- Uma gramática é uma coleção de regras

- Símbolos não-terminais podem ter mais de uma 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 conjunto não-terminal especial
  - start symbol
- Esta geração de sentença é chamada de derivação
- O start symbol representa um programa completo

Forma de Backus-Naur (5/6)

< program > → **begin** < stmt_list > **end**

< stmt_list > → < stmt >

| < stmt >; < stmt_list >

< stmt > → < var > **=** < expression >

< var > → **A** | **B** | **C**

< expression > → < var > **+** < var >

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

| < var >

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

Derivação (1/2)

- Um exemplo de gramática:

< assign > → < id > **=** < expr >

< id > → **A** | **B** | **C**

< expr > → < id > **+** < expr >

| < id > * < expr >

| **(** < expr > **)**

| < id >

Derivação (2/2)

- Derivação à Extrema Esquerda (DEE):

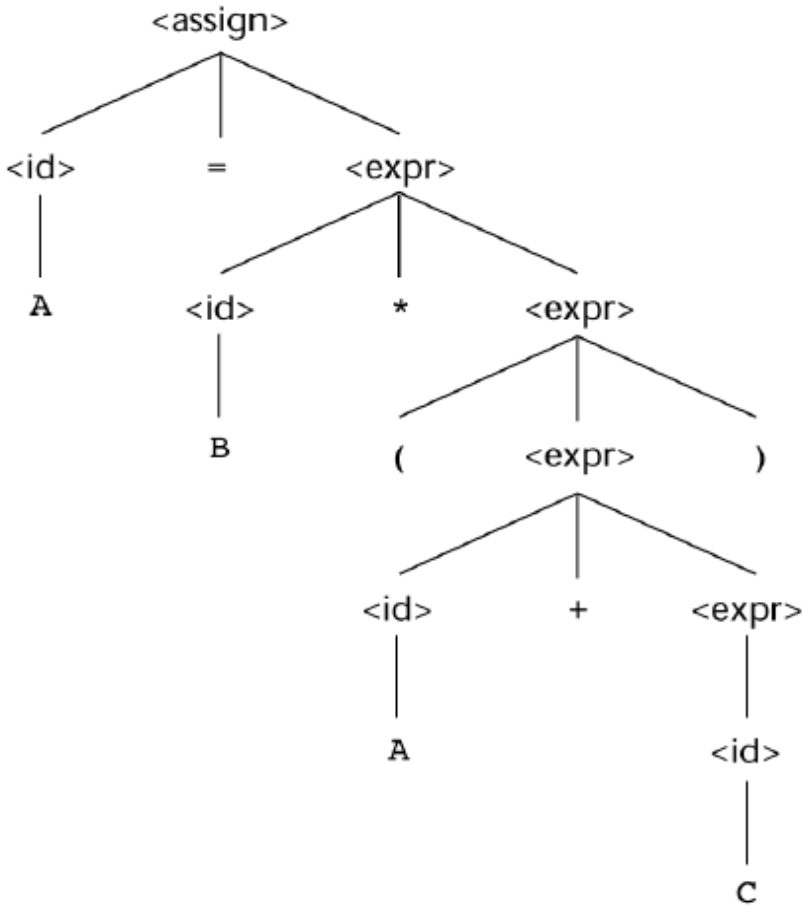
< assign >
< id > = < expr >
A = < expr >
A = < id > * < expr >
A = B * < expr >
A = B * (< expr >)
A = B * (< id > + < expr >)
A = B * (A + < expr >)
A = B * (A + < id >)
A = B * (A + C)

- 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)

