

Estructura de Datos y Algoritmos

ITBA 2024-Q2

Ejercicio

Agregar a la clase Evaluator el método **private String infijaToPostfija()** para implementar el parser de precedencia que utiliza la tabla antes diseñada.

Caso de Uso:

```
Evaluator e = new Evaluator();
```

```
System.out.println(e.evaluate());
```

```
System.out.println(e.evaluate());
```

```
System.out.println(e.evaluate());
```

Si se ingresa

- $2 - 3 * -3$ devuelve 11
- $2 / 4 / 2$ devuelve 0.25
- $2 \ 4 \ *$ no devuelve excepción

El algoritmo

A implementar !!! Bajar de campus y completar

El algoritmo

Posible
solución

```
private String infijaToPostfija()
{
    String postfija= "";
    Stack<String> theStack= new Stack<String>();

    while( scannerLine.hasNext() ) {
        String currentToken = scannerLine.next();

        if ( isOperand(currentToken) ) {
            postfija+= String.format("%s ", currentToken);
        }
        else {
            while ( !theStack.empty() && getPrecedence(theStack.peek(), currentToken) ) {
                postfija+= String.format("%s ", theStack.pop() );
            }

            theStack.push(currentToken);
        }
    }

    while ( !theStack.empty() ) {
        postfija+= String.format("%s ", theStack.pop() );
    }

    return postfija;
}
```

Para chequear correctitud y ante la presencia de métodos private ¿cómo hacemos?

Si bien la idea de Junit es chequear métodos públicos (del contrato) si quisiéramos hacerlo con el método importante `infixaToPostfija()` lo podemos hacer con alguna licencia...

- 1) public?
- 2) reflection

Sea la clase Sorpresa

```
public class Sorpresa {  
  
    private double f(){  
        return 35;  
    }  
  
    private double f(double param){  
        return param;  
    }  
}
```

```
class Testing {  
  
    @Test  
    void test1() throws NoSuchMethodException, SecurityException,  
        IllegalAccessException, IllegalArgumentException,  
        InvocationTargetException {  
  
        Sorpresa sorpresaInstance = new Sorpresa();  
  
        Method myMethod = Sorpresa.class.getDeclaredMethod("f");  
        myMethod.setAccessible(true);  
  
        double result = (Double) myMethod.invoke(sorpresaInstance);  
  
        assertEquals( 35, result);  
    }  
}
```

Sea la clase Sorpresa

```
public class Sorpresa {  
  
    private double f(){  
        return 35;  
    }  
  
    private double f(double param){  
        return param;  
    }  
}
```

```
@Test  
void test2() throws NoSuchMethodException, SecurityException,  
    IllegalAccessException, IllegalArgumentException,  
    InvocationTargetException {  
    Sorpresa sorpresaInstance = new Sorpresa();  
  
    Method myMethod = Sorpresa.class.getDeclaredMethod("f", double.class);  
    myMethod.setAccessible(true);  
  
    double result = (Double) myMethod.invoke(sorpresaInstance, 34.5);  
  
    assertEquals( 34.5, result);  
}
```


Por otro lado, cómo inyectar desde Junit algo en consola input?

```
@Test
void test() {
    // inyecto en la standard input
    String input = "15 + 3";
    InputStream inputStream = new ByteArrayInputStream(input.getBytes());
    System.setIn(inputStream);

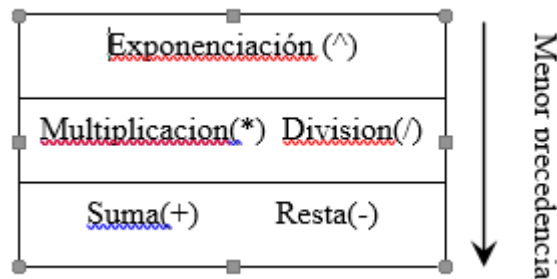
    Evaluator myEval = Evaluator();

    double rta = myEval.evaluate();
    assertEquals( 18, rta);
}
```

Ejercicio

Ahora vamos a incorporar el operador \wedge

La precedencia entre operadores está dado por:



Pero atención, el operador **es asociativo a derecha!!!!**.

