

# 17 | Aplicación de pilas y colas: Intérprete matemático

## Meta

Que el alumno utilice las estructuras de datos lineales pila y cola para resolver un problema de cómputo.

## Objetivos

Al finalizar la práctica el alumno será capaz de:

- Valorizar el uso de la estructura de datos correcta para implementar un algoritmo.
- Utilizar una interfaz de texto para interactuar dinámicamente con el usuario.
- Visualizar cómo se utiliza un intérprete de comandos para ejecutar las instrucciones dadas por el usuario.
- Pasar de notación infija a alguna de las otras dos.
- Evaluar operaciones en notación infija con precedencia, prefija y postfija.

## Antecedentes

Alterar el orden en que se ejecutan ciertas operaciones dentro de una expresión matemática, en el sistema que acostumbramos utilizar, puede alterar el resultado final, es decir, la propiedad de asociatividad no siempre se cumple entre operaciones distintas. Por ello es necesario introducir el concepto de *precedencia* de operaciones para establecer una forma canónica para realizarlas de tal forma que el resultado final sea único. En la Tabla 17.1 se asocia una precedencia mayor a aquellas operaciones que deben realizarse primero, a menos que un par de paréntesis altere este orden. Por este motivo, los paréntesis se tratan como un caso especial: las operaciones que estos encierren sólo podrán evaluarse hasta que se encuentre el paréntesis que cierra.

Precedencia	Operación	Símbolo
1	suma	+
1	resta	−
2	multiplicación	*
2	división	/
2	módulo	%

**Tabla 17.1** Precedencia de operaciones. A mayor precedencia, más pronto debe realizarse la operación.

Un ejemplo de este fenómeno se puede observar entre las operaciones *suma* y *multiplicación*, como se muestra a continuación:

**Ejemplo 17.1.** Sin precedencia de operadores, la expresión matemática siguiente se evalúa:

$$2 + 5 \times -2 = 7 \times -2 = -14 \tag{17.1}$$

con precedencia las multiplicaciones y divisiones se evalúan antes que las sumas y restas, esto es:

$$2 + 5 \times -2 = 2 - 10 = -8 \tag{17.2}$$

Para obtener un resultado equivalente a la Ecuación 17.1, pero con precedencia, debemos utilizar paréntesis:

$$(2 + 5) \times -2 = 7 \times -2 = -14 \tag{17.3}$$

Esta es una característica de la notación que utilizamos para las operaciones binarias, donde el operador se escribe entre los dos operandos, razón por la cual a esta notación se le llama *infija*.

Una calculadora científica o un intérprete matemático de comandos debe ser capaz de evaluar expresiones utilizando precedencia de operadores.

Existen notaciones alternativas que no requieren de la especificación del orden de precedencia. Se trata de las notaciones *prefija o polaca* y *posfija o polaca inversa*. En estas notaciones el operador se coloca a la izquierda o derecha de los operandos, respectivamente.

### Notación prefija o polaca

La estructura básica de la notación prefija para operadores binarios está dada por:

$\langle \text{operación} \rangle ::= \langle \text{operador} \rangle \langle \text{operando} \rangle \langle \text{operando} \rangle$   
 $\langle \text{operando} \rangle ::= \langle \text{número} \rangle \mid \langle \text{operación} \rangle$

Por ejemplo:

$$\begin{aligned}
 &+ \ 4 \ -2 = 2 \\
 &+ \ 4 \ - \ 8 \ 3 = + \ 4 \ 5 = 9 \\
 &+ \ 3 \ * \ 4 \ 2 = + \ 3 \ 8 = 11 \\
 &* \ + \ 3 \ 4 \ 2 = * \ 7 \ 2 = 14
 \end{aligned}$$

Aquí, el orden en que se ejecutan las operaciones depende de la posición que ocupan sus elementos: para ejecutar una operación se requiere evaluar primero sus dos operandos, si éstos a su vez consisten en otra operación, deben ser evaluados antes de proceder. El proceso es recursivo.

**Ejemplo 17.2.** La Ecuación 17.1 en notación prefija se ve:

$$\times \ + \ 2 \ 5 \ -2 \quad (17.4)$$

Mientras que la Ecuación 17.2 se convierte en:

$$+ \ 2 \ \times \ 5 \ -2 \quad (17.5)$$

## Notación postfija o sufija

Es similar a la prefija, pero el operador se escribe a la derecha de los operandos.