Árvore de Derivação

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



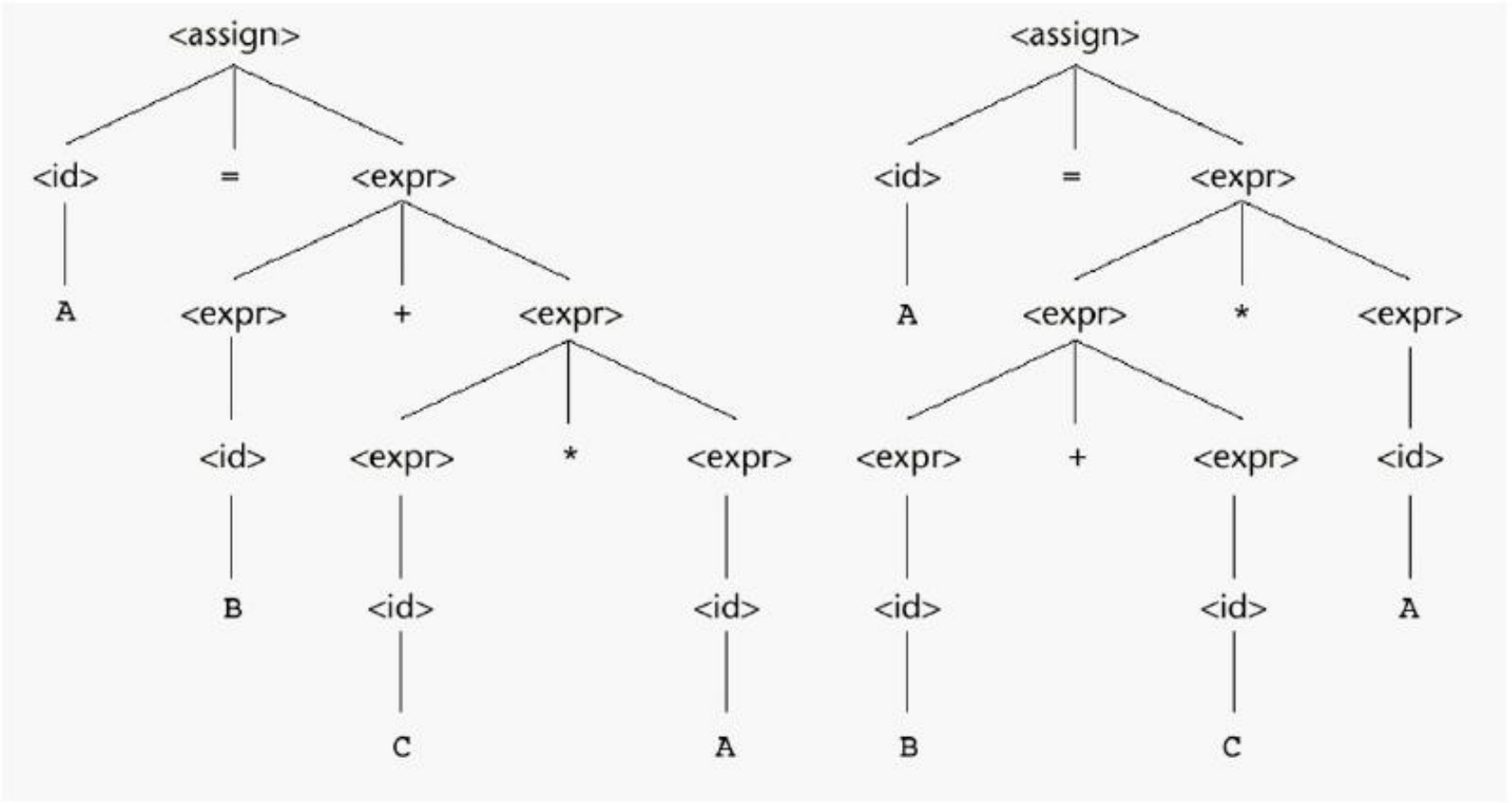
Ambiguidade (1/2)

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

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

Ambiguidade (2/2)