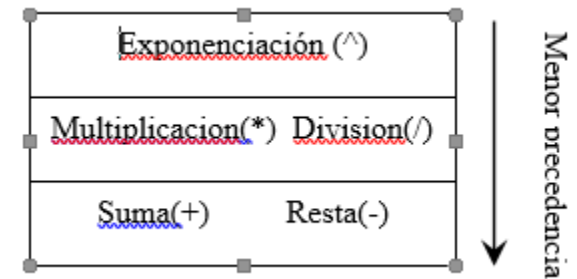
Ver

https://en.wikipedia.org/wiki/Order_of_operations

(Serial Exponentiation)

$$a^{b^c} = a^{(b^c)}$$

Completemos la siguiente tabla sabiendo que $^$ es asociativa a derecha. Lo que representa cada celda es la precedencia es si el tope de la pila tiene mayor precedencia que el elemento actual.

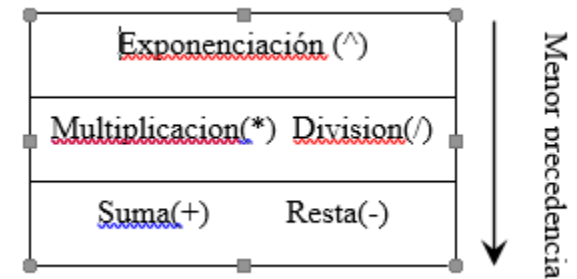


Elemento que está
en el tope
de la pila (previo)

Elemento que está siendo analizado (actual)

	+	-	*	/	^
+	true	true	false	false	
-	true	true	false	false	
*	true	true	true	true	
/	true	true	true	true	
^					

Completemos la siguiente tabla sabiendo que $^$ es asociativa a derecha. Lo que representa cada celda es la precedencia es si el tope de la pila tiene mayor precedencia que el elemento actual.



Elemento que está
en el tope
de la pila (previo)

Elemento que está siendo analizado (actual)

	+	-	*	/	^
+	true	true	false	false	False
-	true	true	false	false	false
*	true	true	true	true	False
/	true	true	true	true	false
^	True	True	True	true	false

Modificar la precedencia agregando el Nuevo operador.

Chequear con las expresiones:

$3 + 10 * 2 / 1$ (deberían obtener $3 + 10 * 2 / 1$ y evalúa a 23

$13 ^ 2 - 1 * 7$ (deberían obtener $13 ^ 2 - 1 * 7$ y evalúa a 162

$5 ^ 2 ^ 3 - 1$ (debería obtenerse $5 ^ 2 ^ 3 ^ 1 -$ y evalúa a 390624

Ejercicio

Implementar en Java el parser de precedencia de operadores que transforme una expresión de notación infija a postfija usando la tabla de precedencia discutida, incorporando la exponenciación.

Elemento que está en el tope de la pila (previo)	Elemento que está siendo analizado (actual)					
	+	-	*	/	^	
	+	true	true	false	false	False
	-	true	true	false	false	false
	*	true	true	true	true	False
	/	true	true	true	true	false
	^	True	True	True	true	false

El algoritmo

Posible solución:

- Solo agregar a la tabla de precedencia el $^$
- actualizar `eval()` para que considere $^$ como válido, devolviendo `Math.pow(a, b)`. No olvidar que `matches` también acepta el nuevo $^$

Ejemplo:

- $2 - 3^{-3}$ devuelve 1.9629
- 2^{4^2} devuelve 65536 (y no 256)
- $3 + 10 * 2 / 1$ (toPostfija() da $3\ 10\ 2\ *\ 1\ /\ +$) y evalúa a 23
- $13^2 - 1 * 7$ (toPostfija() da $13\ 2\ ^\ 1\ 7\ *\ -$) y evalúa a 162
- $5^{2^3} - 1$ (toPostfija() da $5\ 2\ 3\ ^\ ^\ 1\ -$) y evalúa a 390624

Algoritmo ampliado

Ahora bien, ¿cómo incorporamos los paréntesis en las expresiones?

Los mismos no deben aparecer en la salida ya que no son necesarios en las expresiones postfijas (ni en prefijas).

