# Atividade 05 - Expressões Constantes 1 - Análise Sintática

Andrei de Araújo Formiga

16 de julho de 2025

Continuando o processo de análise da linguagem EC1 (Expressões Constantes 1), nesta atividade vamos fazer a análise sintática.

A análise sintática determina a estrutura sintática (gramatical) do programa de entrada, usando a sequência de *tokens* resultante da análise léxica. A saída da análise sintática pode ter várias formas (em compiladores muito simples, pode ser até o código objeto final), mas nos projetos da disciplina, o analisador sintático deve produzir uma árvore de sintaxe abstrata (que será explicada adiante).

## 1 A linguagem EC1 (Expressões Constantes 1)

A linguagem EC1 é a mesma já usada na Atividade 04.

Um programa na linguagem EC1 é uma expressão aritmética com operandos constantes e usando as quatro operações. Todas as operações devem ser escritas entre parênteses, então não vamos nos preocupar com precedência de operadores.

Alguns exemplos de programas na linguagem EC1:

```
333
(6 * 7)
(3 + (4 + (11 + 7)))
(33 + (912 * 11))
((427 / 7) + (11 * (231 + 5)))
```

A gramática para a linguagem EC1 é:

```
<programa> ::= <expressao>
<expressao> ::= <literal-inteiro> | (<expressao> <operador> <expressao>)
<operador> ::= + | - | * | /
<literal-inteiro> ::= <digito>+
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Esta gramática mistura regras da *micro-sintaxe* léxica (as regras para `literal-inteiro` e `digito`) com regras para a sintaxe em si, que são as regras para `programa`, `expressao` e `operador`.

A variável (ou *não-terminal*) `programa` para essa linguagem contém apenas uma `expressao`, então só precisamos usar as regras para `expressao` e `operador`.

# 2 Análise sintática descendente recursiva

Algoritmos para fazer análise sintática são estudados desde o início da computação, o que resultou em uma grande variedade de métodos desenvolvidos para a tarefa. Alguns livros sobre compiladores dedicam centenas de páginas ao estudo da análise sintática e seus métodos.

Um dos resultados da pesquisa em análise sintática foi a criação de ferramentas chamadas *geradores de analisadores sintáticos* (*parser generators* em inglês). Esses geradores recebem uma gramática na entrada (como a gramática para a linguagem EC1 mostrada acima) e geram código (em alguma linguagem de programação) que realiza a análise sintática da entrada.

Mesmo com a existência dessas ferramentas, muitos compiladores usados hoje em dia ainda usam analisadores sintáticos feitos *à mão*, sem o uso de geradores. Existem vários fatores para isso, um dos principais sendo que é mais fácil gerar boas mensagens de erro para o programador que está usando o compilador.

Para analisadores criados diretamente, sem o uso de geradores, um dos algoritmos de análise sintática mais simples e mais utilizados é a *análise descendente recursiva*.

Na análise descendente recursiva, criamos uma função para cada não-terminal que queremos analisar. No caso da gramática para a linguagem EC1, teremos duas funções, para analisar `expressao` e `operador`. Quando um não-terminal possui mais de uma regra, deve ser possível determinar qual regra deverá ser seguida olhando apenas para o próximo *token* na entrada.

O não-terminal `expressao` tem duas regras:

```
<expressao> ::= <literal-inteiro> | (<expressao> <operador> <expressao>)
```

Ao analisar uma expressão, se o próximo *token* na entrada for um literal inteiro, a primeira regra deve ser utilizada; se o próximo *token* for um parêntese abrindo, a segunda regra deve ser utilizada. Se o próximo *token* não for nem um parêntese abrindo nem um literal inteiro, o analisador encontrou um erro sintático. Portanto, a função de analisar expressões terá a seguinte forma (em pseudo-código):

```
analisaExp():
  tok = proximo_token()
  if tok.tipo == LITERAL_INTEIRO:
    # analisa constante inteira
  else if tok.tipo == ABRE_PARENTESE:
    # analisa constante inteira
```

