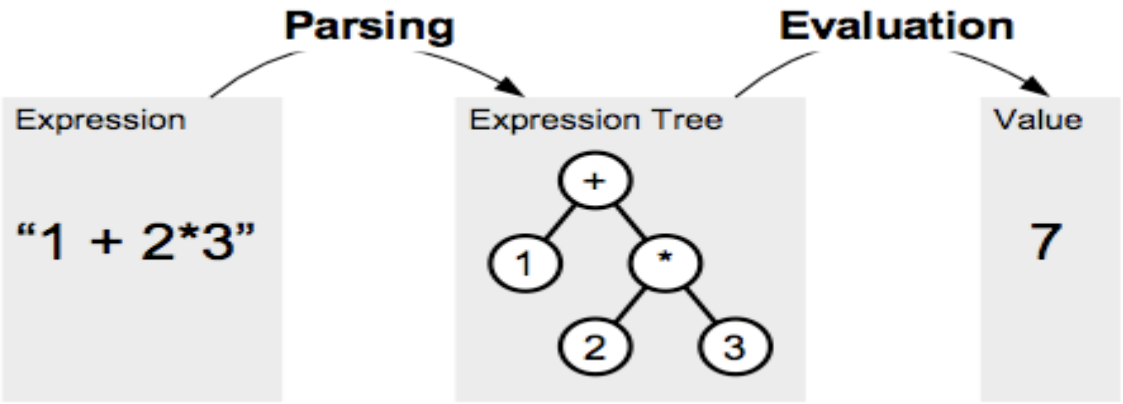
Lenguajes y paradigmas de programación
Proyecto Unidad 3 - Mini-Compilador para Expresiones Matemáticas.



Profesor: Roberto García

Asignatura: Lenguajes y Paradigmas de la programación.

Estudiantes: Rodrigo Andre San Juan Navarro – Martín Alberto Droguett Tondro.

Versión del Proyecto: A.

ÍNDICE

Introducción

En este proyecto se nos propuso desarrollar un mini-compilador matemático en el que tiene que estar presentes las principales etapas del proceso de compilación, específicamente: **tokenización, parsing, conversión y evaluación** de expresiones.

En conjunto, decidimos utilizar el lenguaje de programación **Java**, ya que es el lenguaje que actualmente estamos utilizando en nuestra formación académica. Tras una investigación inicial, buscamos bibliotecas que permitieran transformar expresiones desde notación infija a prefija o postfija, pero al no encontrar soluciones que se ajustaran a nuestras necesidades, optamos por implementar nuestras propias funciones para llevar a cabo dichas conversiones. Además, Java nos ofreció herramientas útiles, como la colección Stack, que nos permitió gestionar fácilmente las estructuras necesarias durante el procesamiento de las expresiones, lo que facilitó gran parte del desarrollo. Más adelante, en el desarrollo del trabajo, se explicará detalladamente cómo se implementaron estas funciones y por qué fueron esenciales en nuestra solución.

Desarrollo

Interfaz

Como primera etapa, decidimos implementar una interfaz en Java que nos permitiera organizarnos mejor y visualizar de forma estructurada las funciones que debíamos desarrollar. Esta planificación inicial nos ayudó a tener una idea más clara de cómo abordar la lógica del programa y facilitó la implementación de cada componente del proyecto.

```
interface Operaciones {  1 usage  1 implementation
    void tokenizador(String input);  1 usage  1 implementation
    int parser();  1 usage  1 implementation
    void convertirPrefija();  1 usage  1 implementation
    void convertirPostfija();  1 usage  1 implementation
}
```

Tokenizador

Al comenzar, decidimos que, al ingresar una expresión, esta debía dividirse en dos partes si se encontraba un signo igual (=). Esto nos permite **diferenciar entre una asignación de variable** y una simple expresión matemática. Así, el lado izquierdo del = se guarda como el nombre de la variable, y el derecho como la expresión a evaluar.

```
String[] parts = input.split( regex: "=",  limit: 2);
if (parts.length == 2) {
    variableAsignacion = parts[0].trim();      //
    expresion = parts[1].trim();              // L
} else {
    variableAsignacion = null;                // N
    expresion = input.trim();                 // T
}
```

