SBVCONC: Construção de Compiladores

Aula 03: Especificação de Linguagens de Programação

Bacharelado em Ciência da Computação
Prof. Dr. David Buzatto

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Especificação de uma Linguagem de Programação

- Sintaxe (forma):
  - Símbolos básicos da linguagem (tokens);
  - Estrutura de símbolos que é permitida para formar programas;
  - Especificada por uma gramática livre de contexto;
- Restrições Contextuais:
  - Regras e restrições do programa que não podem ser especificadas em uma gramática livre de contexto;
  - Consiste primariamente em regras de tipos e escopos;
- Semântica (significado):
  - Comportamento do programa quando é executado em uma máquina;
  - Usualmente especificada informalmente.

Especificação Informal versus Especificação Formal

- Especificação Informal:
  - Normalmente escrita em uma linguagem natural (português, inglês etc);
  - Facilmente entendida;
  - Difícil de fazer com que seja suficientemente precisa;
- Especificação Formal:
  - Notação precisa;
  - Demanda um certo esforço para entender;
- Prática comum:
  - Sintaxe: especificação formal usando gramática livre de contexto;
  - Restrições contextuais: especificação informal;
  - Semântica: especificação informal.

Sintaxe versus Restrições Contextuais

- Em geral, cada linguagem de programação é "aproximada" por uma gramática livre de contexto. As restrições contextuais restringem a linguagem aproximada para coincidir com a linguagem desejada;
- **Exemplos:** sintaxe válida, mas inválida em relação às restrições contextuais.

```
var x : Integer;
begin
   y := 5;
end.
```

```
var c : Char;
begin
   c := -3;
end.
```

Gramáticas Livres de Contexto

- Fornece uma notação formal para especificar a sintaxe de uma linguagem de programação (metalinguagem);
- Usa uma notação finita para descrever a sintaxe de todos os programas da linguagem que possuem sintaxe válida;
- São poderosas o bastante para lidar com infinitas linguagens;
- Mostra a estrutura dos programas na linguagem;
- Têm sido usadas extensivamente para praticamente todas as linguagens de programação desde a definição da linguagem ALGOL 60;
- Conduz o desenvolvimento do parser.

Gramáticas Livres de Contexto

- Também conhecidas como gramáticas na Forma de Backus-Naur (BNF, Backus-Naur Form);
- Na maioria das vezes são processadas por ferramentas para construção de compiladores, os compiladores de compiladores;
- **Observação:** existem diversas notações diferentes, mas similares, usadas na definição de gramáticas livres de contexto.

Gramáticas Livres de Contexto

Uma gramática livre de contexto consiste em quatro componentes:

1. Um conjunto finito T de símbolos terminais (vocabulário);
   - Representa os símbolos que aparecem na linguagem;
   - Exemplos: `25, x, if, <=`
2. Um conjunto finito N de símbolos não-terminais ou variáveis:
   - Representa as classes sintáticas da linguagem;
   - Exemplos: `expression, statement, loopStmt`
3. Símbolo ou variável inicial:
   - Um dos símbolos não-terminais;
   - Algo como `program` ou `compilationUnit`
4. Um conjunto finito de regras P que definem como as frases sintáticas são estruturadas usando símbolos terminais e/ou não-terminais:
   - Também chamadas de equações sintáticas, regras de produção ou simplesmente produções;
   - São caracterizadas pelas substituições possíveis dos símbolos não-terminais.

Gramáticas Livres de Contexto

- **Definição Formal:**

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

- G: gramática livre de contexto, uma quádrupla, onde:
  - V: conjunto finito de variáveis;
  - T ou Σ: conjunto finito, disjunto de V, de símbolos terminais;
  - P: conjunto finito de produções;
  - S: $S \in V$ é a variável ou símbolo inicial.

