PROYECTO CALCULADORA

Ernesto Eliezer Hernández Bernal

Ernesto Palma Molina

Jerónimo Deli Larios

Mariana Isabel Glatz Gutiérrez

Roberto Andrés Muller Ríos

21 de septiembre de 2022

Índice

Descripción del problema.

- Objetivo:

  El objetivo principal del proyecto es crear una “Calculadora” funcional, que pueda realizar las operaciones más básicas.
- Requisitos:

  Debe evaluar sumas, restas, divisiones y productos, así como la operación de la potencia con números negativos, números con punto decimal, números racionales, números irracionales y números enteros. El proyecto debe ser realizado en forma para ser entregado el jueves 22 de septiembre de 2022.
- Restriciones:

  Se deberá entregar una interfaz gráfica que simule una calculadora y emule sus funciones. El proyecto debe ser realizado en el IDE de “Apache NetBeans” y debe contener métodos que usen pilas como su estructura principal de datos.

  Debe cumplir con al menos 3 métodos generales: verificar valides de la expresión, convertir esa expresión a notación postfija y evaluar la expresión postfija para mostrar el resultado de las operaciones.

Solución Diseñada.

Las principales soluciones que debemos contemplar para tener una calculadora funcional y que cumpla con los requisitos son 3.

1. Revisar la sintaxis de la expresión para que al evaluarla tenga coherencia en las operaciones y siga un orden lógico que permita ejecutar operaciones algebricas.
2. Convertir la expresión ya revisada con anterioridad a su forma en Postfija teniendo en cuenta el orden de jerarquía de los operadores y paréntesis.

3. Dada la expresión en su forma Postfija, poder evaluar la expresión de una manera eficaz y que muestre el resultado adecuado.

Ahora, ¿Por qué es preferible convertir de notación Infija a notación Postfija?

“Las expresiones postfijas son, generalmente, más fáciles de evaluar que las expresiones infijas, porque no es necesario tener en cuenta ni reglas de precedencia, ni paréntesis. El orden de los valores y de los operadores en la expresión es suficiente para determinar el resultado. A menudo, los compiladores de los lenguajes de programación en los entornos de ejecución utilizan expresiones postfijas en sus cálculos internos, por esa misma razón. El proceso de evaluación de una expresión postfija puede enunciarse mediante una única regla simple: analizando la expresión de izquierda a derecha, hay que aplicar cada operación a los dos operandos inmediatamente precedentes y sustituir el operador por el resultado. Al final, lo que nos quedará será el resultado total de la expresión.” – John Lewis.

Para resolver los problemas anteriormente mencionados, necesitamos emplear tres clases y hacer uso de la estructura abstracta “Pilas” en ellas para que sea un código eficiente y bien estructurado.

En el apéndice, junto al código completo del proyecto, se presentan los diagramas UML que servirán de base para el diseño de las clases y se incluyen también los de la estructura Pilas como referencia.

Pruebas.

Se realizaron diversas pruebas para confirmar el correcto funcionamiento de nuestros códigos y nuestra GUI.

- Expresión con caracteres que no son dígitos
- Expresión con errores de sintaxis (balanceo de paréntesis, letras y números, expresiones erróneas.)
- Expresión con positivos y negativos
- Expresión con todos los operadores
- Expresión con diversas jerarquías de operaciones (paréntesis que cambiaban la jerarquía)
- Expresión con operadores juntos
- Expresión con un operador junto al símbolo menos que indicaba número negativo (así se decidió en equipo)
- Expresión con división entre cero y cero a la potencia cero

