`minc`
Exemplo de um compilador/interpretador de expressões mínimo

Marco A.F.S.

2 de outubro de 2019

1 Introdução

O objetivo deste documento é apresentar o projeto e a implementação do compilador/interpretador `minc` para uma linguagem mínima que gera expressões aritméticas, `min`.

2 Gramática de `min`

A gramática da linguagem `min` está descrita a seguir, por meio da linguagem BNF, EBNF e de um diagrama sintático (somente para comparação de metalinguagens):

- **BNF**:

  ⟨*expression*⟩ ::= ⟨*expression*⟩ '+' ⟨*term*⟩
  | ⟨*expression*⟩ '-' ⟨*term*⟩
  | ⟨*term*⟩

  ⟨*term*⟩ ::= ⟨*term*⟩ '*' ⟨*factor*⟩
  | ⟨*term*⟩ '/' ⟨*factor*⟩
  | ⟨*factor*⟩

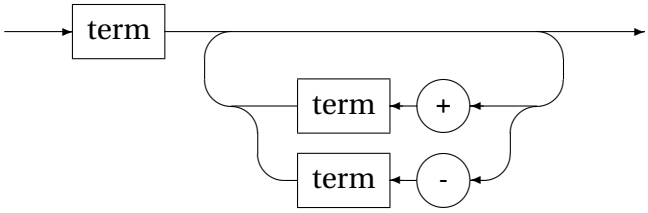
  ⟨*factor*⟩ ::= `digit`
  | '('⟨*expresion*⟩')'

- **EBNF**:

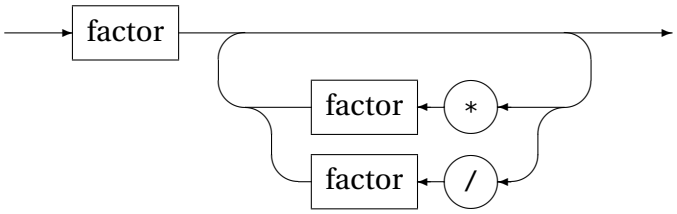
```
expression = term, {('+'|'-'), term};
term = factor, {('*'|'/'), factor};
factor = digit | '(', expression, ')';
```

- **Diagramas sintáticos**:

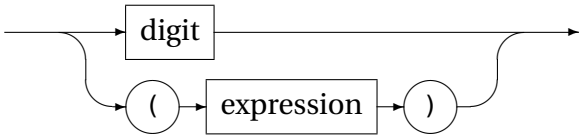
  expression



term



factor



3 Requisitos para o `minc`

3.1 Analisador léxico

O analisador léxico para a linguagem `min` deverá reconhecer e classificar as marcas (*tokens*) a seguir:

- **Dígitos**: símbolos '0' a '9';
- **Parênteses**: símbolos '(' e ')';
- **Operadores**: símbolos '+', '-', '*' e '/'.

Para outras marcas será utilizada uma classe especial que posteriormente será utilizada pelo analisador sintático para sinalizar um erro.

3.2 Analisador sintático

Para o analisador sintático será utilizada uma técnica denominada ***análise sintática recursiva descendente***. Neste tipo de análise, uma **regra gramatical** definida para um **símbolo** não terminal *A*, é escrita como um de **procedimento** para **reconhecer** esse símbolo. Normalmente para a **criação** do **analisador sintático** se emprega a **metalinguagem EBNF**, pois nela **eliminam-se** as **recursões infinitas** e **não-determinismo** na construção do analisador.

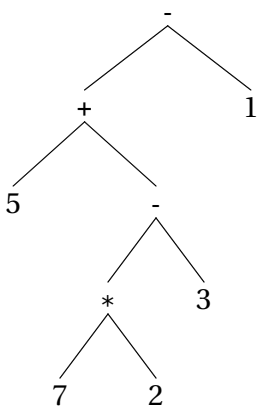
De acordo com a gramática, é comum que as chamadas dos procedimentos elaborados gerem uma cadeia de chamada recursiva (daí o nome recursivo no método).

Por exemplo, na implementação do analisador sintático do `minc` serão escritos métodos para reconhecer os elementos `expression`, `term`, `factor`. Não é difícil perceber que existirá uma chamada recursiva a partir de `expression`

3.3 Árvore de sintaxe abstrata

O analisador sintático retornará, caso a entrada esteja de acordo com a gramática, uma **árvore de sintaxe abstrata** (ou AST – *abstract syntax tree*). Árvores de sintaxe simplificam **a estrutura sintática**.

Para **transformar** uma **árvore de análise sintática** em uma **árvore sintática abstrata**, basta **mover** os símbolos **terminais** representando as **operações** e **comandos** para os **nós-raiz** das **subárvores**, deixando os **operandos** como seus nós filhos.Por exemplo, a expressão `5 + (7 * 2 - 3) - 1`, produz a AST:



Notar a precedência dos operadores e dos parênteses.

3.4 Interpretador

Com a AST produzida pelo analisador sintático, pode-se proceder a uma **interpretação** da mesma, gerando-se um valor resultante. Neste processo, a semântica das operações é dada pelo símbolo dos operadores armazenados nas raízes das subárvores e por um percorrimento de árvore do tipo **pós-ordem**:

- Primeiro processa-se a subárvore esquerda em pós-ordem;
- Depois processa-se a subárvore direita em pós-ordem;
- Finalmente, processa-se a raiz (nós-folha não possuem subárvores – eles terminam o processo recursivo).

Desse modo, obtém-se os valores das subárvores e aplica-se o operador correspondente, gerando um resultado que será considerado posteriormente.

