Universidade de Brasília

Estrutura de Dados - Turma B - Trabalho I

Avaliação de Expressões Aritméticas Forma Posfixa

CIC - Departamento de Ciência da Computação

Autores:
Tiago L. P. de Pádua - 12/1042457
Ronaldo S. Ferreira Jr. - 09/48721
Alex Leite - 05/97694

Professor:
Eduardo A. P. Alchieri

Dezembro de 2012

1 Introdução

Através do estudo de estrututa de dados, este trabalho tem o objetivo a confecção de um projeto, onde deverá ser feito um *software* capaz de de validar, avaliar e realizar a conversão de uma expressão aritmética escrita em forma infixa para a forma posfixa.

Dada uma expressão aritmética qualquer, inicialmente iremos realizar sua validação, verificando se todos os parênteses abertos na expressão foram devidamente fechados, ou seja, procurando por leves inconsistencias na expressão aritmética. A seguir, iremos converter a expressão de sua forma infixa para a forma posfixa, conforme exemplo abaixo:

Exemplo I:

Forma infixa: $A * B + C - (D/E + F)$

Forma posfixa: $AB * C + DE/F + -$

Exemplo II:

Forma infixa: $10 * 5 + 6 - (8/2 + 7)$

Forma posfixa: $10\ 5 * 6 + 82/7 + -$

Após obtermos a forma posfixa da expressão iremos realizar sua avaliação, isto é, efetuar o cálculo da expressão e exibir seu resultado, caso a entrada de parâmetros seja numérica.

2 Implementação

A implementação do código foi realizada utilizando a linguagem de programação Java (javac 1.6.0_37) através da plataforma de desenvolvimento Netbeans 7.0.1.

As seguintes classes foram criadas:

- Elemento.java
- Pilha.java
- PilhaVaziaException.java
- ExpressaoAritmetica.java
- ExpressaoGUI.java
- ProcessadorException.java

2.1 Elemento.java

Classe que serve de base para implementação de uma Pilha, possui somente os campos valor:Object e proximo:Elemento e seus métodos auxiliares. Como "valor"é do tipo Object, esta implamentação de pilha é genérica e servirá a qualquer tipo de dados. A classe Elemento carrega os objetos contidos em uma Pilha.

2.2 Pilha.java

Possui os campos tamanho:int, que contém o tamanho corrente da pilha e, topo:Elemento, onde está indicado o elemento localizado no topo da Pilha. Os métodos implementados são os necessários para a utilização da pilha:

- public void empilhar(Object valor);
  Para empilhar elementos na Pilha (operação *push*).
- public Object desempilhar( ) throws PilhaVaziaException;
  Para desempilhar elementos da Pilha (operação *pop*), caso a Pilha esteja vazia, uma mensagem de erro é lançada ao sistema.
- public int getTamanho( );
  Exibe o tamanho da Pilha.
- public boolean isVazio( );
  Checa se a Pilha está vazia.
- public Object getTopo( ) throws PilhaVaziaException;
  Lê o objeto no topo da Pilha.

2.3 PilhaVaziaException.java

Classe criada para identificar a mensagem de erro lançada, caso tente-se desempilhar uma Pilha vazia.

2.4 ExpressaoAritmetica.java

Classe principal do programa, possui o campo expressao:String que deve ser iniciado com a expressão a ser trabalhada, são implementados os seguintes métodos:

2.4.1 private boolean possuiCaracteresInvalidosNumericos( )

Este método retorna *true*, caso a expressão aritmética numérica informada na classe possua caracteres estranhos à uma expressão aritmética numérica válida, para isso, utilizou-se uma expressão regular.

2.4.2 private boolean possuiCaracteresInvalidosVariaveis( )

Este método retorna *true* caso a expressão algébrica sob forma de variáveis ($A+B*C$) informada na classe possua caracteres estranhos à uma expressão válida, para isso, utilizou-se uma expressão regular.

2.4.3 public int avaliar( ) throws ProcessadorException

Dada uma expressão aritmética numérida válida, este método retorna o valor resultante da expressão, *representa o requisito 3.4 do trabalho.*

2.4.4 public void validarVariavel( ) throws ProcessadorException

Valida uma expressão algébrica formada por variáveis simbólicas.

2.4.5 public void validarNumerico( ) throws ProcessadorException

Valida uma expressão aritmética formada por valores numéricos.

2.4.6 public static int getPrioridade(String operador) throws ProcessadorException

Retorna a prioridade do operador aritmético informado (multiplicação, divisão, soma, subtração).

2.4.7 public static boolean isParenteses(String s)

Retorna *true* caso o caractere informado seja um parêntese.

2.4.8 public static boolean isOperador(String s)

Retorna *true* caso o caractere informado seja um operador aritmético válido (+, -, *, /).