Durante el transcurso de las pruebas se notó la ausencia o ineficiencia de algunos métodos. Por ejemplo, en la clase EvaluacionSintaxis se tuvo que agregar varios métodos para considerar la mayor parte de casos posibles en los que había error de escritura. Ejemplo de lo anterior es anotar caracteres que no fueran números u operadores. También el caso “ ((8 +6)) ”, en el primer intento arrojó error debido a que consideraba el segundo ‘ (‘ como algo diferente a un operador y se tuvo que especificar el caso “ ((“ para seguir con la expresión. Otro caso fue que de manera ambiciosa hicimos válida una operación junto a ‘ – ‘ para indicar números negativos, es decir, el usuario no tenía necesidad de especificar con paréntesis los números negativos, lo que hace posible expresiones como “ 7 * - 2 ”. Otro caso notable fue el de indicar inválida la expresión “ 6 (8 ”, no puede haber paréntesis sin operador a lado que indique la operación a realizar. Pudimos ambicionar más e indicar que en esos casos tenía que multiplicar los números, pero entonces se tenía que especificar que fueran números o incluso “ (4) (5) ” lo tomara como válido y era reestructurar parte de la conversión a postfija y la parte de la evaluación ¿Cómo se

podría saber si el caso “ (8) (9) (7) ” es una triple multiplicación o son 3 números juntos que en postfija harán otras operaciones?. En teoría, por el análisis previo de la expresión, podríamos considerar que nos indica una triple multiplicación, pero, como en los métodos dejamos paréntesis para agrupar los números, era aumentar las condiciones y se perdería la limpieza del código que llevamos, que cabe aclarar, se nos ocurrió tarde esa implementación.

En las otras dos clases lo más notable fue precisamente agrupar números enteros mayores a un dígito, pues usamos métodos que analizan caracter por caracter y reaccionan con base en ese carácter ¿Cómo indicar que es 23 o 345.343? Por eso recurrimos a los paréntesis de tal forma que un operador nos indique el tamaño del número previo a ese operador, por ejemplo “ 34.56 - 5664” al convertirse en postfija quedaría “ (34.56) (5664) - ” lo que muestra en parte el porqué no nos aventuramos a señalar una multiplicar con “ ( ) ( ) ”. Cabe destacar que dejamos hasta la evaluación de Postfija la señalización de las operaciones “x / 0“ y “0 ^ 0” como erróneas, pues es ahí donde nos da los posibles resultados inválidos. Por practicidad, lanzamos una excepción con el mensaje “Error” en la interfaz hasta que es ejecutada. Un ejemplo de lo anterior es “ (2 – 2) ^ (5 / 5 – 1)” que al analizar el texto no tiene errores de sintaxis, pero al ser evaluada queda “ 0 ^ 0 ” que no es válido.

Limitaciones.

La Calculadora no está desarrollada para revolver expresiones matemáticas avanzadas:

- Calculo de funciones
- Calculo de derivadas o integrales
- Calculo de raíz cuadrada
- Calculo de raíz cubica
- Calculo de funciones trigonométricas
- Generar estadísticas
- Exhibir gráficos

Además, no contemplamos por cuestión de tiempo y especificaciones del proyecto: multiplicación indicada como " (2) (3) ", valuación de números complejos, indicar el error específico de la sintaxis o de la operación matemática, y, seguramente, alguna operación larga y que mezcle diferentes niveles de jerarquías y paréntesis.

Otra limitación fueron las pruebas Junit, en las que no pudimos contemplar todos los casos pues podrían extenderse tanto como quisiéramos. También, en la clase que convierte a postfija no había manera de ocupar adecuadamente las pruebas, pues cualquier expresión la modificaba, incluso con paréntesis mal balanceados o caracteres diferentes y, analizando el contexto, tiene sentido, pues así la diseñamos, con plena confianza en el método anterior que evalúa que la sintaxis de la expresión sea correcta. Incluso, aunque quisiéramos que el método de conversión postfija indicara si la expresión postfija es correcta tendríamos que evaluarla para saber si arroja un resultado válido o tendríamos que usar una comparación con un molde o la expresión postfija correcta, por ejemplo, " 2 - 3 + 7 " convertido quedaría "(2) (3) – (7) + " y comparamos con " (número)(número)operador (número)operador" pero si tuviéramos "(número)(número)(número)operador operador " también sería una expresión válida; en todo caso, lo que podríamos hacer es garantizar un mínimo correcto, en el que al inicio de postfija haya 2 números juntos y al final un operador, es decir, " (número)(número) . . . operador ".