Formato das Regras

As regras têm a seguinte forma:

- Um símbolo de igual "=" separa o lado esquerdo do lado direito;
- O lado esquerdo é formado por um único símbolo não-terminal e, todos os símbolos não-terminais, devem aparecer do lado esquerdo de pelo menos uma regra;
- O lado direito é formado por uma sequência de símbolos terminais, não-terminais e outros símbolos especiais;
- Um ponto "." é usado para finalizar uma regra.

Gramáticas Estendidas (EBNF)

A EBNF permite o uso de símbolos extras no lado direito das regras:

- **|** para alternância (lê-se "ou" ou "ou alternativamente");
- **(** e **)** para agrupamento;
- ***** como um operador pós-fixado para indicar que a expressão sintática pode ser repetida zero ou mais vezes;
- **+** como um operador pós-fixado para indicar que a expressão sintática pode ser repetida uma ou mais vezes;
- **?** como um operador pós-fixado para indicar que a expressão sintática é opcional, ou seja, pode ou não aparecer.

Usando Gramáticas Estendidas

- As gramáticas estendidas permitem que as regras sejam expressas de uma forma simples e mais fácil de serem entendidas:
- Exemplo:

```
identifiers = identifier ( "," identifier )* .
```

Ao invés de:

```
identifiers = identifier | identifier "," identifiers .
```

Algumas Convenções Comuns para as Regras

- Símbolos terminais são apresentados entre aspas ou especificados explicitamente;
- O símbolo inicial está à esquerda da primeira regra;
- O conjunto T consiste em todos os símbolos terminais que aparecem nas regras;
- O conjunto N consiste em todos os símbolos não-terminais que aparecem nas regras;
- Assim como em LFO, toda a especificação da gramática estará contida nas regras se usarmos as convenções acima.

Exemplo: Trecho da Gramática da CPRL

```
program         = declarativePart statementPart "." .
declarativePart = initialDecls subprogramDecls .
initialDecls    = ( initialDecl )* .
initialDecl     = constDecl | arrayTypeDecl | varDecl .
constDecl       = "const" constId ":=" literal ";" .
literal         = intLiteral | charLiteral | stringLiteral | booleanLiteral .
booleanLiteral  = "true" | "false" .
arrayTypeDecl   = "type" typeId "=" "array" "[" intConstValue "]" "of" typeName ";" .
varDecl         = "var" identifiers ":" typeName ";" .
identifiers     = identifier ( "," identifier )* .
typeName        = "Integer" | "Boolean" | "Char" | typeId .
```

Obs: a especificação completa da gramática da linguagem CPRL pode ser encontrada no Apêndice D, fornecido no material da disciplina, cedido gentilmente pelo Professor Moore, autor da obra Introduction to Compiler Design: an Object Oriented Approach Using Java. 2. ed.

Gramáticas Léxicas versus Gramáticas Estruturais

- Gramática Léxica (manipulada pelo *scanner*)
  - Regras simples que expressam símbolos terminais na gramática do parser;
  - Baseada em expressões regulares, sendo possível expressão a sintaxe de cada símbolo em uma única regra;
  - **Exemplo:**

```
identifier = letter ( letter | digit )* .
letter = [A-Za-z] .
digit  = [0-9] .
```

- Gramática Sintática ou Estrutural (manipulada pelo *parser*)
  - Regras mais complexas que descrevem a estrutura da linguagem, sendo possível usar recursão;
  - **Exemplo:**

```
assignmentStmt = variable ":=" expression ";" .
```

Regra para Fim de Arquivo

- As gramáticas normalmente usam uma regra especial para garantir que toda a entrada é correspondida;
- Essa regra pode ser explícita ou simplesmente "aceita";
- **Exemplo:**

```
system = program <EOF> .
```

Onde `<EOF>` representa "fim do arquivo".