A := B + C * A

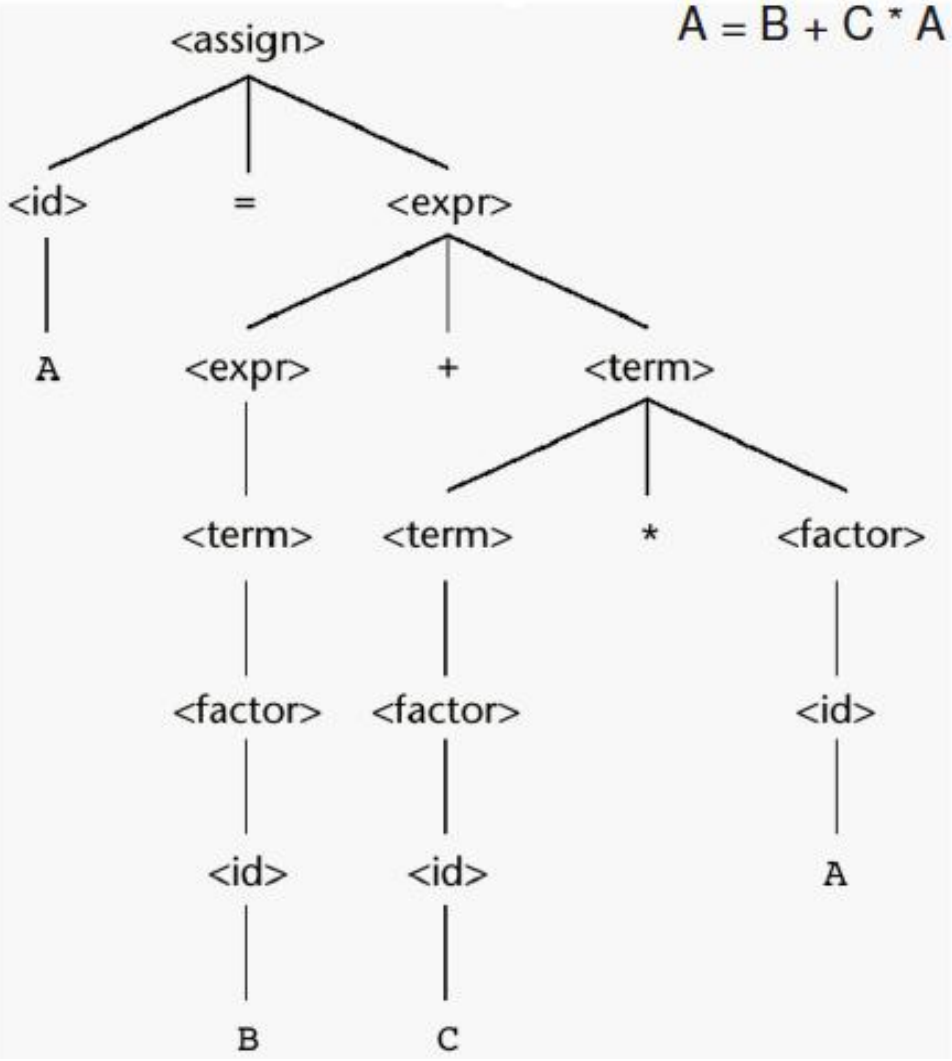


Precedência de Operadores (1/3)