Mejoras y Conclusiones.

Además de las mejoras obvias, como calcular operaciones más complejas, el proyecto podría mejorarse en el apartado de los errores que arroja al usuario. Por ejemplo, mostrar el error

> Nuestra experiencia en general fue buena, como cualquier proyecto individual o en equipo tiene su grado de complejidad y de trabajo, logramos identificar las habilidades de cada uno de nosotros, y establecimos roles de trabajo para reunir con más rapidez y eficiencia el trabajo de cada uno.
>
> Siempre es un reto colaborar con personas nuevas, cada una tiene ideas, conocimientos y habilidades muy valiosos que al fusionarlos se logra completar el objetivo dispuesto. En cada línea de código se ve demostrada el conocimiento y sobretodo el conocimiento aplicado de las diversas estructuras de datos, incluida la “Pila”. Es preciso afirmar que nuestro proyecto quedó concluido de la mejor manera y establecimos nuevas relaciones de amistad y laborales para colaborar en el futuro.

REFERENCIAS:

- Guardati Buemo Silvia del Carmen, 2015, *Estructuras de datos básicas: programación orientada a objetos con Java*, 1ª ed., México, D. F., Alfaomega Grupo Editor.

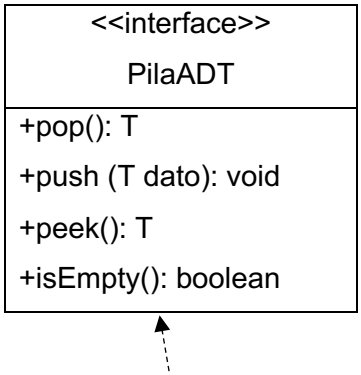
- John Lewis, Joseph Chase, 2006, *Estructuras de datos con Java: diseño de estructuras y algoritmos,* 2a ed., Madrid, Pearson Addison Wesley.

IMÁGENES:

- https://calculadorasonline.com/calculadora-cientifica-online/

Apéndice.

Diagramas



EvaluacionSintaxis
+EvaluacionSintaxis()
+noCaracteresNoAceptados(String formula):boolean
+parentesisBalanceados(String formula): boolean
+esNumero(char c): boolean
+esOperador(char c): boolean
+esOperadorSinMenos(char c): boolean
+noDobleOperador(String formula):boolean
+noDobleDecimal(String formula): boolean
+ noOperadorAntesDeParentesis (String formula): boolean
+expresionValida(String formula): boolean

Posfijo
+Posfijo() + pesoOp(Charactern): + cadena(String expresion): String

EvaluacionPostfija
+EvaluacionPostfija() +evaluaPost(String cadena): double +operador(char cadena):boolean +parentesis(char cadena): boolean

Ya que tenemos las clases en UML con sus métodos, es preciso también incluir las clases en UML de nuestra interfaz gráfica:

CalculadoraDemo
+main()

CalculadoraGUI
+keyPanel +textPanel +equalPanel +0-9button +inputTF +CalculadoraGUI() +display() +ClearListener() +KeyListener() +EqualListener()

- Se podrá notar la diferencia entre nombres de métodos y parámetros, lo que resalta la diferencia en la forma de trabajar de cada miembro del equipo.