Notações Alternativas para as Regras

- Usar "→", "::=", ou simplesmente ":", ao invés de "=" para separar o lado esquerdo e direito das regras;
- Usar o fim de linha, ao invés de ponto, para finalizar as regras;
- Usar "{" e "}" para envolver expressões sintáticas que podem ser repetidas zero ou mais vezes. Similarmente, usar "[" e "]" para envolver expressões sintáticas opcionais;
- Envolver símbolos não-terminais usando "<" e ">" e omitir as aspas duplas em torno de símbolos terminais;
- Usar estilos de fonte diferentes, como negrito, para distinguir entre símbolos terminais e não-terminais.

Alguns Exemplos de Gramáticas

- **Exemplo 1:** A regra de uma gramática estendida:

```
initialDecl = constDecl | arrayTypeDecl | varDecl .
```

- Corresponde a três regras em uma gramática simples:

```
initialDecl = constDecl .
initialDecl = arrayTypeDecl .
initialDecl = varDecl .
```

- **Exemplo 2:** A regra de uma gramática estendida:

```
identifiers = identifier ( "," identifier )* .
```

- Corresponde a três regras em uma gramática simples:

```
identifiers = identifier identifiersTail .
identifiersTail = "," identifiers .
identifiersTail = ε .
```

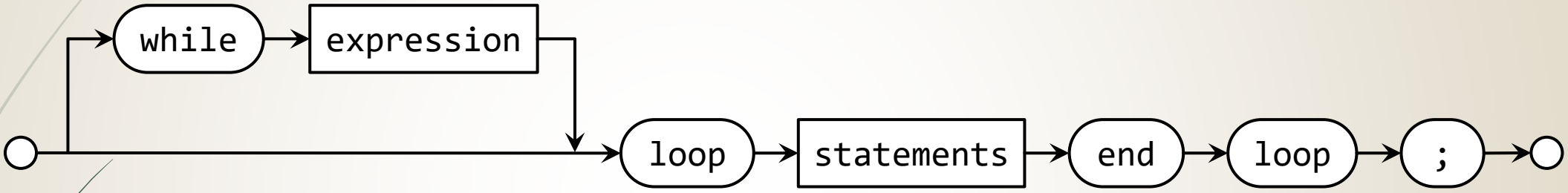
Na obra que adotamos, o autor utiliza a letra grega lambda (λ) como símbolo para indicar o caractere ou a string vazia.

Diagramas Sintáticos

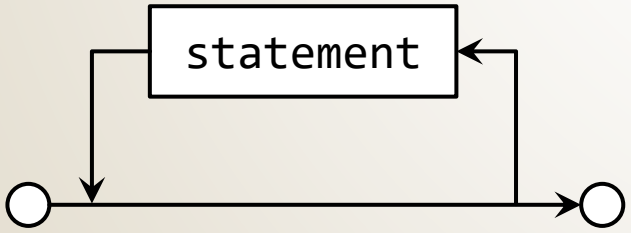
- Os diagramas sintáticos, também conhecidos como diagramas de ferrovia, fornecem uma alternativa gráfica à representação textual para as gramáticas;
  - Representações textuais são mais facilmente processadas pelas ferramentas para compiladores.
  - Os diagramas sintáticos são mais fáceis de entender por humanos;
- Ideia básica:
  - Grafo direcionado;
  - Todo diagrama tem um ponto de entrada e um ponto final;
  - O diagrama descreve os caminhos possíveis entre esses dois pontos passando através de outros não-terminais e terminais;
  - Terminais são representados por retângulos com cantos arredondados;
  - Não-terminais são representados por retângulos.

Diagramas Sintáticos
Exemplos

- `loopStmt`



- `statements`



Convenções de Nomenclatura para Símbolos Não-Terminais

- Muitos projetistas de linguagens de programação adotam convenções de nomenclatura para símbolos não-terminais de modo a comunicar informações contextuais ou semânticas dentro da gramática;
- **Exemplo 1:**

```
functionCall = funcId ( actualParameters )? .
funcId = identifier .
```

Equivalente à:

```
functionCall = identifier ( actualParameters )? .
```

- **Exemplo 2:**

```
loopStmt = ( "while" booleanExpr )? "loop" statements "end" "loop" ";" .
booleanExpr = expression .
```

Transformações em Gramáticas: Substituição de Símbolos Não-Terminais

- Suponha que temos uma regra da seguinte forma:

```
N = X .
```

onde a regra não é recursiva e é a única regra para o não-terminal N.

