Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación
CC3001 – Algoritmos y Estructuras de Datos

Expresiones Algebraicas

TAREA 3

Nombre: Gabriel Iturra Bocaz
Correo: gabrieliturrab@ug.uchile.cl
Profesor: Patricio Poblete
Profesor Auxiliar: Manuel Caceres
Sergio Peñafiel
Fecha: 2 de Mayo de 2016

Contenido

I. Introducción

El siguiente informe detalla el desarrollo de la Tarea 3, del curso Algoritmos y Estructuras de Datos (CC3001). Como tercera tarea del curso, se pretende que el alumno se interiorice con el uso de Estructuras de datos elementales, en este caso Arboles Binarios, para la implementación de un Tipo de Dato Abstracto (TDA), y generando una interfaz para las operaciones pedidas.

El objetivo de esta tarea, es implementar un traductor de expresiones algebraicas de notación postfijo a notación infijo cuidando las reglas de la aritmética. Utilizando solo los operadores comunes (suma +, sustracción -, multiplicación *, división /, y el menos unario _) y como operandos las letras del abecedario (a – z).

Como solución al problema, fue necesario programar un Tipo de Dato Abstracto (TDA) llamado Expresion, cuyo constructor transforma el String de notación postfijo a un Árbol Binario de expresiones matemáticas, y con la ayuda de dos métodos que recorren el Árbol en in Orden se obtiene el mismo String en notación infijo.

II. Análisis del Problema

II.1. Descripción del Problema

Como se mencionó anteriormente, el objetivo de esta tarea es un implementar un traductor de expresiones algebraicas de notación postfijo a notación infijo, utilizando un TDA que permita representar estas fórmulas, crearlas e imprimirlas.

Para implementarlo utilizó un "Arbol Binario de expresiones matemáticas", el cual tiene en sus hojas los operandos de la expresión y en los demás nodos los operadores. Tal como se presenta en el ejemplo la figura 1 para la expresión "a + ((b – d) * c) – e".

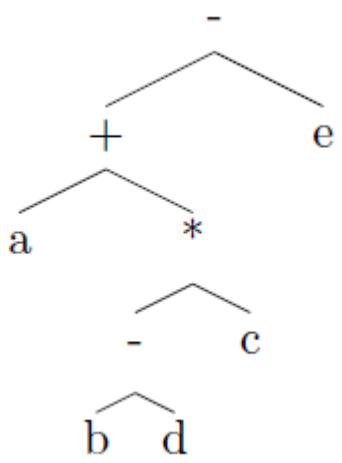

Figura 1: Árbol de Expresiones Algebraicas.

II.2. Implementación

A continuación, se detalla el TDA a implementar tanto su constructor como métodos. El TDA recibe el nombre de Expresion, en él existe un objeto NodoArbol vacío que se utiliza para el uso de los métodos.A continuación se explica cada uno de ellos:

- **Constructor Expresion(String Formula):** construye la representación de árbol para la expresión algebraica a partir de una fórmula que viene contenida en el String, cuya fórmula viene en notación postfijo, como en el siguiente ejemplo "a b +". Donde las operaciones permitidas son suma +, sustracción -, multiplicación *, división / y el menos unario, que se denotaba _ para diferenciar con la resta.
- **El método String toSimpleString():** método que genera una representación imprimible en forma de String en la notación normal, con un paréntesis para cada operación realizada. Para el caso "a b d – c * + e –" retorna ((a + ((b – d) * c) – e)".
- **El método String toString(): ti**ene la misma funcionalidad que toSimpleString, pero usando el mínimo de paréntesis. Para la fórmula "a b d – c * + e –" entrega a + (b – d) * c – e.

Usando el TDA señalado se debe programar el traductor, un ejemplo del traductor esperado se muestra a continuación.

Ingrese fórmula en postfijo:

a b +

Su traducción es:

(a + b) = a + b

**** Como consideraciones especiales, las únicas operaciones permitidas fueron las señaladas en el constructor, los operandos solo pueden ser letras del abecedario (a – z), los parámetros del String deben ser ingresados separados por espacios y solo en notación postfijo.*

III. Solución del Problema

III.1. Estrategia de Desarrollo

El desarrollo del TDA Expresion se puede separar en tres partes que son la programación del constructor y los dos métodos pedidos (toSimpleString() y toString()), en ellos se explica la estrategia de implementación, y se muestra las partes del código que refleja dicha implementación.