Código

1. Evalúa sintaxis

```
package
calcula
dora;

          /**
           *
           * @author jeronimo
           */
          public class EvaluacionSintaxis {
              /**
               * Determina si una expresion matematica tiene los parametros
          balanceados
               *
               * @param formula String de la expresion que queremos evaluar
               * @return <ul>
               * <li>true: si los parentesis estan balanceados
               * <li>false: si los parentesis no estan balanceados
               * </ul>
               */
              public static boolean parentesisBalanceados(String formula){
                  boolean res=true;
                  Character c;
                  PilaA<Character> pila= new PilaA();
                  int i=0;
                  while(i<formula.length() && res){
                      c= formula.charAt(i);
                      if(c=='('){
                          pila.push(c);
                      }
                      else{
                          if(c==')'){
```

```
                    if(pila.isEmpty()){
                        res=false;
                    }
                    else{
                        pila.pop();
                    }
                }
            }
            i++;
        }
        return res;
    }
    /**
     * Metodo que nos permite determinar si un character es un operador
('+','/','-','*')
     *
     * @param c el caracter que queremos evaluar
     * @return <ul>
     *<li>true: si el character es '+','/','-','*';
     *<li>false: si el character no es '+','/','-','*';
     *</ul>
     */
    public static boolean esOperador(char c){
        boolean res= false;
        if (c=='*' || c=='/' || c=='+' || c=='-'){
            res= true;
        }
        return res;
    }
    /**
     * Metodo para revisar que no existan dos operados juntos en una
expresion
     *
     * @param formula String que representa la expresion que queremos
evaluar
     * @return <ul>
     * <li>true: si no hay dos operadores juntos en la expresion
     * <li>flase: si hay dos operados juntos en la expresion
     * </ul>
     */
    public static boolean noDobleOperador(String formula){
        boolean res= true;
        Character c;
```

```
        Character c2;
        //nunca debe llegar a .length ya que checamos dos valores uno mayor
al contador
        int i=0;
        while(i<formula.length()-1 && res){
            c= formula.charAt(i);
            c2= formula.charAt(i+1);
            if(esOperador(c) && esOperador(c2)){
                res= false;
            }
            i++;
        }
        return res;
    }
    /**
     * Metodo para revisar que no existan dobles decimales
     *
     * @param formula String que representa la expresion que queremos
evaluar
     * @return <ul>
     * <li>true: si no hay doble decimal
     * <li>flase: si hay doble decimal
     * </ul>
     */
    public static boolean noDobleDecimal(String formula){
        boolean res= true;
        Character c;
        PilaA<Character> pila= new PilaA();
        int i=0;
        while(i<formula.length() && res){
            c= formula.charAt(i);
            if(c=='.'){
                if(pila.isEmpty()){
                    pila.push(c);
                }
                else{
                    res=false;
                    }
            }
            else{
                if(esOperador(c)&& !pila.isEmpty()){
                    pila.pop();
                }
```

```
            }
            i++;
        }
        return res;
    }
    /**
     * Metodo para revisar que no existan parentesis sin operador
     *
     * @param formula String que representa la expresion que queremos
evaluar
     * @return <ul>
     * <li>true: si no hay parentesis sin operador
     * <li>flase: si hay parentesis sin operador
     * </ul>
     */
    public static boolean noOperadorAntesDeParentesis(String formula){
        boolean res= true;
        Character c;
        Character c2;
        //empezamos en 1 ya que queremos revisar el del contador y el
anteriro
        int i=1;
        while(i<formula.length() && res){
            c= formula.charAt(i);
            c2= formula.charAt(i-1);
            if(c=='(' && !esOperador(c2)){
                res= false;
            }
            i++;
        }
        return res;
    }
    /**
     * Metodo para evaluar que la expresion este bien escrita
     *
     * @param formula String que representa la expresion que queremos
evaluar
     * @return <ul>
     * <li>true: si esta bien escrita
     * <li>flase: si no esta bien escrita
     * </ul>
     */
    public static boolean expresionValida(String formula){
```

```
            boolean res= false;

        if(parentesisBalanceados(formula)&&noDobleOperador(formula)&&noDobleDecimal
        (formula)&&noOperadorAntesDeParentesis(formula)){
                res=true;
            }
            return res;
        }
```