- Podemos substituir todas as ocorrências de N por X na gramática, eliminando o não-terminal N;
- Dependendo a situação, por exemplo, para auxiliar na legibilidade da gramática, o projetista pode decidir deixar uma regra desse tipo na gramática:

```
booleanExpr = expression .
```

Transformações em Gramáticas: Fatoração à Esquerda

- Suponha que uma regra possua alternativas na forma:

X Y | X Z

- Podemos substituir essas alternativas com a seguinte expressão equivalente:

X (Y | Z)

Transformações em Gramáticas : Eliminação de Recursão à Esquerda

- Suponha que haja uma regra da seguinte forma:

```
N = X | N Y .
```

Onde X e Y são expressões arbitrárias.

- Uma regra dessa forma é chamada de recursiva à esquerda.
- Podemos reescrever essa regra de modo a obter uma regra equivalente que não seja recursiva à esquerda:

```
N = X ( Y )* .
```

Exemplo: Transformações em Gramáticas

- Gramática original:

```
identifier = letter
           | identifier letter
           | identifier digit .
```

- Fatoração à esquerda:

```
identifier = letter
           | identifier ( letter | digit ) .
```

- Eliminação de recursão à esquerda:

```
identifier = letter ( letter | digit )* .
```

Gramáticas versus Linguagens

- Qual a diferença entre uma gramática e uma linguagem?
- Gramáticas diferentes podem gerar (definir) a mesma linguagem (já vimos isso, não é?).

Derivação

- Aplicar as regras sistematicamente, uma por vez, começando pelo símbolo de início;
- Gramática:

```
expr = expr op expr  |  id  |  intLit .
op = "+"  |  "*" .
```

Mostre que "2 + 3 * x" é válido

- Derivação mais à esquerda:

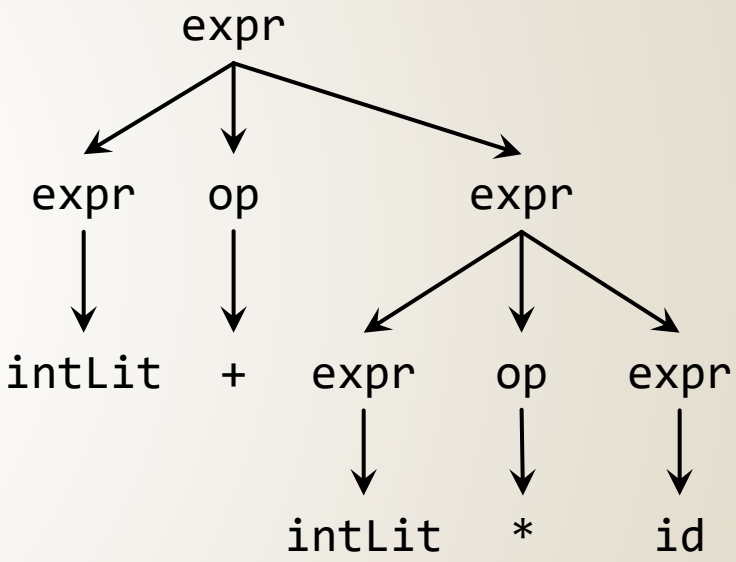
```
expr => expr op expr
     => intLit op expr
     => intLit + expr
     => intLit + expr op expr
     => intLit + intLit op expr
     => intLit + intLit * expr
     => intLit + intLit * id
```

Árvores de Análise Sintática

- Uma **árvore de análise sintática** de uma gramática G é uma árvore rotulada que:
  - As folhas (nós folha/terminais) são rotuladas por símbolos terminais;
  - Os nós internos (nós não-terminais) são rotulados por símbolos não-terminais;
  - O(s) filho(s) de um nó interno N corresponde(m) a uma regra de N;
- Uma árvore de análise sintática provê uma representação gráfica de uma derivação.