2.4.9 public String getPosfixa(boolean comVariaveis) throws ProcessadorException

Retorna a expressão aritmética dada para sua forma posfixa, *representa o requisito 3.3 do trabalho.*

2.4.10 public String getExpressao( )

Retorna a expressão infixa associada ao objeto.

2.4.11 public void setExpressao(String expressao)

Atribui ao objeto a expressão aritmética infixa a ser trabalhada.

2.4.12 public static boolean isValorVariavel(String s)

Verifica se o caractere apresentado representa uma variável simbólica, ou seja, uma letra do conjunto [A-Z].

2.4.13 public boolean isExpressaoValida( )

Retorna *true* caso os parênteses da expressão dada estejam corretamente aninhados e balanceados, *representa o requisito 3.2 do trabalho.*

2.4.14 public String[ ] getVetorNumerico( )

Transforma a expressão aritmética numérica infixa em um vetor de Strings que representam cada componente da expressão.

2.4.15 public String[ ] getVetorVariavel( )

Transforma a expressão aritmética com variáveis simbólicas infixa em um vetor de Strings que representam cada componente da expressão.

2.4.16 public String[ ] getParenteses( )

Retorna um vetor de Strings contendo todos os parenteses da expressão.

2.4.17 public static String[ ] processaRegex(String regex, String s)

Método utilitário usado para converter uma String em um vetor de Strings utilizando uma expressão regular.

2.5 ExpressaoGUI.java

Classe que representa a interface gráfica (GUI - *Graphical User Interface*), utilizada pelo usuário na execução do programa.

2.6 ProcessadorException.java

Exceção criada para identificar erros que decorrem do processamento da expressão aritmética.

3 Estudo de Complexidade

Um estudo acerca da complexidade dos métodos apresentado foi realizado, como requisitos para o trabalho, são eles:

3.1 public boolean isExpressaoValida()

```
1   public boolean isExpressaoValida() {
        Pilha p = new Pilha();
3       String[] vetor = getParenteses();
        for (String e : vetor) {
5           if ("(".equals(e)) {
                p.empilhar(e);
7           } else {
                try {
9                   p.desempilhar();
                } catch (PilhaVaziaException ex) {
11                  return false;
                }
13          }
        }
15      return p.isVazio();
    }
```

Estudo da complexidade linha a linha:

- Linha 2 - 1

- Linha 3 - 1
- Linha 4 - n
- Linha 5 - n
- Linha 6 - n/2
- Linha 7 - n/2
- Linha 15 - 1

Ao final teremos $3n + 3$, ou seja $O(n) = n$.

Ou seja, a curva $O(n)$ apresenta um crescimento linear, que varia de acordo com n (a quantidade de elementos da pilha).

3.2 public String getPosfixa(boolean comVariaveis) throws ProcessadorException

```
    public String getPosfixa(boolean comVariaveis) throws ProcessadorException {
2       String[] vetor;
        if(comVariaveis){
4           validarVariavel();
            vetor = getVetorVariavel();
6       }else{
            validarNumerico();
8           vetor = getVetorNumerico();
        }
10      String resposta = "";
        Pilha pilha = new Pilha();
12      for (String e : vetor) {
            if (!comVariaveis && isValorNumerico(e)) {
14              resposta += " " + e;
            }
16          if (comVariaveis && isValorVariavel(e)) {
                resposta += " " + e;
18          }
            if (isOperador(e)) {
20              while (!pilha.isVazio() && !isParenteses((String) pilha.getTopo())
                    && !(getPrioridade(e) > getPrioridade((String) pilha.getTopo
                    ()))) {
                    resposta += " " + pilha.desempilhar();
22              }
                pilha.empilhar(e);
24          }
            if (isParenteses(e)) {
26              if ("(".equals(e)) {
                    pilha.empilhar(e);
28              } else if (")".equals(e)) {
                    while (!"(".equals(pilha.getTopo())) {
30                      resposta += " " + pilha.desempilhar();
                    }
```

```
32                      pilha.desempilhar();
                    }
34              }
            }
36          while (!pilha.isVazio()) {
                resposta += " " + pilha.desempilhar();
38          }
            resposta = resposta.trim();
40          return resposta;
        }
```

Estudo da complexidade linha a linha:

- Linhas: 3, 4, 5, 10, 11 - 1
- Linhas: 12, 13, 14, 16, 17, 19, 20, 21, 23, 25,26, 27, 28, 29, 30, 32, 36, 37 - n
- Linhas: 39, 40 - 1

Ao final teremos $18n + 7$, ou seja $O(n) = n$.

Ou seja, a curva $O(n)$ apresenta um crescimento linear, que varia de acordo com n (a quantidade de elementos da pilha).

