Proyecto LPP-A

Profesor: Roberto García
Asignatura: Lenguajes y Paradigmas de la programación.
Estudiantes: Rodrigo San Juan - Martín Droguett

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.

Tokenizador

Tomemos por ejemplo la expresión X = 20 + 5

```java
@Override
public void tokenizador(String input) {
    // Divide la línea en nombre de variable y expresión, si existe un "=".
    String[] parts = input.split(regex:"=", limit:2);
    if (parts.length == 2) {
        variableAsignacion = parts[0].trim();     // Lado izquierdo antes del "="
        expresion = parts[1].trim();              // Lado derecho tras el "="
    } else {
        variableAsignacion = null;                // No hay asignación
        expresion = input.trim();                 // Toda la línea es expresión
    }

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

    // 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(newLength:0);
            }
            // Luego agrega el carácter actual como token individual
            tokens.add(Character.toString(c));
        }
    }
    // Al finalizar, si hay algo en el buffer, agrégalo
    if (actual.length() > 0) {
        tokens.add(actual.toString());
    }

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

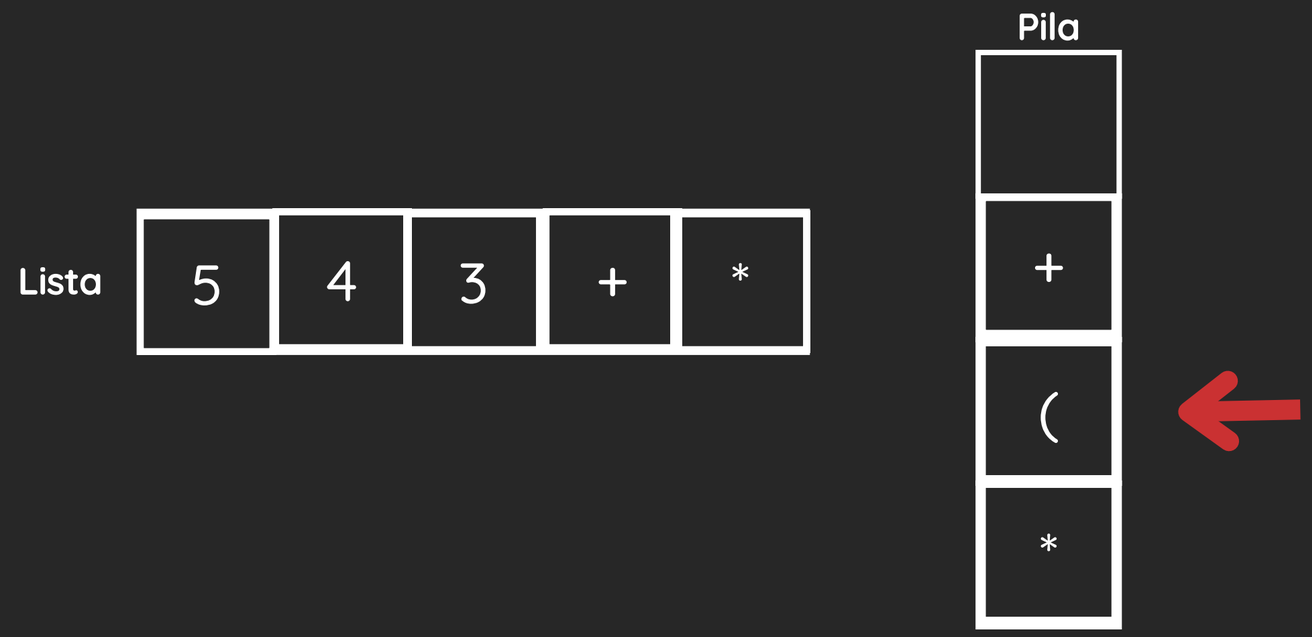
Convertir a Prefija

Tomemos por ejemplo la expresión X = (3 + 4) * 5

["(", "3", "+", "4", ")", "*", "5"] Invertimos → ["5", "*", ")", "4", "+", "3", "("] Invertimos los paréntesis → ["5", "*", "(", "4", "+", "3", ")"]

["5", "*", "(", "4", "+", "3", ")"]

Si es un paréntesis de cierre “)”, se van sacando elementos de la pila y agregando a la lista hasta encontrar el paréntesis de apertura correspondiente.



Se vacían los operadores restantes a la lista

Después la lista se invierte y que la notación prefija

[*, +, 3, 4, 5]

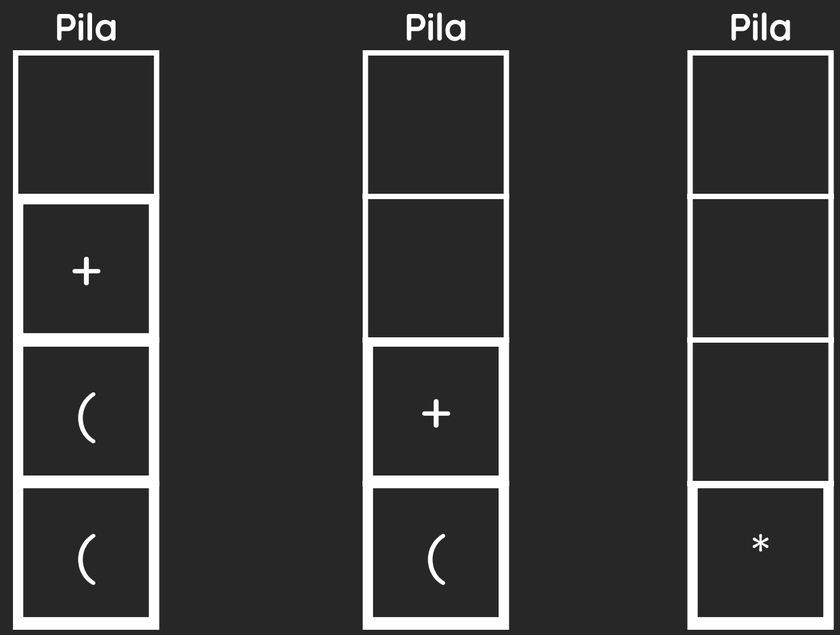
```java
@Override
public void convertirPrefija() {
    // Limpia la lista prefija de conversiones anteriores
    prefija.clear();
    // Paso 1: invierte la lista de tokens y cambia paréntesis
    Collections.reverse(expresionFormateada);
    for (int i = 0; i < expresionFormateada.size(); i++) {
  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