$\langle \text{operación} \rangle ::= \langle \text{operando} \rangle \langle \text{operando} \rangle \langle \text{operador} \rangle$   
 $\langle \text{operando} \rangle ::= \langle \text{número} \rangle \mid \langle \text{operación} \rangle$

Por ejemplo:

$$\begin{aligned}
 &4 \ -2 \ + = 2 \\
 &4 \ 8 \ 3 \ - \ + = 4 \ 5 \ + = 9 \\
 &3 \ 4 \ 2 \ * \ + = 3 \ 8 \ + = 11 \\
 &3 \ 4 \ + \ 2 \ * = 7 \ 2 \ * = 14
 \end{aligned}$$

### Actividad 17.1

Escribe algunos ejemplos de operaciones en notación infija utilizando operaciones con diferentes órdenes de precedencia. Escribe algunas variantes utilizando paréntesis para alterar el orden de evaluación. Ahora trata de expresar estas operaciones en las

notaciones prefija y postfija. Utilizarás estos ejemplos para probar tu código más adelante.

## Evaluación

Evaluar una expresión postfija es más sencillo que evaluar una expresión infija. Sólo se requiere de una pila como estructura auxiliar.

La expresión se lee símbolo por símbolo de izquierda a derecha. Obsérvese que en la notación postfija los operandos aparecen primero y el operador al final. Conforme vayamos leyendo los operandos se guardarán en la pila hasta que se lea la operación que se debe aplicar sobre ellos. El algoritmo en pseudocódigo se muestra en Algoritmo 3.

### Algoritmo 3 Evaluación postfija

```

1: function EVALÚAPOSFIJA(expresión)
2:   pila  $\leftarrow \emptyset$ 
3:   for all símbolo  $\in$  expresión do
4:     if símbolo  $\in$  Números then
5:       pila  $\leftarrow$  símbolo
6:     else if símbolo  $\in$  Operadores then
7:       operando2  $\leftarrow$  pila.pop()
8:       operando1  $\leftarrow$  pila.pop()
9:       pila  $\leftarrow$  operando1 operación(símbolo) operando2
10:    end if
11:  end for
12:  return pila.pop() ▷ El resultado queda al fondo de la pila
13: end function

```

Para la notación prefija basta con leer la expresión de derecha a izquierda y se aplica el mismo algoritmo. Obsérvese sin embargo, que los operandos salen de la pila en el orden contrario a este caso.

## Conversión de infija a postfija

Para evaluar una expresión infija, podemos convertirla a notación prefija o postfija. Este algoritmo requiere de dos estructuras auxiliares: una pila y una cola.

**Pila** La pila nos ayudará a mantener en memoria las operaciones que ya vió el sistema, pero que aún no puede evaluar, pues no se está seguro de si ya les toca o va otra primero, depende de la precedencia de sus sucesoras.

**Cola** En la cola se irán guardando los símbolos en el orden requerido por la notación postfija.

El algoritmo se muestra en Algoritmo 4.

---

**Algoritmo 4** Infija a postfija
 

---

```

1: function CONVIERTAPOSFIJA(expresiónInfija)
2:   pila  $\leftarrow \emptyset$ 
3:   cola  $\leftarrow \emptyset$ 
4:   for all símbolo  $\in$  expresiónInfija do
5:     if símbolo  $\in$  Números then
6:       cola  $\leftarrow$  símbolo
7:     else
8:       if símbolo es ) then
9:         while pila.peek() no es ( do
10:          cola  $\leftarrow$  pila.pop()
11:        end while
12:        pila.pop() ▷ Saca el paréntesis )
13:      else if símbolo es ( then
14:        pila  $\leftarrow$  símbolo
15:      else
16:        prec  $\leftarrow$  precedencia(símbolo)
17:        while pila  $\neq \emptyset$ , pila.peek()  $\neq$  ( y precedencia(pila.peek())  $\geq$  prec do
18:          cola  $\leftarrow$  pila.pop()
19:        end while
20:        pila  $\leftarrow$  símbolo
21:      end if
22:    end if
23:  end for
24:  while pila no está vacía do
25:    cola  $\leftarrow$  pila.pop()
26:  end while
27:  return cola
28: end function

```

---

## Desarrollo

Harás un programa que evalúe expresiones introducidas por el usuario en las tres notaciones.