```
  else:
    # sinaliza erro sintatico
```

Entre as duas possibilidades válidas, o caso do literal inteiro é o mais simples. Como o próximo *token* é um literal, não é preciso analisar mais nada, isso já é uma expressão completa. Se a análise sintática fosse apenas um processo de verificar se o programa possui um erro sintático ou não, bastaria retornar um valor verdadeiro nesse caso.

Mas o papel mais importante da análise sintática é determinar a estrutura do código de entrada. Nesse caso, o que a função acima deve retornar? Para capturar a estrutura do programa de entrada, nosso compilador vai retornar uma árvore sintática que será utilizada pelas etapas seguintes.

# 3 Árvore de sintaxe abstrata

Como já mencionado, a análise sintática produz na saída alguma representação intermediária do código de entrada. Uma forma comum de representação intermediária são as *árvores de sintaxe abstrata*, geralmente chamadas apenas de *árvores sintáticas*.

Uma árvore sintática é uma árvore composta por nós que descrevem as estruturas do programa de entrada; esses nós podem possuir filhos na árvore que representam sub-componentes da estrutura representada pelo nó pai.

Por exemplo, um programa na linguagem EC1 que é apenas uma constante inteira vai ser representado por uma árvore que é apenas um nó de constante inteira:

```
# codigo
42

# arvore
42
```

Se o programa for uma multiplicação, como `(7 * 6)`, a árvore seria um nó para a operação binária de multiplicação, e dois componentes que são os operandos da multiplicação:

```
# codigo
(7 * 6)

# arvore
  *
 / \
7   6
```

Vejamos a árvore para uma expressão um pouco mais complicada:

```
# codigo
(33 + (912 * 11))

# arvore
  +
 / \
33  *
   / \
 912  11
```

Note que a estrutura da árvore captura a estrutura da expressão: a expressão é uma soma tendo como operando esquerdo a constante 33, e como operando direito uma multiplicação (entre 912 e 11). Para realizar a soma, é preciso saber o valor de ambos operandos, o que significa que a multiplicação deve ser efetuada antes da soma. Isso é o que esperamos ao ver a expressão (33 + (912 * 11)).

# 4 Representação da árvore

A árvore sintática pode ser representada no programa do compilador de várias formas. Em linguagens orientadas a objetos, é comum usar uma hierarquia de classes para representar a árvore.

Para a linguagem EC1, podemos definir uma classe-base `Exp` que representa expressões (e nós da árvore) em geral. Desta classe derivamos duas sub-classes: `Const` para uma constante inteira, e `OpBin` para uma operação binária. A classe `Const` precisa ter apenas um campo para o valor inteiro da constante, enquanto que `OpBin` precisa ter campos para o operador, e referências para os operandos esquerdo e direito (que são objetos da classe `Exp`). A classe `Exp` pode ser abstrata, já que nenhum objeto é criado diretamente com essa classe.

Essas classes poderiam ser definidas da seguinte forma (em pseudo-código similar a Java):

```
abstract class Exp { }

class Const : Exp {
  int   valor;
}

class OpBin : Exp {
  op    operador;
  Exp   opEsq;
  Exp   opDir;
}
```

Neste código, não definimos o tipo `op` para o operador, mas esse campo apenas precisa distinguir entre as quatro operações possíveis: soma, subtração, multiplicação e divisão. Podemos usar uma enumeração ou um conjunto de constantes inteiras para isso.

# 5 Análise das expressões em EC1

Agora que sabemos como representar a árvore sintática, vamos finalizar a função de análise sintática de expressões. Se o próximo *token* for um literal inteiro, a função de análise deve criar um nó do tipo constante com o valor inteiro do literal. No código abaixo, usamos uma função chamada `inteiro` para converter da string do lexema para um valor inteiro:

```
analisaExp():
  tok = proximo_token()
  if tok.tipo == LITERAL_INTEIRO:
    return new Const(inteiro(tok.lexema))
  else if tok.tipo == ABRE_PARENTESE:
    # analisa constante inteira
  else:
    # sinaliza erro sintatico
```

Para o caso do próximo *token* ser um parêntese abrindo, a análise deve seguir a produção para operações binárias:

```
<expressao> ::= (<expressao> <operador> <expressao>)
```

Ou seja, após o parêntese deve aparecer uma nova expressão (que é o operando esquerdo da operação), seguida de um operador e de uma outra expressão que é o operando direito da operação.

Após determinar que o próximo *token* é um parêntese abrindo, a análise deve reconhecer uma expressão. Como analisar uma expressão? Chamando a própria função `analisaExp` que estamos escrevendo (por isso o nome dessa técnica é análise descendente *recursiva*). Precisamos guardar o resultado dessa análise em alguma variável, depois reconhecer o operador, depois a outra expressão. Por fim, após a expressão do operando direito, é necessário que o *token* seguinte seja um parêntese fechando para concluir a expressão, caso contrário é um erro. O pseudo-código correspondente à essa descrição é:

```
analisaExp():
  tok = proximo_token()
  if tok.tipo == LITERAL_INTEIRO:
    return new Const(inteiro(tok.lexema))
  else if tok.tipo == ABRE_PARENTESE:
```