Exemplo: Árvore de Análise Sintática

```
expr => expr op expr
     => intLit op expr
     => intLit + expr
     => intLit + expr op expr
     => intLit + intLit op expr
     => intLit + intLit * expr
     => intLit + intLit * id
```



Algumas Gramáticas têm Problemas...

- Usando a mesma gramática, podemos realizar outra derivação à esquerda para "2 + 3 * x" ao escolher uma alternativa diferente da regra `expr` no início:

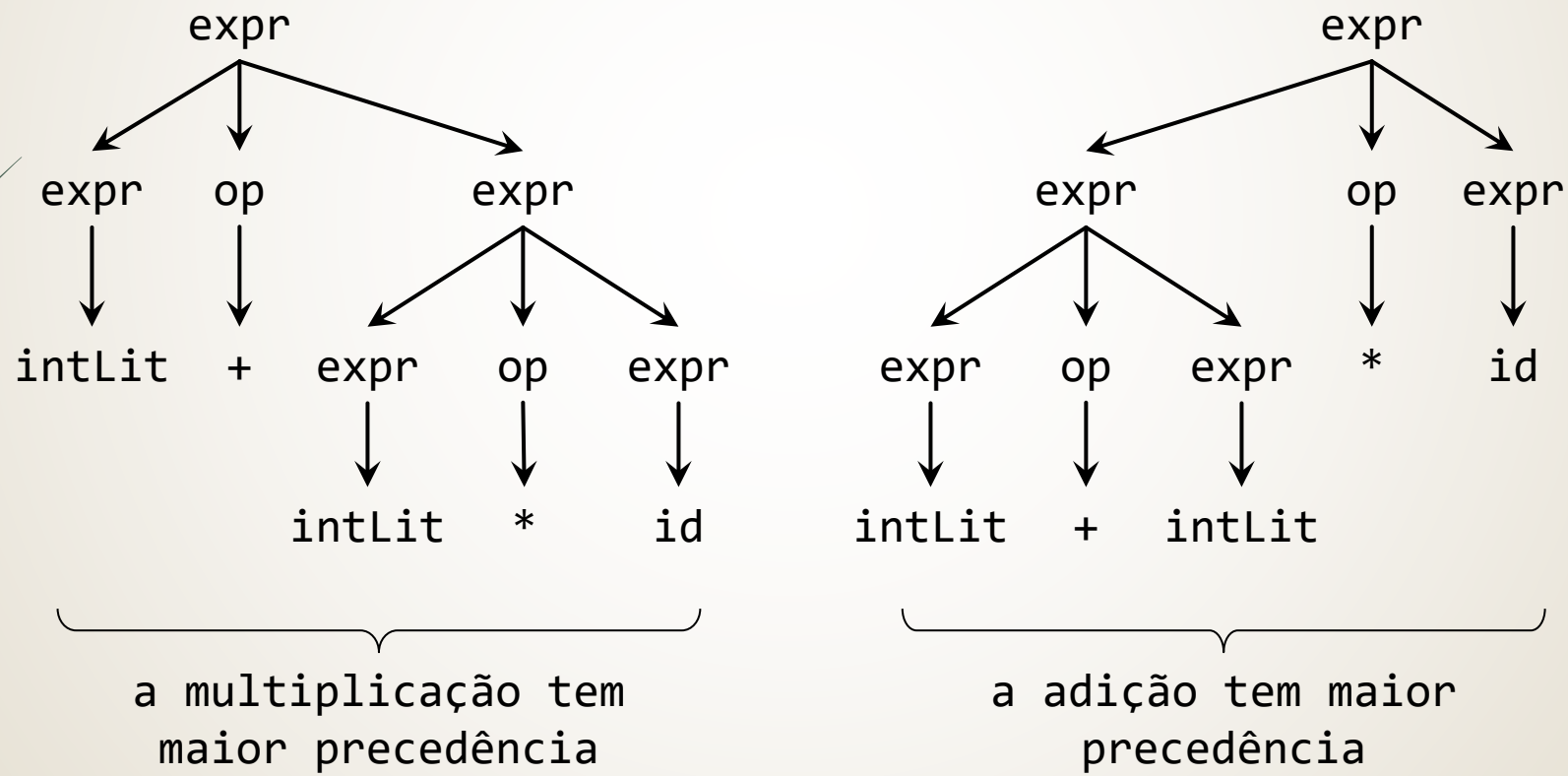
```
expr = expr op expr  |  id  |  intLit .
op = "+"  |  "*" .
```