3.3 public int avaliar() throws ProcessadorException

```
1       public int avaliar() throws ProcessadorException {
            validarNumerico();
3           Pilha p = new Pilha();
            String[] vetor = processaRegex("[0-9]+|.", getPosfixa(false));
5           for(String e : vetor){
                if(" ".equals(e)){
7                   continue;
                }
9               if(isValorNumerico(e)){
                    p.empilhar(e);
11              }
                if(isOperador(e)){
13                  int valor1, valor2, res = 0;
                    valor1 = Integer.parseInt((String)p.desempilhar());
15                  valor2 = Integer.parseInt((String)p.desempilhar());
                    if("+".equals(e)){
17                      res = valor1+valor2;
                    }
19                  if("-".equals(e)){
                        res = valor1-valor2;
21                  }
                    if("*".equals(e)){
23                      res = valor1*valor2;
                    }
25                  if("/".equals(e)){
                        res = valor1/valor2;
27                  }
```

```
                p.empilhar(Integer.toString(res));
29          }
        }
31      if(p.getTamanho()>1){
            throw new ProcessadorException("Erro ao processar expressao: " +
                getExpressao());
33      }
        return Integer.parseInt((String)p.getTopo());
35  }
```

Estudo da complexidade linha a linha:

- Linhas: 2, 3 , 4 - 1;
- Linhas: 5, 6, 7, 9, 10, 12, 13, 14, 15, 16, 17, 19, 20, 22, 23, 25, 26, 28 - n;
- Linhas: 31, 32, 34 - 1;

Ao final teremos $18n + 6$, ou seja $O(n) = n$.

Ou seja, a curva $O(n)$ apresenta um crescimento linear, que varia de acordo com n (a quantidade de elementos da pilha).

4 Listagem de testes executados

Para a realização dos testes, foi criada a classe TesteExpressaoAritmetica.java e foram realizados os testes conforme consta abaixo:

```
1  public class TesteExpressaoAritmetica
       extends TestCase {
3
   public TesteExpressaoAritmetica(String testName) {
5      super(testName);
   }
7
   public static Test suite() {
9      return new TestSuite(TesteExpressaoAritmetica.class);
   }
11
   public void testExpressoes() throws ProcessadorException {
13     //   Infixa                          Posfixa
       //1  (A+B*C)                         A B C * +
15     //2  (A*(B+C)/D-E)                   A B C + * D / E -
       //3  (A+B*(C-D*(E-F)-G*H)-I*3)       A B C D E F - * - G H * - * + I 3 * -
17     //4  (A+B*C/D*E-F)                   A B C * D / E * + F -
       //5  (A+(B-(C+(D-(E+F)))))           A B C D E F + - + - +
19     //6  (A*(B+(C*(D+(E*(F+G))))))       A B C D E F G + * + * + *
       ExpressaoAritmetica e = new ExpressaoAritmetica();
21
       //1
23     e.setExpressao("(A+B*C)");
       assertEquals("A B C * +", e.getPosfixa(true));
25
```

```
        //2
27      e.setExpressao("(A*(B+C)/D-E)");
        assertEquals("A B C + * D / E -", e.getPosfixa(true));
29      e.setExpressao("(100*(200+300)/400-500)");
        assertEquals(-375, e.avaliar());
31
        //3
33      e.setExpressao("(A+B*(C-D*(E-F)-G*H)-I*J)");
        assertEquals("A B C D E F - * - G H * - * + I J * -", e.getPosfixa(true));
35      e.setExpressao("(100+200*(300-400*(500-600)-700*800)-900*1000)");
        assertEquals(-104839900, e.avaliar());
37
        //4
39      e.setExpressao("(A+B*C/D*E-F)");
        assertEquals("A B C * D / E * + F -", e.getPosfixa(true));
41      e.setExpressao("(100+200*300/400*500-600)");
        assertEquals(74500, e.avaliar());
43
        //5
45      e.setExpressao("(A+(B-(C+(D-(E+F)))))");
        assertEquals("A B C D E F + - + - +", e.getPosfixa(true));
47      e.setExpressao("(100+(200-(300+(400-(500+600)))))");
        assertEquals(700, e.avaliar());
49
        //6
51      e.setExpressao("(A*(B+(C*(D+(E*(F+G))))))");
        assertEquals("A B C D E F G + * + * + *", e.getPosfixa(true));
53      e.setExpressao("(1*(2-(3*(4+(5*(6+7))))))");
        assertEquals(-205, e.avaliar());
55
    }
57  }
```

Após a realização dos testes, foi verificado que todos foram bem sucedidos.

5 Conclusão

Pode-se concluir que a utilização de pilhas é recomendável quando da necessidade de algoritmos para processar expressões aritméticas, realizando a conversão de sua forma infixa para posfixa e avaliando seu resultado. Como já ocorre em inúmeras calculadoras gráficas e científicas, tendo em vista, que a forma posfixa, e suas derivadas, de uma expressão separa elemento a elemento de uma maneira mais eficiente computacionalmente.

Referências

[1] Roberto Tamassia Michael T. Goodrich. *Data Structures and Algorithms in Java.* 2005.