    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(anObject:"(")) {
            // Paréntesis de apertura → push
            stack.push(token);
        } else if (token.equals(anObject:")")) {
            // Paréntesis de cierre → pop hasta encontrar "("
            while (!stack.isEmpty() && !stack.peek().equals(anObject:"(")) {
                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(anObject:"(")
                    && 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);
}
```

Convertir a Postfija

Tomemos por ejemplo la expresión X = ((2 + 2) + 2) * 5

["(", "(", "2", "+", "2", ")", "+", "2", ")", "*", "5"]

Si es un paréntesis de cierre “)”, se van sacando elementos de la pila y agregando a la lista hasta encontrar el paréntesis de apertura correspondiente.



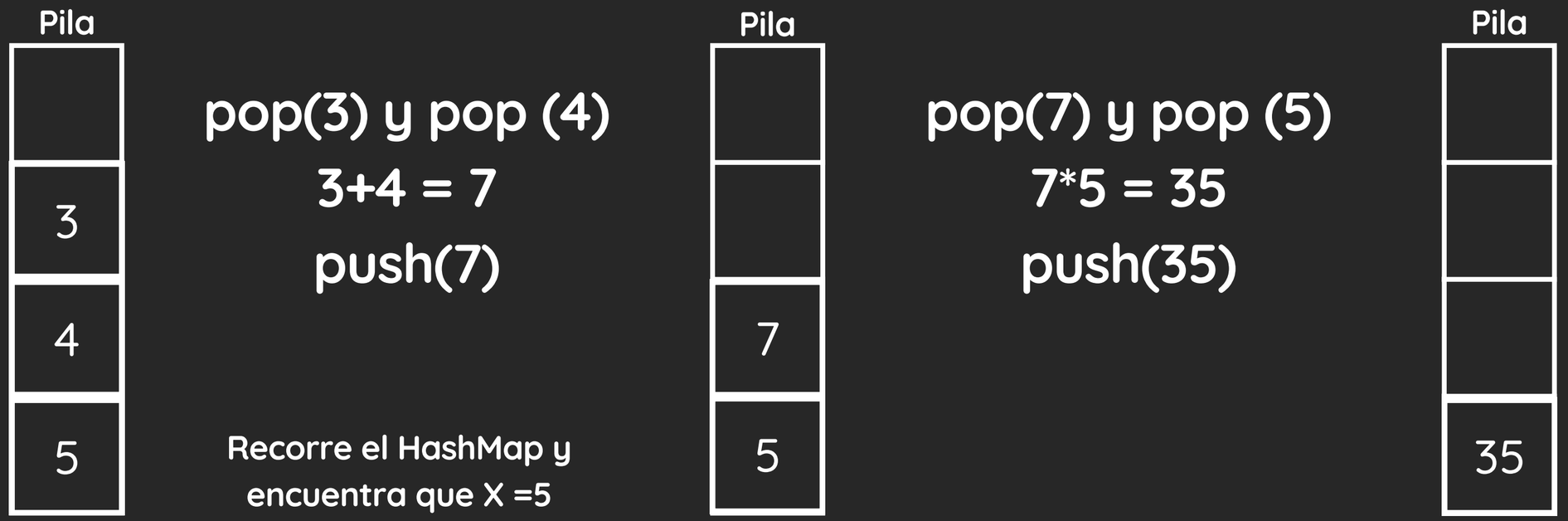
```java
@Override
public void convertirPostfija() {
    postfija.clear();
    Stack<String> stack = new Stack<>();
    // Algoritmo shunting-yard clásico para notación postfija
    for (String token : expresionFormateada) {
        if (token.matches(regex:"\\d+") || token.matches(regex:"[a-zA-Z]+"))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