- Derivação mais à esquerda:

```
expr => expr op expr
     => expr op expr op expr
     => intLit op expr op expr
     => intLit + expr op expr
     => intLit + intLit op expr
     => intLit + intLit * expr
     => intLit + intLit * id
```

Algumas Gramáticas têm Problemas...

Considerando as árvores de análise sintática da primeira e da segunda derivação:



Algumas Gramáticas têm Problemas...

- **Gramática Ambígua!**
  - Uma gramática é ambígua quando, para uma sentença sintaticamente válida, existe mais de uma árvore de análise sintática.

Especificando a Precedência de Operadores

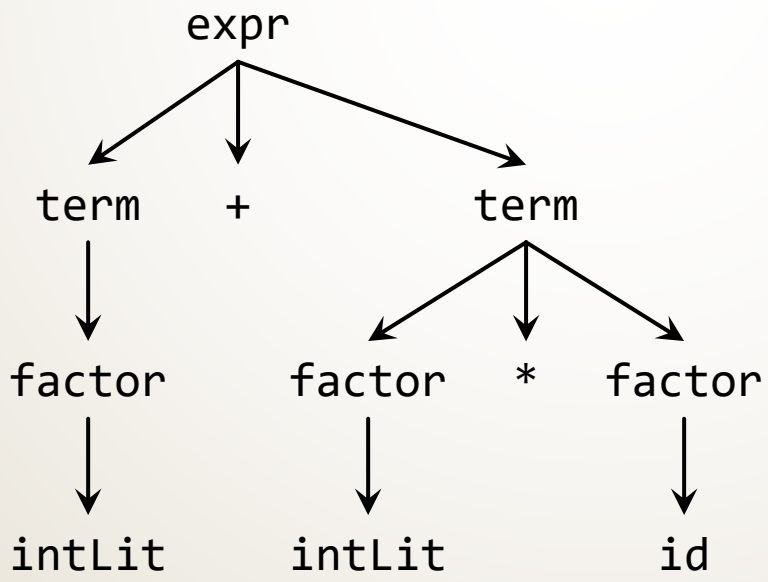
- A precedência dos operadores refere-se à prioridade relativa dos mesmos;
- Há duas abordagens para especificar a precedência de operadores:
  - Dentro da gramática;
  - Usar um mecanismo de especificação adicional (uma tabela), separado da gramática;
- Na definição da linguagem CPRL a primeira abordagem é utilizada;
- A segunda abordagem é suportada por algumas ferramentas como o Yacc.

Especificando a Precedência de Operadores Dentro da Gramática

- Gramática:

```
expr = term ( "+" term )* .
term = factor ( "*" factor )* .
factor = id | intLit
```

- Árvore de análise sintática para a expressão "`2 + 3 * x`"



Note que agora, o operador * tem precedência maior que o operador + na gramática.

