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
Alex Leite
Ronaldo S. Ferreira Jr.

Professor:
Eduardo A. P. Alchieri

Dezembro de 2012

Resumo

Este trabalho tem o objetivo de validar, avaliar e transformar da forma infixa para a forma posfixa expressões aritméticas.

1 Introdução

Dada uma expressão aritmética qualquer, inicialmente iremos realizar sua validação, verificando se todos os parênteses abertos na expressão foram devidamente fechados. A seguir, iremos converter a expressão de sua forma infixa para a forma posfixa, conforme exemplo abaixo:
Forma infixa: $A * B + C - (D/E + F)$
Forma posfixa: $AB * C + DE/F + -$

Após obtermos a forma posfixa da expressão iremos realizar sua avaliação, isto é, efetuar o cálculo da expressão e exibir seu resultado.

2 Implementação

A implementação foi realizada utilizando a linguagem de programação Java (javac 1.6.0_37) através da IDE 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 acessores. Como "valor"é do tipo Object, esta implamentação de pilha é genérica e servirá a qualquer tipo de dado.

2.2 Pilha.java

Possui os campos tamanho:int e topo:Elemento, os métodos implementados são os necessários para a utilização da pilha:

- public void empilhar(Object valor) ;
- public Object desempilhar() throws PilhaVaziaException;
- public int getTamanho();

- public boolean isVazio();
- public Object getTopo() throws PilhaVaziaException;

2.3 PilhaVaziaException.java

Exceção criada para identificar o erro de pilha vazia que pode ocorrer ao se tentar desempilhar de uma pilha que não tenha mais elementos.

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 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 caso a expressão aritmética 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 aritmética formada por variáveis.

2.4.5 public void validarNumerico() throws ProcessadorException

Valida uma expressão aritmética numérica.

2.4.6 public static int getPrioridade(String operador) throws ProcessadorException

Retorna a prioridade do operador aritmético informado.

2.4.7 public static boolean isParenteses(String s)

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

2.4.8 public static boolean isOperador(String s)

Retorna *true* caso o caracter 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 caracter apresentado representa uma variável, 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 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

Realizaremos o estudo da complexidade dos métodos apresentados 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 $O(n) = n$.

3.2 public String getPosfixa(boolean comVariaveis) throws ProcessadorException

```
    public String getPosfixa(boolean comVariaveis) throws ProcessadorException {
        String[] vetor;
        if(comVariaveis){
            validarVariavel();
            vetor = getVetorVariavel();
        }else{
            validarNumerico();
            vetor = getVetorNumerico();
        }
        String resposta = "";
        Pilha pilha = new Pilha();
        for (String e : vetor) {
            if (!comVariaveis && isValorNumerico(e)) {
                resposta += " " + e;
            }
            if (comVariaveis && isValorVariavel(e)) {
                resposta += " " + e;
            }
            if (isOperador(e)) {
                while (!pilha.isVazio() && !isParenteses((String) pilha.getTopo())
                    && !(getPrioridade(e) > getPrioridade((String) pilha.getTopo
                    ()))) {
                    resposta += " " + pilha.desempilhar();
                }
                pilha.empilhar(e);
            }
            if (isParenteses(e)) {
                if ("(".equals(e)) {
                    pilha.empilhar(e);
                } else if (")".equals(e)) {
                    while (!"(".equals(pilha.getTopo())) {
                        resposta += " " + pilha.desempilhar();
                    }
                    pilha.desempilhar();
                }
            }
        }
        while (!pilha.isVazio()) {
            resposta += " " + pilha.desempilhar();
        }
        resposta = resposta.trim();
        return resposta;
    }
```

Podemos verificar que existe apenas uma estrutura de repedição no código sem aninhamentos internos de outras estruturas de repetição, assim, concluímos que a complexidade do algoritmo é dada por $O(n) = n$.

3.3 public int avaliar() throws ProcessadorException

```
    public int avaliar() throws ProcessadorException {
```

```
        validarNumerico();
        Pilha p = new Pilha();
        String[] vetor = processaRegex("[0-9]+|.", getPosfixa(false));
        for(String e : vetor){
            if(" ".equals(e)){
                continue;
            }
            if(isValorNumerico(e)){
                p.empilhar(e);
            }
            if(isOperador(e)){
                int valor1, valor2, res = 0;
                valor1 = Integer.parseInt((String)p.desempilhar());
                valor2 = Integer.parseInt((String)p.desempilhar());
                if("+".equals(e)){
                    res = valor1+valor2;
                }
                if("-".equals(e)){
                    res = valor1-valor2;
                }
                if("*".equals(e)){
                    res = valor1*valor2;
                }
                if("/".equals(e)){
                    res = valor1/valor2;
                }
                p.empilhar(Integer.toString(res));
            }
        }
        if(p.getTamanho()>1){
            throw new ProcessadorException("Erro ao processar expressao: " +
                getExpressao());
        }
        return Integer.parseInt((String)p.getTopo());
    }
```

Podemos verificar que existe apenas uma estrutura de repedição no código sem aninhamentos internos de outras estruturas de repetição, assim, concluímos que a complexidade do algoritmo é dada por $O(n) = n$.

4 Listagem de testes executados

Para a realização dos testes, criamos a classe TesteExpressaoAritmetica.java e realizamos os testes conforme abaixo, todos foram bem sucedidos:

```
package gov.unb.cic.ed.trabalhopilhaed;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class TesteExpressaoAritmetica
```

```
        extends TestCase {
9
    public TesteExpressaoAritmetica(String testName) {
11      super(testName);
    }
13
    public static Test suite() {
15      return new TestSuite(TesteExpressaoAritmetica.class);
    }
17
    public void testExpressoes() throws ProcessadorException {
19      //   Infixa                              Posfixa
        //1  (A+B*C)                            A B C * +
21      //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 * -
23      //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 + - + - +
25      //6  (A*(B+(C*(D+(E*(F+G))))))          A B C D E F G + * + * + *
        ExpressaoAritmetica e = new ExpressaoAritmetica();
27
        //1
29      e.setExpressao("(A+B*C)");
        assertEquals("A B C * +", e.getPosfixa(true));
31
        //2
33      e.setExpressao("(A*(B+C)/D-E)");
        assertEquals("A B C + * D / E -", e.getPosfixa(true));
35
        //3
37      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));
39
        //4
41      e.setExpressao("(A+B*C/D*E-F)");
        assertEquals("A B C * D / E * + F -", e.getPosfixa(true));
43
        //5
45      e.setExpressao("(A+(B-(C+(D-(E+F)))))");
        assertEquals("A B C D E F + - + - +", e.getPosfixa(true));
47
        //6
49      e.setExpressao("(A*(B+(C*(D+(E*(F+G))))))");
        assertEquals("A B C D E F G + * + * + *", e.getPosfixa(true));
51
    }
53 }
```

5 Conclusão

Após o término do trabalho pudemos 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.

Referências

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