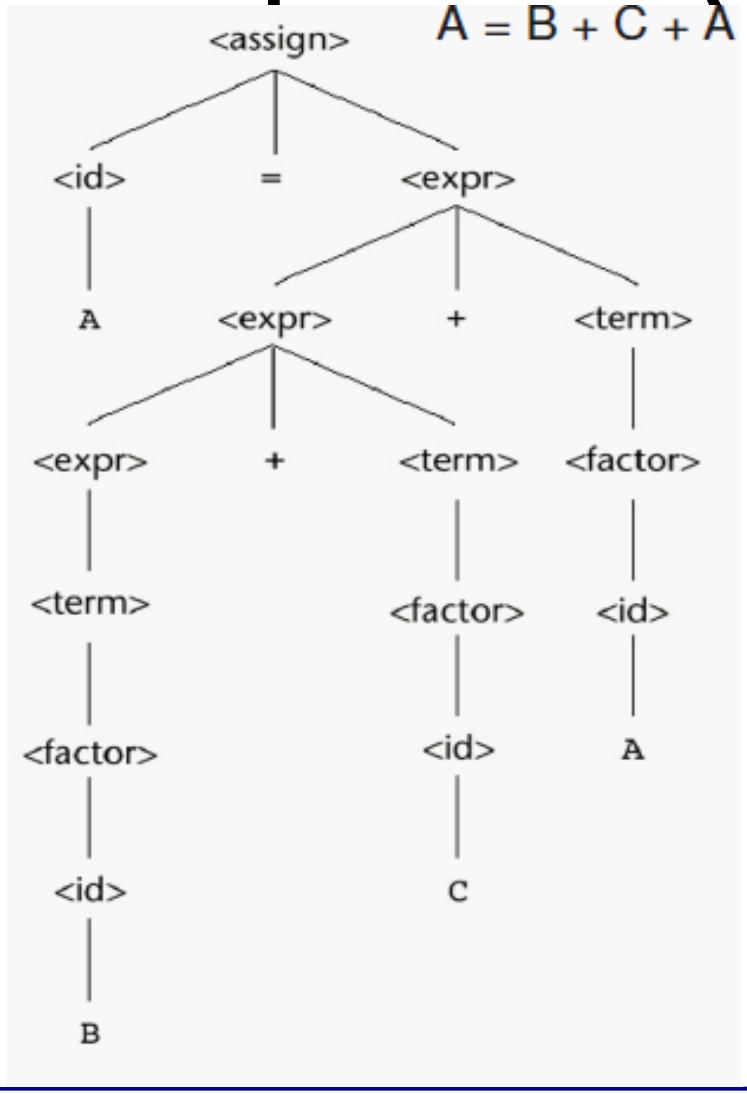
- Uma árvore de derivação pode ser usada para indicar precedência de operadores

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

Precedência de Operadores (2/3)



Precedência de Operadores (3/3)



BNF Estendida (1/3)

- Três extensões foram propostas
  - Não melhoram o poder descritivo, mas aumenta a legibilidade e a redigibilidade

- Parte opcional:
  - < selection > → **if(** < expression > **)** < statment > [**else** < statement >] **;**

- Repeticao:
  - < ident_list > → < identifier > {**,** < identifier >}

- Múltipla escolha:
  - < for_stmt > → **for** < var > **:=** < expr > (**to** | **downto**) < expr > **do** < stmt >

BNF Estendida (2/3)

- BNF

```
< expr >   →  < expr > + < term >
           |  < expr > - < term >
           |  < term >
< term >   →  < term > * < factor >
           |  < term > / < factor >
           |  < factor >
```

- EBNF

```
< expr >   →  < term > { ( + | - ) < term > }
< term >   →  < factor > { ( * | / ) < factor > }
```

BNF Estendida (3/3)

- Algumas versões de EBNF permitem que se coloque um número sobrescrito indicando a quantidade de repetições

- Ou que se coloque um sinal de mais (+) para indicar uma ou mais vezes

< program > → **begin** < stmt > { < stmt > } **end**

< program > → **begin** { < stmt > }$^{+}$ **end**

Grafo de Sintaxe (1/2)

- Um grafo é uma coleção de nós, alguns conectados através de linhas chamadas arestas
- Um grafo direcionado possui arestas direcionais
- As informações de regras BNF podem ser expressas em termos de grafos direcionais
  - Chamados de grafos sintáticos
- Usam diferentes tipos de nós para representar os terminais e os não-terminais
- Usar grafos melhora a legibilidade

Grafo de Sintaxe (2/2)

- Grafo sintático e descrição EBNF da instrução if na linguagem ADA