Parser

Tomemos por ejemplo la expresión Z = (3 + 4) * X

Prefija: [*, +, 3, 4, X]

[*, +, 3, 4, X]



```java
@Override
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(s:"División por cero");
                    result = a / b;
                    break;
                case "^":
                    result = (int) Math.pow(a, b);
                    break;
                default:
                    throw new RuntimeException("Operador desconocido: " + token);
            }
            // Imprimir paso intermedio
            System.out.println(a + " " + token + " " + b + " = " + result);
            stack.push(result);
        }
    }

    // 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;
}
```

Pruebas

Potencia: 2 ^ 3 (Esperado: 8)

```
2^3
2 ^ 3 = 8
Prefija: [^, 2, 3]
Postfija: [3, 2, ^]
```

División: 5 / 0

```
5/0
Expresión inválida
```

División: 12 / 4 (Esperado: 3)

```
12/4
12 / 4 = 3
Prefija: [/, 12, 4]
Postfija: [4, 12, /]
```

División: 0 / 4 (Esperado: 0)

```
0/4
0 / 4 = 0
Prefija: [/, 0, 4]
Postfija: [4, 0, /]
```

Pruebas

Paréntesis: X = ((2 + 2) + 2) * 5 (Esperado: 30)

```
x = ((2+2)+2)*5
2 + 2 = 4
4 + 2 = 6
6 * 5 = 30
Variable asignada: x = 30
Variables: {x=30}
Prefija: [*, +, +, 2, 2, 2, 5]
Postfija: [5, 2, 2, 2, +, +, *]
```

Uso de variables: y = x * 3 (Esperado: y = 30 * 3 = 90)

```
y = x*3
30 * 3 = 90
Variable asignada: y = 90
Variables: {x=30, y=90}
Prefija: [*, x, 3]
Postfija: [3, x, *]
```

Conclusión

Este proyecto nos permitió explorar nuevas formas de programar mediante el uso de estructuras algorítmicas, fundamentales para implementar la conversión de expresiones en Java. Uno de los principales desafíos fue desarrollar la función parser, donde resultó necesario convertir la expresión infija a una notación más manejable. En nuestro caso, optamos por transformarla a notación prefija. Y tomamos como referencia el algoritmo de algoritmo Shunting-Yard