Luego, para analizar la expresión, se utilizan dos estructuras: un ArrayList llamado tokens, donde se almacenan los elementos individuales de la expresión (números, variables, operadores, paréntesis, etc.), y un StringBuilder llamado actual, que se

usa para construir temporalmente los tokens carácter por carácter.

```
List<String> tokens = new ArrayList<>();
StringBuilder actual = new StringBuilder();
```

A continuación, se recorre toda la cadena de la expresión. Los espacios en blanco se ignoran. Si el carácter actual es una letra o número, se acumula en el StringBuilder. Si se encuentra un símbolo que representa un operador o paréntesis, primero se guarda el contenido acumulado (si lo hay) como token, y luego se agrega el símbolo como un token separado.

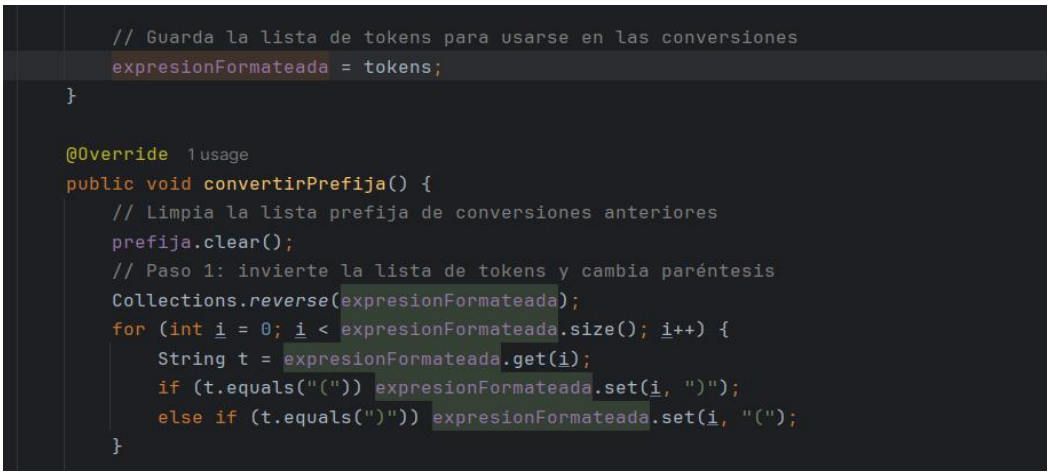
Al finalizar el recorrido, si el StringBuilder aún contiene texto, se agrega también como un token. Finalmente, todos los tokens extraídos se guardan en la lista expresionFormateada, dejándola lista para el proceso de conversión y evaluación.

```
// Recorre carácter a carácter la expresión para separar tokens
for (int i = 0; i < expresion.length(); i++) {
    char c = expresion.charAt(i);
    if (Character.isWhitespace(c)) {
        continue; // Ignora espacios
    }
    if (Character.isLetterOrDigit(c)) {
        // Si es letra o dígito, acumula en el buffer actual
        actual.append(c);
    } else {
        // Si encuentra un operador o paréntesis, primero agrega el token acumulado
        if (actual.length() > 0) {
            tokens.add(actual.toString());
            actual.setLength(0);
        }
        // Luego agrega el carácter actual como token individual
        tokens.add(Character.toString(c));
    }
}
// Al finalizar, si hay algo en el buffer, se agrega
if (actual.length() > 0) {
    tokens.add(actual.toString());
}
```

Prefija

Durante el desarrollo de esta función, nos dimos cuenta de un problema importante: para evaluar correctamente las expresiones, era necesario respetar un **orden de precedencia de los operadores**. Por esta razón, decidimos trabajar con expresiones en notación **infija**, pero para poder evaluarlas con mayor facilidad, primero era necesario convertirlas a **notación prefija**. Para ello, utilizamos la función convertirPrefija.