Los métodos pedidos son los siguientes:

- **El constructor Expresion(String Formula):** Para construir el Árbol Binario de Expresiones a partir del String en notación postfijo, fue necesario usar una pila (Para simplificar código y tiempo, se utilizó el objeto Stack predefinido en JAVA) que almacenara objetos, ya que en ella se almacenan objetos tipo String y tipo NodoArbol (que será explicada en la sección Modo de uso). Cuando se inserta un String dentro del argumento del constructor, si este mantiene el formato mencionado en ***, este se convierte en objeto de la clase String[] para ser iterado dentro de un bucle for, que itera la cantidad de objetos que posee el String. Al recorrer el primer elemento como es un operando este se guarda en la pila, sí el operador es binario, se guarda el siguiente elemento, una vez que lleva al operador (que se compara con el método equals de la clase String para determinar el tipo de operador), se crea un nuevo objeto NodoArbol y el operador toma el parámetro del nodo, luego se extraen los dos primeros valores dentro de la pila, el primero se enlaza a la derecha y el segundo a la izquierda, y el nuevo nodo se inserta dentro de la pila, y así sucesivamente. Es importante mencionar que si el primero elemento dentro de la pila es un árbol, este se enlaza al nodo creado donde se inserta un operador, lo mismo pasa si el siguiente elemento dentro de la pila también es un nodo o un árbol. Sí el operador es unario, solo se retira el primer elemento de la pila y se enlaza al lado derecho del nodo del operador, no sin antes crear un nodo para el elemento extraído por la pila. Sí el String insertado en el argumento del constructor no cumple el formato pedido se retorna un mensaje de error y el programa se cierra. En la figura 3, podemos ver un extracto del código que detalla esta parte.

```
if (str.equals("+")){
    if (pila.size()==1 || pila.isEmpty()){
        System.out.println("Error: La notación ingresada es errónea !!!!!");
        System.out.println("Verifique que la notación ingresa es la correcta,
        System.exit(-1);
    }
    arbol.info = str;
    Object obj2 = pila.pop();
    if (obj2 instanceof NodoArbol){
        arbol.der = (NodoArbol) obj2;
    }
    else{
        NodoArbol h2 = new NodoArbol(obj2,null,null);
        arbol.der = h2;
    }
    Object obj1 = pila.pop();
    if (obj1 instanceof NodoArbol){
        arbol.izq = (NodoArbol) obj1;
    }
    else{
        NodoArbol h1 = new NodoArbol(obj1,null,null);
        arbol.izq = h1;
    }
    pila.push(arbol);
}
```

Figura 2: Extracto Código del Constructor del TDA

Para el resto de las operaciones el código es análogo.

- **Método String toSimpleString():** Para este método, fue necesario implementar un método auxiliar propio de la clase NodoArbol, llamado String toSS(), este método recursivo recorre los elementos del árbol binario en In Order, que a través de los string i y d, se almacenan los string de los subárboles izquierdo y derecho para armar la expresión en infijo haciendo llamadas recursivas solo sí, los nodos hijos son distintos de null, y se añaden los paréntesis al principio de cada y al final. Finalmente este método es retornado por el método toSimpleString para la clase Expresion. Cabe mencionar que se deja una condición especial para insertar los paréntesis del menos unario, donde el nodo izquierdo es null y el nodo derecho no.

```
public String toSS(){
    String i = "(";
    String d = "";
    if (izq != null){
        i += izq.toSS();
    }
    else if (izq == null && der !=null){
        i = "(";
    }
    else {
        i = "";
    }
    if (der != null){
        d += der.toSS()+")";
    }
    else{
        d = "";
    }
    return i + info + d ;
}
```

Figura 3: Método String toSS()

- **Método String toString():** Para el último método se siguió una estrategia similar creando un método auxiliar llamado toS() para la clase NodoArbol. Sin embargo, para devolver una menor cantidad de paréntesis se fue comparando los nodos operadores padres con sus hijos, y se evaluó que casos requería paréntesis, indicando la excepción dentro de un if o else if para caso, como por ejemplo para un operador * en nodo padre y un operador – en un nodo hijo, dicha comparación debía hacerse en los nodos hijos izquierdo y derecho. En la figura 5, podemos ver la implementación de dicha función en el nodo hijo izquierdo.

```
public String toS(){
    String i = "";
    String d = "";
    if (izq != null){
        if (info.equals("*") && izq.info.equals("-")){
            i = "(";
            //info = ")"+info+"(";
            i += izq.toS()+")";
        }
        else if (info.equals("*") && izq.info.equals("+")){
            i = "(";
            i += izq.toS()+")";
        }
        else if (info.equals("/") && izq.info.equals("-")){
            i = "(";
            i += izq.toS()+")";
        }
        else if (info.equals("/") && izq.info.equals("+")){
                i = "(";
                i += izq.toS()+")";
        }
```

Figura 4: Método String toS().

****La implementación del traductor interactivo se explica en la próxima sección.*

IV. Modos de Uso.

IV.1. Clase NodoArbol.

Para la implementación del TDA señalando anteriormente, fue necesario crear una clase pública NodoArbol, donde en la variable info que es tipo objeto se almacena el parámetro del nodo, y en izq y der se tiene referencia a los nodos izquierdo y derecho respectivamente. En figura 5 se ve el código de la clase.

```
static public class NodoArbol{

    Object info;
    NodoArbol izq;
    NodoArbol der;

    public NodoArbol(){
        info = null;
        izq = null;
        der = null;
    }
```

Figura 5: Clase pública NodoArbol

IV.2. Traductor Interactivo.

