

calculadora cientifica



Alumnos: marcos contreras, felipe flores, benjamin perez rossi

Profesora: María Belén Hipólito

**Introducción**

La Notación Polaca Inversa (NPI), también conocida como notación polaca postfix, es una forma de escribir expresiones matemáticas en las que los operadores siguen a sus operandos. Esto elimina la necesidad de paréntesis para indicar el orden de las operaciones, ya que la notación polaca inversa garantiza un orden claro de evaluación.

**Concepto**

En la notación polaca inversa, los operadores se colocan después de sus operandos. Por ejemplo, la expresión "3 + 4" se escribiría como "3 4 +". Esto permite una evaluación más sencilla de las expresiones matemáticas sin la necesidad de paréntesis para indicar el orden de las operaciones, es una forma eficiente de escribir y evaluar expresiones matemáticas sin la necesidad de paréntesis para indicar el orden de las operaciones. La implementación en Java proporciona una manera sencilla de realizar cálculos utilizando esta notación.

**CODIGO:**

import java.util.Scanner;

import java.util.Stack;

public class CalculadoraPolacaInversa1 {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Ingrese la expresión en notación infija (por ejemplo, '2 + 3 \* (4 + 5)'):");

String expresion = scanner.nextLine();

String rpnExpresion = convertirInfijaARPN(expresion);

double resultadoFinal = evaluarRPN(rpnExpresion);

System.out.println("El resultado es: " + resultadoFinal);

}

private static String convertirInfijaARPN(String expresion) {

StringBuilder rpn = new StringBuilder();

Stack<Character> operadores = new Stack<>();

for (int i = 0; i < expresion.length(); i++) {

char caracter = expresion.charAt(i);

if (esNumero(caracter)) {

rpn.append(caracter);

} else if (esOperador(caracter)) {

while (!operadores.isEmpty() && precedencia(caracter) <= precedencia(operadores.peek())) {

rpn.append(operadores.pop());

}

operadores.push(caracter);

} else if (caracter == '(') {

operadores.push(caracter);

} else if (caracter == ')') {

while (!operadores.isEmpty() && operadores.peek() != '(') {

rpn.append(operadores.pop());

}

operadores.pop(); // Quitar el '('

}

}

while (!operadores.isEmpty()) {

rpn.append(operadores.pop());

}

return rpn.toString();

}

private static double evaluarRPN(String expresion) {

Stack<Double> pila = new Stack<>();

for (int i = 0; i < expresion.length(); i++) {

char caracter = expresion.charAt(i);

if (esNumero(caracter)) {

pila.push((double)(caracter - '0'));

} else if (esOperador(caracter)) {

double segundoNumero = pila.pop();

double primerNumero = pila.pop();

double resultado = aplicarOperador(caracter, primerNumero, segundoNumero);

pila.push(resultado);

}

}

return pila.pop();

}

private static boolean esOperador(char caracter) {

return caracter == '+' || caracter == '-' || caracter == '\*' || caracter == '/';

}

private static double aplicarOperador(char operador, double a, double b) {

switch (operador) {

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

if (b != 0) {

return a / b;

} else {

throw new ArithmeticException("División por cero");

}

default:

throw new IllegalArgumentException("Operador no válido: " + operador);

}

}

private static boolean esNumero(char caracter) {

return caracter >= '0' && caracter <= '9';

}

private static int precedencia(char operador) {

switch (operador) {

case '+':

case '-':

return 1;

case '\*':

case '/':

Return 2;

default:

Return 0;

}

}

}

**Método main:**

**\***Este es el punto de entrada del programa.

**\***Se crea un objeto Scanner para leer la entrada del usuario desde la consola.

**\***Se pide al usuario que ingrese la expresión en notación infija.

**\***La expresión en notación infija se convierte a notación polaca inversa (RPN) utilizando el método convertirInfijaARPN.

**\***La expresión en RPN se evalúa utilizando el método evaluarRPN.

**\***El resultado final se muestra en la consola.

**Método convertirInfijaARPN:**

**\***Este método convierte una expresión en notación infija a notación polaca inversa (RPN).

**\***Utiliza un StringBuilder llamado rpn para construir la expresión en RPN.

**\***Utiliza un Stack llamado operadores para almacenar los operadores mientras se procesa la expresión.

**\***Recorre cada carácter de la expresión en notación infija y realiza las siguientes acciones:

**\***Si el carácter es un número, lo agrega directamente a la expresión en RPN.

**\***Si el carácter es un operador (+, -, \*, /), se maneja según su precedencia y asociatividad.

**\***Si el carácter es (, se agrega a la pila de operadores.

**\***Si el carácter es), se desapilan los operadores de la pila y se agregan a la expresión en RPN hasta encontrar (.

**\***Al finalizar el bucle, se desapilan los operadores restantes y se agregan a la expresión en RPN.

**\***La expresión en RPN se devuelve como una cadena de texto.

**Método evaluarRPN:**

**\***Este método evalúa una expresión en notación polaca inversa (RPN) y devuelve el resultado.

**\***Utiliza un Stack llamado pila para almacenar los números mientras se procesa la expresión.

**\***Recorre cada carácter de la expresión en RPN y realiza las siguientes acciones:

**\***Si el carácter es un número, lo convierte a double y lo apila en la pila.

**\***Si el carácter es un operador (+, -, \*, /), se desapilan los dos últimos números de la pila, se aplica el operador y se vuelve a apilar el resultado.

**\***Al finalizar el bucle, el único elemento restante en la pila es el resultado final, que se devuelve.

**Métodos de utilidad:**

**\***esOperador: Verifica si un carácter es un operador (+, -, \*, /).

**\***aplicarOperador: Aplica un operador a dos números y devuelve el resultado.

**\***esNumero: Verifica si un carácter es un dígito numérico.

**\***precedencia: Devuelve la precedencia de un operador (+, -, \*, /).

El código proporciona una forma de evaluar expresiones matemáticas utilizando la notación polaca inversa, lo que puede ser útil en situaciones donde se necesita evaluar expresiones de forma eficiente y sin ambigüedades de precedencia de operadores.