1. Revisa el código auxiliar que acompaña esta práctica. Puedes compilarlo con `ant` y ejecutar la interfaz de texto con `ant run`.
2. Revisa el código y la documentación de la clase `Fija`. Implementa los métodos:

- `private static double evalúa(char operador, double operando1, double operando2)`
- `public static double evalúaPrefija(String[] tokens)`
- `public static double evalúaPostfija(String[] tokens)`

**TIP:** Para distinguir a los números de los operadores, revisa el funcionamiento de `Double.parseDouble(String s)`.

Observa que se incluyen algunas pruebas unitarias y la clase ya tiene un método `main` donde puedes probar tu código, puedes ejecutarlo con `ant run-fija`.

Ten cuidado, los usuarios de la interfaz pudieran enviar arreglos de cadenas con expresiones inválidas; por ejemplo, si en la terminal ingresan:

```
+ 3 9 8
```

Esto se convertirá en el arreglo de cadenas `{"+", "3", "9", "8"}`, que tiene un número de más. O como esta:

```
* + 3 9 - 3
```

que no tiene los operandos suficientes. Por este motivo, tus funciones deben lanzar la excepción correspondiente si detectan inconsistencias como esta, de lo contrario harán fallar a tu código. Detecta tantos casos como puedas.

**TIP:** Puedes programar tu algoritmo asumiendo que el usuario envió una entrada válida y verifica que funcione correctamente, luego podrás atrapar cualquier excepción y sustituirla por la tuya, esto contendrá la mayoría de los casos.

3. Revisa el código y la documentación de la clase `Infija`. Implementa los métodos:

- `private static int precedencia(char operador)`
- `public static String[] infijaASufija(String[] tokens)`
- `public static double evalúaInfija(String[] tokens)`

Recuerda estar al pendiente de las entradas inválidas.

Aquí está el método `main` que te permite elegir entre cualquiera de las notaciones, lo puedes ejecutar con `ant run`.

4. La interfaz de usuario en `Infija.java` ya realiza las funciones adecuadas, pero falta que imprima una guía para el usuario donde indique cómo utilizar la interfaz. Agrega estas instrucciones.
5. Agrega más pruebas unitarias para las funciones `evaluaPrefija`, `evaluaPostfija`, `infijaASufija` y `evaluaInfija`. Debes hacer al menos dos pruebas por método. Si tu conjunto de pruebas va más allá podrás ganar hasta un punto extra. Ya que este es un proyecto ese punto extra ayudará aún más con tu promedio final.

**TIP:** Observa las pruebas unitarias incluidas con tus prácticas. El `build.xml` ya está preparado para compilar y correr las pruebas; para que funcione los archivos

con pruebas se colocan dentro de un directorio llamado `test` y su nombre debe comenzar con `Test`, por ejemplo: `TestCalculadora.java`.

Este es un ejemplo de interacción con el usuario (no es necesario imprimir los *tokens*, pero aquí se utilizó para verificar lo que está haciendo el programa):

### Listado 17.1: Infija

```
Calculadora en modo notación infija
4 + 5 - ( 2 * 5 )
Tokens: [4, +, 5, -, (, 2, *, 5, )]
Sufija: [4, 5, +, 2, 5, *, -]
= -1.0
4.2 + 5 - ( 2.1 * -5 )
Tokens: [4.2, +, 5, -, (, 2.1, *, -5, )]
Sufija: [4.2, 5, +, 2.1, -5, *, -]
= 19.7
prefija
Cambiando a notación prefija
+ -4 - 8 2
[+, -4, -, 8, 2]
= 2.0
postfija
Cambiando a notación postfija
3 5 -7 * 3 -
[3, 5, -7, *, 3, -]
= -38.0
exit
```

## Preguntas

1. ¿Cuál es el orden de complejidad del algoritmo para evaluar una expresión en notación prefija? Justifica.
2. ¿Cuál es el orden de complejidad del algoritmo para pasar de notación infija a postfija? Justifica.