```
    opEsq = analisaExp()
    operador = analisaOperador()
    opDir = analisaExp()
    verificaProxToken(FECHA_PARENTESE)
    return new OpBin(operador, opEsq, opDir)
  else:
    # sinaliza erro sintatico
```

A função `verificaProxToken` verifica se o próximo *token* tem o tipo necessário (`FECHA_PARENTESE`, no caso) e sinaliza um erro caso não seja deste tipo. Se a verificação for bem sucedida, a análise continua retornando um nó que representa uma operação binária, juntando as três informações necessárias (os operandos e o operador).

A função `analisaOperador` é bem simples: verifica se o próximo *token* é um dos quatro operadores e cria um valor para representar o operador adequado; se não for um dos quatro operadores reconhecidos, é um erro sintático.

As funções `analisaExp` e `analisaOperador` fazem toda a análise sintática para a linguagem EC1.

# 6 A interface entre o analisador léxico e o sintático

Um detalhe importante nas funções de análise sintática é a interação entre o analisador sintático e o analisador léxico. Como vimos no pseudo-código acima, as funções de análise sintática acessam os *tokens* criados pela análise léxica em sequência, usando a função `proximo_token`. Essa função retorna o próximo *token* na entrada e avança a leitura da entrada para o *token* seguinte, até chegar ao fim da entrada.

Como discutido na atividade anterior, o analisador léxico pode ser organizado para retornar um *token* por vez (através da função `proximo_token`) ou pode varrer a entrada inteira e retornar uma coleção com todos os *tokens* de uma vez. Nesse segundo caso, caso a implementação não tenha uma função `proximo_token`, é fácil criar esta função a partir da coleção completa de todos os *tokens* na entrada; para isso é só lembrar qual o índice do próximo *token* na coleção; toda vez que `proximo_token` é chamada, o *token* atual da coleção é retornado, e o índice do *token* atual é incrementado.

Outra capacidade que o analisador sintático precisa é verificar se a entrada acabou. Isso pode ser feito com a função `proximo_token` retornando um *token* especial do tipo EOF (final da entrada), ou o analisador léxico pode disponibilizar uma função que retorna um valor booleano se a entrada chegou no fim ou não (`final_da_entrada_`). Nesse último caso, o analisador sintático precisa verificar, antes de cada chamada a `proximo_token`, se a saída está no final ou não.

# 7 Um interpretador

A árvore sintática possui todas as informações necessárias para executar o programa EC1 (ou seja, a expressão) que foi analisado.

Executar o programa de entrada sem traduzi-lo para outra linguagem é um processo chamado de *interpretação* (ao invés de compilação). Uma forma simples de interpretador é o interpretador de varredura de árvore (*tree-walking interpreter*), que executa diretamente o programa a partir da árvore sintática.

Para um programa EC1 cuja árvore sintática é conhecida, o interpretador é um processo recursivo simples:

- Se o nó é uma constante, o valor do nó é o valor da constante
- Se o nó é uma operação binária, obtenha o valor do operando esquerdo e o valor do operando direito; o valor do nó é o valor do operando esquerdo operado com o valor do operando direito segundo o operador especificado no nó.

Se a árvore é representada com objetos de uma hierarquia de classes como sugerido acima, uma forma de implementar o interpretador é definir um método na classe base `Exp` e implementar o método nas sub-classes de maneira adequada.

# 8 Impressão da árvore

Um outro processo de varredura da árvore, e que é útil para testes, é imprimi-la. Para a linguagem EC1 podemos obter uma string similar à do programa de entrada usando um processo de impressão simples:

- Se o nó for uma constante, imprimir a constante
- Se o nó for uma operação binária, imprimir em sequência: parêntese `(`,operando esquerdo, operador direito e parêntese `)`

Uma outra possibilidade é gerar um formato gráfico que possa ser transformado em imagem. Uma forma simples seria especificar a árvore como um grafo para a ferramenta graphviz.

# 9 Artefato para entrega

O grupo deve entregar um programa que faz análise léxica e sintática do programa de entrada (usando o analisador léxico da atividade anterior), produz a árvore sintática do programa de entrada e obtém o valor do programa através de interpretação por varredura da árvore.

O projeto deve incluir um conjunto de testes que verifica tanto a produção correta da árvore sintática para um conjunto de programas, como o valor do programa obtido pela interpretação. Os testes também devem incluir exemplos de programas com erros de sintaxe, e o compilador deve ser capaz de detectar e reportar esses erros.

O projeto também deve incluir documentação de uso do analisador e como executar os testes.