Primero, esta función **limpia** cualquier conversión anterior. Luego, **invierte** la expresión formateada (generada por el tokenizador) y **cambia los paréntesis**: los (se transforman en) y viceversa. Esto es necesario porque el algoritmo que usaremos (una variación del Shunting-Yard) requiere operar sobre la expresión invertida.



```
        // Guarda la lista de tokens para usarse en las conversiones
        expresionFormateada = tokens;
    }

    @Override  1 usage
    public void convertirPrefija() {
        // Limpia la lista prefija de conversiones anteriores
        prefija.clear();
        // Paso 1: invierte la lista de tokens y cambia paréntesis
        Collections.reverse(expresionFormateada);
        for (int i = 0; i < expresionFormateada.size(); i++) {
            String t = expresionFormateada.get(i);
            if (t.equals("(")) expresionFormateada.set(i, ")");
            else if (t.equals(")")) expresionFormateada.set(i, "(");
        }
```

Después de eso, la función recorre cada token de la expresión. Si el token es un **número o una variable**, se agrega directamente a la lista prefija.
Si es un **paréntesis de apertura (()**, se agrega a la pila (Stack).
Si es un **paréntesis de cierre ())**, se van **sacando elementos de la pila y agregando a la salida** hasta encontrar el paréntesis de apertura correspondiente, que luego se descarta.
Si el token es un **operador**, se comparan sus prioridades con los operadores que están en la cima de la pila. Si los de la pila tienen **mayor prioridad**, se van sacando y agregando a la salida. Finalmente, se agrega el nuevo operador a la pila.

Una vez procesados todos los tokens, se **vacía la pila** y se agregan sus contenidos restantes a la lista de salida. Por último, se **invierte la lista prefija** para que quede en el orden correcto de evaluación.

```
    Stack<String> stack = new Stack<>();
    // Paso 2: aplicar algoritmo shunting-yard sobre la lista invertida
    for (String token : expresionFormateada) {
        if (token.matches( regex: "\\d+") || token.matches( regex: "[a-zA-Z]+")) {
            // Si es número o identificador, agregar directamente a la salida
            prefija.add(token);
        } else if (token.equals("(")) {
            // Paréntesis de apertura → push
            stack.push(token);
        } else if (token.equals(")")) {
            // Paréntesis de cierre → pop hasta encontrar "("
            while (!stack.isEmpty() && !stack.peek().equals("(")) {
                prefija.add(stack.pop());
            }
            if (!stack.isEmpty()) stack.pop(); // Remover "("
        } else {
            // Operador: pop mientras la prioridad sea mayor en la pila
            while (!stack.isEmpty() && !stack.peek().equals("(")
                    && prioridad(token) < prioridad(stack.peek())) {
                prefija.add(stack.pop());
            }
            stack.push(token);
        }
    }
```

```
    // Vaciar pila restante
    while (!stack.isEmpty()) {
        prefija.add(stack.pop());
    }
    // Finalmente, revertir la salida para obtener la notación prefija correcta
    Collections.reverse(prefija);
}
```

Prioridad

Para poder determinar la prioridad de cada operador, se creó la función prioridad, la cual recibe un **operador como argumento** y retorna un **valor numérico** que representa su nivel de precedencia. De esta manera, operadores como ^ (potencia) tienen mayor prioridad que * o /, y estos a su vez tienen mayor prioridad que + o -. Esta función es fundamental para que tanto la conversión como la evaluación de expresiones respeten el **orden correcto de operaciones**, tal como se muestra en la tabla de prioridades definida en el proyecto.

```java
// Método auxiliar para determinar prioridad de operadores
private int prioridad(String op) {  4 usages
    switch (op) {
        case "^": return 3;
        case "*": case "/": return 2;
        case "+": case "-": return 1;
        default: return 0;
    }
}
```

Parser

Después de aplicar la función convertirPrefija, comienza la evaluación con el método parser, ya que utilizamos la notación **prefija** para evaluar las expresiones de forma más sencilla. El proceso recorre los tokens **de derecha a izquierda**.

Durante este recorrido:

- Si encuentra un **número**, lo convierte a entero y lo **guarda en la pila (Stack)**.
- Si encuentra una **variable**, busca su valor en el mapa y lo **agrega a la pila**.
- Si encuentra un **operador**, **extrae dos operandos** de la pila, **realiza la operación correspondiente** (como suma, resta, multiplicación, etc.) y **guarda el resultado de vuelta en la pila**.

