TP - Unidad 05-A
Árboles Binarios. Casos de Uso: árboles de expresiones

Ejercicio 1

Dadas las siguientes expresiones en notación infija, formada por los operadores binarios: +, -, *, / y ^ se pide mostrar cómo es el árbol binario correspondiente o indicar error en caso de que no se pueda construir.

1.1) ((2 + 3.5) * (-5 / -1))

1.2) ((2 + 3))

1.3) ((2 + 3.5) * -10)

Ejercicio 2

La clase que permite representar expresiones deberá implementar finalmente los siguientes métodos (por ahora comentados):

```
package core;

public interface ExpressionService {

	// lanza exception si no se puede evaluar porque hay algo mal formado en la expresion
//	double eval();

//	void preorder();

//	void inorder();

//	void postorder();

}
```

2.1) Bajar de Campus la clase ExpTree (y sus auxiliares). Su constructor lee un input ingresado por entrada estándar. Completar el siguiente código para terminar con su implementación (el método buildExpresison() de la clase Node está incompleto) . Las reglas que acepta son:

E -> (E + E)
E -> (E – E)
E -> (E * E)

E -> (E / E)
E -> (E ^ E)
E -> cte

Si la expresión no es correcta debe lanzar excepción.

```
package core;

import java.util.Scanner;


public class ExpTree implements ExpressionService {

	private Node root;

	public ExpTree() {
	    System.out.print("Introduzca la expresión en notación infija con todos los
paréntesis y blancos: ");

	      // token analyzer
	    Scanner inputScanner = new Scanner(System.in).useDelimiter("\\n");
	    String line= inputScanner.nextLine();
	    inputScanner.close();

	    buildTree(line);
	}

	private void buildTree(String line) {
		 // space separator among tokens
		 Scanner lineScanner = new Scanner(line).useDelimiter("\\s+");
		 root= new Node(lineScanner);
		 lineScanner.close();
	}



	static final class Node {
		private String data;
		private Node left, right;

		private Scanner lineScanner;

		public Node(Scanner theLineScanner) {
			lineScanner= theLineScanner;

			Node auxi = buildExpression();
			data= auxi.data;
			left= auxi.left;
			right= auxi.right;

			if (lineScanner.hasNext() )
				throw new RuntimeException("Bad expression");
		}

		private Node()	{
		}
```

```
            private Node buildExpression() {
                  // COMPLETAR
                  return null;
            }


      }  // end Node class


      // hasta que armen los testeos
      public static void main(String[] args) {
            ExpressionService myExp = new ExpTree();

      }

}  // end ExpTree class
```

Tip: Además de hasNext() se puede usar hasNext(string) para chequear si el próximo token es o no el esperado pero no consumirlo (como el peek() de la pila).

2.2) Armar testeos de unidad (por ejemplo, los del ejercicio 1) para verificar correcto funcionamiento en su construcción.

2.3) Realizar el seguimiento del método recursivo buildExpression() indicando paso a paso qué regla de derivación se utiliza, qué parte del árbol se construye y qué parte queda pendiente para la expresión: **" ((3 * (5 - 10.2)) - 2)"**

Ejercicio 3

3.1) Descomentar los métodos preOrder(), postOrder() e inOrder(). Implementarlos en la clase. Para el último deben agregarse los paréntesis. Los mismos imprimen en la salida estándar el resultado del tipo de recorrido solicitado.

Caso de Uso

Si el usuario ingresa: **"((2 + 3.5) * -10)\n"**

myExp= **new ExpTree();**

myExp.preorder();
myExp.postorder();
myExp.inorder();

debería obtenerse:
* + 2 3.5 -10

2 3.5 + -10 *

((2 + 3.5) * -10)

Implementarlos delegando en el nodo.

3.2) Agregar los testeos de unidad correspondientes para verificar correctitud.

Ejercicio 4

4.1) Agregar a la clase TreeExp el método eval() que se encarga de evaluar la expresión, es decir, devolver un número double.

Caso de Uso

Si el usuario ingresa: **"**

myExp= **new ExpTree();**
System.out.println(myExp.eval());

Devolvería -55

4.2) Agregar los testos correspondientes para verificar correctitud.

Ejercicio 5

Incorporar la evaluacación de expresiones que contengan variables. Podrían proporcionarse, por ejemplo, en el constructor.