3.5 Compilador

O compilador de `minc` é bem simples: ele gera instruções para uma **máquina de pilha** primitiva que, para a gramática apresentada, necessitará apenas das operações: `PUSH`, para inserir uma constante na pilha e as operações aritméticas `ADD`, `SUB`, `MUL`, `DIV`, que se aplicam a dois operandos que estão no topo da pilha e cujo resultado irá para o topo da pilha, também. Por exemplo, a expressão: `3+5/(2-(4+7*2)-3)+2/(3/(7+1))` é compilada em:

```
PUSH 3
PUSH 5
PUSH 2
PUSH 4
PUSH 7
PUSH 2
MULT
ADD
SUB
PUSH 3
SUB
DIV
```

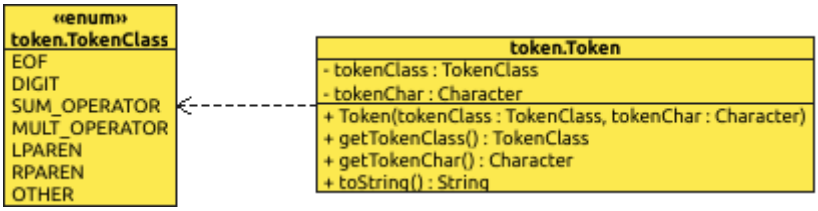
```
ADD
PUSH 2
PUSH 3
PUSH 7
PUSH 1
ADD
DIV
DIV
ADD
```

4 Projeto orientado a objetos do `minc`

Para a implementação do `minc` será utilizada a linguagem de programação Java. Aqui se apresenta uma visão geral das classes utilizando a linguagem UML (será estudada em Engenharia de Software), organizadas em pacotes da aplicação.

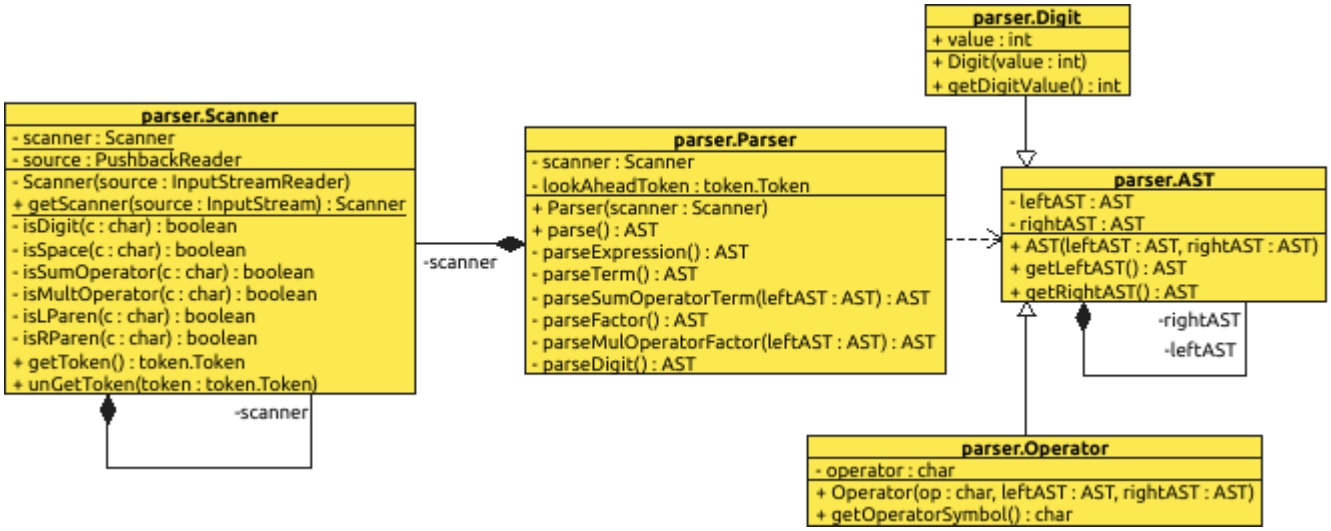
4.1 Pacote `token`

Possui as classes responsáveis por classificar uma marca reconhecida pelo analisador léxico.



4.2 Pacote `parser`

Possui as classes responsáveis por criar o analisador léxico e sintático, além da árvore de sintaxe abstrata.

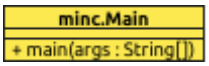


4.3 Pacote `vm`

Possui as classes responsáveis por implementar um interpretador e um compilador que operam sobre árvore AST.

4.4 Pacote `minc`

Possui a classe principal que é o ponto de partida do programa, processando as opções de linha de comando e invocando ou o interpretador ou o compilador.



A descrição de cada classe pode ser encontrada na documentação gerada pelo JavaDoc (veja a seção 5).

5 Implementação em Java

O código Java de `minc` está disponibilizado no GitHub. Para clonar:

```
git clone https://github.com/mafsouza/minc-nb.git
```

Este projeto pode ser aberto pelo NetBeans. A documentação, em HTML, está na pasta `dist/javadoc` do projeto (abrir `index.html`). Já existe um arquivo JAR para ser executado dentro da pasta `dist`.

6 Modo de usar

Compilar o projeto em um arquivo JAR. Na linha de comando, executar assim:

- Para o interpretador:

```
    java -jar minc.jar -i < nome.minc
```

- Para o compilador:

```
    java -jar minc.jar -c < nome.minc
```

Onde `nome.minc` é um arquivo texto contendo uma única expressão que segue a gramática de `min`. No caso do compilador será criado um arquivo com nome `output.out` contendo o código gerado.

Referências

GRUNE, D. et al. **Modern Compiler Design**. Chichester, UK: John Wiley, 2000.