Al finalizar el recorrido, el **resultado final** de la expresión estará en la cima de la pila, listo para ser mostrado o asignado a una variable si corresponde.

```java
public int parser(){
    // Evaluar la notación prefija
    Stack<Integer> stack = new Stack<>();
    for (int i = prefija.size() - 1; i >= 0; i--) {
        String token = prefija.get(i);
        if (token.matches( regex: "\\d+")) {
            // Si es número, convertir y push
            stack.push(Integer.parseInt(token));
        } else if (variables.containsKey(token)) {
            // Si es variable, recuperar su valor
            stack.push(variables.get(token));
        } else {
            // Operador → pop de operandos y aplicar operación
            int a = stack.pop();
            int b = stack.pop();
            int result;
            switch (token) {
                case "+": result = a + b; break;
                case "-": result = a - b; break;
                case "*": result = a * b; break;
                case "/":
                    if (b == 0)
                        throw new ArithmeticException("División por cero");
                    result = a / b;
                    break;
                case "^":
                    result = (int) Math.pow(a, b);
                    break;
```

```java
    // El resultado final queda en la pila
    int salida = stack.pop();

    // Si había asignación, guardarla en el mapa de variables
    if (variableAsignacion != null) {
        variables.put(variableAsignacion, salida);
        System.out.println("Variable asignada: " + variableAsignacion + " = " + salida);
        System.out.println("Variables: " + variables);
    }

    return salida;
}
```

PostFija

Para transformar una expresión a **notación postfija**, utilizamos el **algoritmo Shunting-Yard** clásico. Este algoritmo permite convertir expresiones infijas (como las que usamos normalmente) a una forma más fácil de evaluar por una computadora, eliminando paréntesis y respetando las prioridades de los operadores.

Primero, se limpia cualquier contenido previo en la lista postfija. Luego, se recorre cada **token** (número, variable, operador o paréntesis) de la expresión formateada:

- Si el token es un **número o una variable**, se agrega directamente a la salida postfija.
- Si es un **operador**, se compara con los que hay en la pila (Stack):
    - Si el operador en la pila tiene **mayor o igual prioridad**, se extrae de la pila y se agrega a la salida.
    - Luego, el operador actual se agrega a la pila.
- Si se encuentra un **paréntesis de apertura (**, se agrega a la pila.
- Si se encuentra un **paréntesis de cierre)**, se extraen operadores de la pila hasta encontrar el paréntesis de apertura correspondiente.

Al finalizar el recorrido, se **vacía la pila**, agregando cualquier operador restante a la salida. El resultado es una expresión postfija lista para ser evaluada correctamente, respetando el orden de operaciones.

```java
@Override  1 usage
public void convertirPostfija() {
    postfija.clear();
    Stack<String> stack = new Stack<>();
    // Algoritmo shunting-yard clásico para notación postfija
    for (String token : expresionFormateada) {
        if (token.matches( regex: "\\d+") || token.matches( regex: "[a-zA-Z]+")) {
            // Números o variables van directo a la salida
            postfija.add(token);
        } else if (token.equals("(")) {
            stack.push(token);
        } else if (token.equals(")")) {
            // Pop hasta encontrar "("
            while (!stack.isEmpty() && !stack.peek().equals("(")) {
                postfija.add(stack.pop());
            }
            if (!stack.isEmpty()) stack.pop();
        } else {
            // Operador: pop según prioridad
            while (!stack.isEmpty() && !stack.peek().equals("(")
                    && prioridad(token) <= prioridad(stack.peek())) {
                postfija.add(stack.pop());
            }
            stack.push(token);
        }
    }
    // Vaciar pila restante
    while (!stack.isEmpty()) {
        postfija.add(stack.pop());
    }
}
```

Main

Por último, en el main utilizamos estas funciones de forma secuencial: primero llamamos al **tokenizador** con la entrada del usuario, que divide y organiza la expresión en componentes (tokens). Luego, aplicamos la conversión a **notación prefija o postfija** según sea necesario. Finalmente, se llama al método **parser**, que se encarga de evaluar la expresión convertida y simular el comportamiento de un evaluador matemático.