2. Convierte Infija a Postfija

```
package
calculador
a;

        import java.util.HashMap;
        import pilas.PilaA;
        import pilas.PilaADT;

        /**
         * Definición de clase Postfijo que convierte expresión en postfija
         * @author m-gla
         */
        public class Posfijo {
            /**
             * Método que regresa cadena de caracteres en postfija
             * @param expresion
             * @return String
             */
            public String cadena(String expresion){
                String posfija = "";
                PilaADT <Character> pila = new PilaA();
                HashMap <Character, Integer> operadores = new HashMap <>();
                operadores.put('+',1);
                operadores.put('-', 1);
                operadores.put('*',2);
                operadores.put('/', 2);
                operadores.put('^', 3);

                int i = 0;
                Character c;

                while(i < expresion.length()){
```

```
            c = expresion.charAt(i);
            if((i == 0 && expresion.charAt(i) == '-') ||
Character.isDigit(c) || c == '.' || (c == '-' &&
!Character.isDigit(expresion.charAt(i-1)))){
                posfija += '(';
                posfija += c;
                int j = i + 1;
                while(j < expresion.length() &&
(Character.isDigit(expresion.charAt(j)) || expresion.charAt(j) == '.')){
                    posfija += expresion.charAt(j);
                    j++;
                }posfija += ')';
                i = j - 1;
            }
            else if(!pila.isEmpty()){
                if(c == '('){
                    pila.push(c);
                }else if(c == ')'){
                    while(pila.peek() != '('){
                        posfija += pila.pop();
                    }
                    pila.pop();
                }else if(pila.peek() == '(' || operadores.get(c) >
operadores.get(pila.peek())){
                    pila.push(c);
                }else if(operadores.get(c) <=
operadores.get(pila.peek())){
                    posfija+= pila.pop();
                    pila.push(c);
                }
            }else{
                    pila.push(c);
                }i++;
        }while(!pila.isEmpty()){
            posfija += pila.pop();
        }
        return posfija;
    }

}
```

3. Evalúa la expresión Postfija

```
package
calculadora;

                    import pilas.PilaA;
                    import pilas.PilaADT;

                    /**
                     *
                     * @author Ernesto Palma
                     * * Definiciòn de la clase Evaluación de Postfija que usa
                    pilas
                     * @param <T>
                     */
                    public class EvaluacionPostfija <T>{

                        /**
                         * Constructor por omisión
                         */
                        public EvaluacionPostfija(){

                        }

                        /**
                         * Método que regresa el resultado de une expresión en
                    postfija usando pilas
                         * @param cadena
                         * @return double
                         */
                        public double evaluaPost(String cadena){
                            PilaADT<Double> pila=new PilaA();
                            int i=0;
                            double resul, numer;
                            while(i<cadena.length()){
                                System.out.println("Valor de i en inicio: "+i);
                                if(this.operador(cadena.charAt(i))){
                                    numer=pila.pop();
```

```
                switch(cadena.charAt(i)){ // es más fácil usar
casos que condicional if
                    case'+': resul=pila.pop() + numer;
                            pila.push(resul);
                            break;
                    case'-': resul=pila.pop() - numer;
                            pila.push(resul);
                            break;
                    case'*': resul=pila.pop() * numer;
                            pila.push(resul);
                            break;
                    case'/': resul=pila.pop() / numer;
                            pila.push(resul);
                            break;
                    case'^': resul=Math.pow(pila.pop(), numer);
                            pila.push(resul);
                            break;
                }
            }
            else if(this.parentesis(cadena.charAt(i))){
                StringBuilder sb=new StringBuilder();
                int cont=0;
                while(!this.parentesis(cadena.charAt(i+1))){
                    sb.append(cadena.charAt(i+1));
                    i++;
                    cont++;
                }
                pila.push(Double.parseDouble(sb.toString()));
                i++;
                System.out.println("valor de i"+i);
            }
            else

pila.push(Double.parseDouble(cadena.substring(i, i+1)));
            i++;
        }
        return pila.peek();
    }

    /**
     * Método que indica si un caracter de una cadena de texto
es un operador
     * @param cadena
```

```
     * @return boolean
     */
    public boolean operador(char cadena){
        boolean resp=false;
        if(cadena == '+' | cadena == '-' | cadena == '*' |
cadena == '/' | cadena == '^')
            resp=true;
        return resp;
    }

    /**
     * Método que indica si un caracter de una cadena de texto
es un paréntesis
     * @param cadena
     * @return
     */
    public boolean parentesis(char cadena){
        boolean resp=false;
        if(cadena == '(' | cadena== ')')
            resp = true;
        return resp;
    }

}
```