Associatividade

- Especifica a ordem de avaliação dos operadores com a mesma precedência quando não houver parênteses;
- **Exemplo 1:** os operadores + e – na CPRL estão no mesmo nível de precedência e são associativos à esquerda:

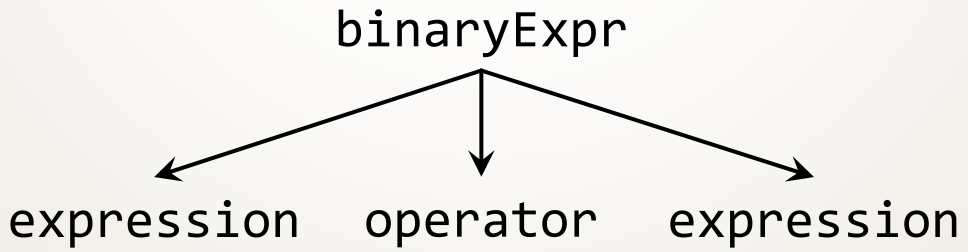
  `8 - 3 + 2` é avaliado como `(8 - 3) + 2`

- **Exemplo 2:** Algumas linguagens, não sendo o caso da CPRL, usam o símbolo ^ como operador de exponenciação e a exponenciação é usualmente definida como sendo associativa à direita:

  `2^2^3` é avaliado como as `2^(2^3)`

Árvores de Análise Sintática Abstratas

- São similares às árvores sintáticas, mas sem os símbolos terminais e não-terminais;
- **Exemplo 1:** Para expressões, podemos omitir todos os não-terminais adicionais que foram introduzidos para definir precedência (`relation`, `simpleExpr`, `term`, `factor` etc). Todas as expressões binárias reteriam apenas o operador e os operandos esquerdo e direito.

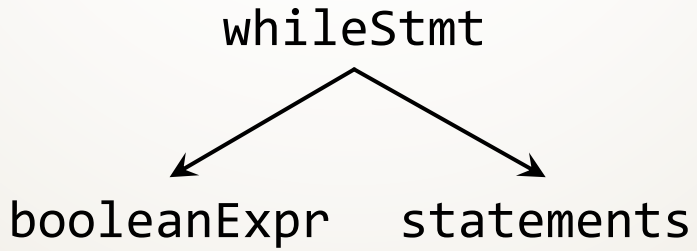


Árvores de Análise Sintática Abstratas

- **Exemplo 2:** Considere a seguinte gramática para uma expressão `while`:

```
whileStmt = "while" booleanExpr "loop" statements "end" "loop" ";" .
```

- Uma vez que a expressão `while` é analisada, não precisamos reter os símbolos terminais. A árvore sintática abstrata para a expressão `while` conteria apenas `booleanExpr` e `statements`.



Uma Gramática Livre de Contexto para Definir Gramáticas Livres de Contexto

```
grammar      = ( rule )+ .
rule         = identifier "=" syntaxExpr "." .
syntaxExpr   = syntaxTerm ( "|" syntaxTerm )* .
syntaxTerm   = ( syntaxFactor )+ .
syntaxFactor = identifier | terminalSym | "(" syntaxExpr ")" ( multChar )? .
multChar     = "*" | "+" | "?" .
identifier   = letter ( letter | digit )* .
terminalSym  = "\"" ( terminalChar | escapedChar )* "\"" .
terminalChar = [ !#-\[\]-~] .
// qualquer caractere ASCII com exceção de contrabarra e aspas duplas
escapedChar  = "\\" ("\"" | "\\") .
letter       = [A-Za-z] .
digit        = [0-9] .
```

Bibliografia

MOORE JR., J. I. **Introduction to Compiler Design: an Object Oriented Approach Using Java**. 2. ed. [s.l.]:SoftMoore Consulting, 2020. 284 p.

AHO, A. V.; LAM, M. S.; SETHI, R. ULLMAN, J. D. **Compiladores: Princípios, Técnicas e Ferramentas**. 2. ed. São Paulo: Pearson, 2008. 634 p.

COOPER, K. D.; TORCZON, L. **Construindo Compiladores**. 2. ed. Rio de Janeiro: Campus Elsevier, 2014. 656 p.

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo: Elsevier, 2016. 307 p.

SANTOS, P. R.; LANGOLOIS, T. **Compiladores: da teoria à prática**. Rio de Janeiro: LTC, 2018. 341 p.