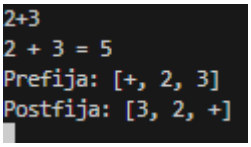
```java
public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    Programa programa = new Programa();
    while (in.hasNextLine()) {
        String linea = in.nextLine();
        if (linea.isBlank()) break;
        try {
            programa.tokenizador(linea);
            programa.convertirPrefija();
            programa.convertirPostfija();
            programa.parser();
            System.out.println("Prefija: "+ programa.getPrefija());
            System.out.println("Postfija: "+ programa.getPostfija());
        } catch (RuntimeException e) {
            // Captura errores de parsing, operadores desconocidos, división por cero, etc.
            System.out.println("Expresión inválida");

        }
    }
```

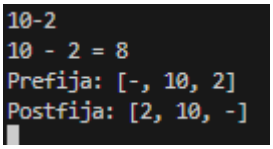
Pruebas

Entradas y Salidas de Prueba

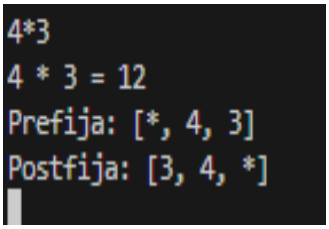
• Suma: 2 + 3 (Esperado: 5)



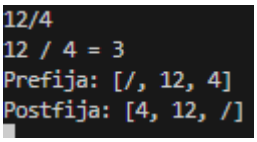
• Resta: 10- 2 (Esperado: 8)



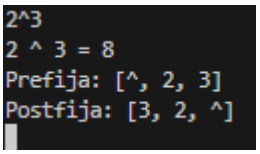
• Multiplicación: 4 * 3 (Esperado: 12)



• División: 12 / 4 (Esperado: 3)



• Potencia: 2 ^ 3 (Esperado: 8)



• Paréntesis: (5 + 3) * 2 (Esperado: (5+3) * 2 = 16)

```
x = (5+3)*2
5 + 3 = 8
8 * 2 = 16
Variable asignada: x = 16
Variables: {x=16}
Prefija: [*, +, 5, 3, 2]
Postfija: [2, 3, 5, +, *]
```

• Asignación: x = 5 + 2 (Esperado: x = 7)

```
x=5+2
5 + 2 = 7
Variable asignada: x = 7
Variables: {x=7}
Prefija: [+, 5, 2]
Postfija: [2, 5, +]
```

• Uso de variables: y = x * 3 (Esperado: y = 7 * 3 = 21)

```
y = x *3
7 * 3 = 21
Variable asignada: y = 21
Variables: {x=7, y=21}
Prefija: [*, x, 3]
Postfija: [3, x, *]
```

Conclusión

Este proyecto nos permitió explorar nuevas formas de programar utilizando **estructuras algorítmicas**, las cuales fueron fundamentales para implementar la conversión de expresiones en Java. Uno de los principales desafíos fue manejar correctamente la **prioridad de los operadores**, especialmente al momento de realizar el **parser**. Para resolver este problema, fue necesario convertir primero la expresión infija a una notación más manejable, ya sea **postfija o prefija**, utilizando como base el **algoritmo Shunting-Yard**.

Este algoritmo permite transformar expresiones infijas (como 3 + 4 * 2) a postfijas (3 4 2 * +) usando una **pila de operadores**, donde los operandos van directo a la salida, los operadores se apilan según su prioridad, y los paréntesis ayudan a controlar el orden. Al finalizar, los operadores restantes en la pila se agregan a la salida, permitiendo una evaluación sencilla.

Además, incrementamos nuestros conocimientos al utilizar **expresiones regulares** para diferenciar entre variables y números, y al aplicar correctamente los métodos propios de estructuras como **Stack**. En resumen, esta experiencia no solo reforzó conceptos conocidos, sino que también nos dio una nueva perspectiva sobre cómo estructurar y organizar código de forma más eficiente usando estructuras de datos.