Para el traductor interactivo, se creó otro archivo java en una clase main donde a través de la función void main, a través de la clase Scanner es posible generar las entradas para que un usuario inserte la consulta que tenga de una expresión en postfijo. A través de un ciclo while el programa pregunta infinitamente todas las fórmulas que el usuario necesite consultar de una por línea, retornando la operación en notación infija con paréntesis para cada operación y con el mínimo de paréntesis, y hace uso de un objeto de la clase Expresion y sus dos métodos.

```
public class main {

    public static void main(String[] args){
        while (true){
        Scanner scn = new Scanner(System.in);
        String expresion;
        System.out.println("Ingrese expresión matemática en notación postfija: ");
        System.out.println("Recuerde: Solo puede ingresar variables (letras) [a - z].");
        System.out.println("Y las operaciones permitidas son +, -, *, / y _ (menos unari
        expresion = scn.nextLine();
        Expresion e = new Expresion(expresion);
        System.out.println(e.toSimpleString()+" = "+e.toString());
```

Figura 6: Traductor Interactivo.

V. Ejemplos de Funcionamiento.

En esta sección se presentan ejemplos de funcionamiento mostrando entradas y salidas del programa en ejecución.

- En la figura 7, podemos ver la traducción de la expresión de ejemplo en la figura 1, un ejemplo del uso del menos unario y otro ejemplo que muestra el correcto funcionamiento del método toString().

```
Ingrese expresión matemática en notación postfija:
Recuerde: Solo puede ingresar variables (letras) [a - z].
Y las operaciones permitidas son +, -, *, / y _ (menos unario). Para terminar presione Enter sin parametros.
a b d - c * + e -
((a+((b-d)*c))-e) = a+(b-d)*c-e
Ingrese expresión matemática en notación postfija:
Recuerde: Solo puede ingresar variables (letras) [a - z].
Y las operaciones permitidas son +, -, *, / y _ (menos unario). Para terminar presione Enter sin parametros.
a _
(-a) = -a
Ingrese expresión matemática en notación postfija:
Recuerde: Solo puede ingresar variables (letras) [a - z].
Y las operaciones permitidas son +, -, *, / y _ (menos unario). Para terminar presione Enter sin parametros.
a b + c +
((a+b)+c) = a+b+c
```

Figura 7

- En la figura 8, se aprecia un ejemplo en el que el usuario ingresa un formato erróneo.

```
Ingrese expresión matemática en notación postfija:
Recuerde: Solo puede ingresar variables (letras) [a - z].
Y las operaciones permitidas son +, -, *, / y _ (menos unario). Para terminar presione Enter sin parametros.
a + b
Error: La notación ingresada es errónea !!!!!
Verifique que la notación ingresa es la correcta, y recuerde que + es un operador binario.
```

Figura 8

- En la figura 9, se aprecia un ejemplo donde los se ingresan operadores no permitidos o parámetros que no son letras.

```
Ingrese expresión matemática en notación postfija:
Recuerde: Solo puede ingresar variables (letras) [a - z].
Y las operaciones permitidas son +, -, *, / y _ (menos unario). Para terminar presione Enter sin parametros.
1 2 <
Error: Formato erróneo!!!!!!!!!!!!!!!
Recuerde: El String solo puede estar en notacion Polaca Inversa y separados por espacios.
Por ejemplo: a b +.
```

Figura 9

VI. Resultados y Conclusiones.

Si bien los TDA, pueden ser implementados con distintas Estructuras de Datos Elementales como árboles generales, listas enlazadas (ya sean simples, dobles, circulares, etc) o arreglos. Para el caso particular del traductor de notación postfijo a infijo resulta sumamente conveniente usar Arboles Binarios debido a cada una de las operaciones binarias pueden representarse fácilmente entre un nodo padre y sus hijos (izquierdo y derecho), también los operadores unario, solo que se considera nulo uno de los nodos hijos. En cambio, si se usara una lista enlazada o un arreglo los elementos se ordenan de una forma "lineal", por lo que en el caso que sean muchas operaciones u operandos resultad difícil distinguir las operaciones al realizar el parsing del String a la lista enlazada o el arreglo. Por eso, se concluye que un Arbol Binario es la estructura de datos más eficiente en este proceso, que deja mayor orden y facilidad al implementar los distintos métodos al recorrer los datos.

Otros métodos que podrían ser útiles:

- ➔ int result(String Formula) : que retorna el resultado de la expresión.
- ➔ Boolean isOperator(String op): que retorna un booleano para decir si un nodo es operador o no.
- ➔ String posfixToprefix(): este método podria cambiar la notación de postfijo a prefijo.
- ➔ Void seeTree(): un método que muestre el árbol de expresión.
- ➔ NodoArbol add(String operador, String operando): un método que agregue nuevos elementos al árbol de expresión, debido a que no los ingreso en el constructor, por ejemplos, ingresa "a b +" pero en realidad quiere traducir "a b + c d - *".

Entre muchas otras que se pueden implementar, la gran ventaja que tienen los TDA es que la única limitante es la creatividad del programador.

En cuanto a los resultados se llego al objetivo de traducción de las notaciones en postijo.