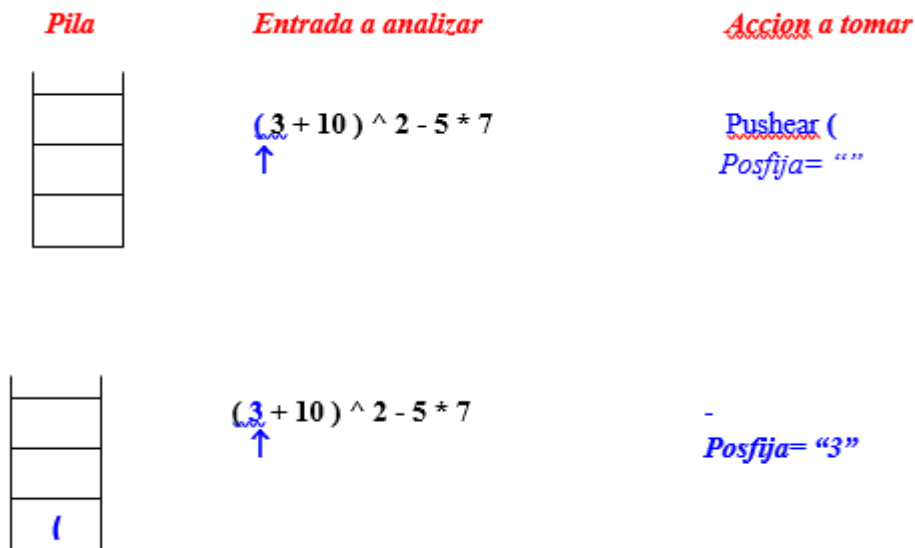
Extensión del algoritmo: ()

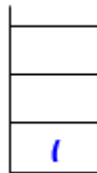
Una forma sencilla es considerarlos como operadores especiales, es decir:

- Si el operador current es un un “(”, el mismo debe postergarse hasta que aparezca “)”. Es decir, completar la tabla para que se lo pushee siempre.
- Si el operador current es un “)” el mismo debe sacar todos los operadores de la pila y concatenarlos en el string de salida hasta encontrar el “(” que aparea con él. Cuando en el tope aparezca el “(” debe sacarlo del tope de la pila pero no concatenarlo (ya que los paréntesis no van a la expresión postfija). Completar la tabla para manejar esta situación y colocar que la precendencia entre “(” y “)” es false para manejarla como un caso especial => sino se vacía la pila!

Ej: ((4 - 3) * 2)

- Ejemplo: Supongamos que tenemos la expresión infija $(3 + 10) ^ 2 - 5 * 7$

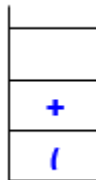




$$(3 + 10)^2 - 5 * 7$$

↑

Push +
Posfija = "3"



$$(3 + 10)^2 - 5 * 7$$

↑

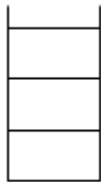
-
Posfija = "3 10"



$$(3 + 10)^2 - 5 * 7$$

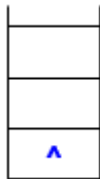
↑

Pop + (concatenarlo)
Pop (
Posfija = "3 10 +"



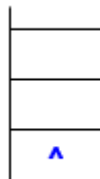
$(3 + 10) \wedge 2 - 5 * 7$
 \uparrow

Push \wedge
 Posfija = "3 10 +"



$(3 + 10) \wedge 2 - 5 * 7$
 \uparrow

-
 Posfija = "3 10 + 2"



$(3 + 10) \wedge 2 - 5 * 7$
 \uparrow

Pop \wedge (concatenarlo)
 Push -
 Posfija = "3 10 + 2 ^"

-

$$(3 + 10)^{\wedge} 2 - 5 * 7$$

↑

-

Posfija = "3 10 + 2 ^ 5"

-

$$(3 + 10)^{\wedge} 2 - 5 * 7$$

↑

Pushear *

Posfija = "3 10 + 2 ^ 5"

*
-

$$(3 + 10)^{\wedge} 2 - 5 * 7$$

↑

-

Posfija = "3 10 + 2 ^ 5 7"

*
-

Fin de la entrada

Pop * (concatenarlo)
 Pop - (concatenarlo)
Posfija = "3 10 + 2 ^ 5 7 * -"

Pila vacía

Ejercicio en papel, mostrar el pasaje a postfija y la pila instantánea en cada una de las siguientes expresiones infijas

$$3 * ((5 - 10.2) / 0.5) - 2$$

Rta 3 5 10.2 - 0.5 / * 2 -

Y evalúa como -33.19999

$$3 * ((5 - 10.2 / 0.5) -$$

Rta: Error falta)

$$3 * (5 - 10.2) / 0.5) - 2$$

Rta: Error falta (

Completemos la siguiente tabla con el agregado de paréntesis. Lo que representa cada celda es la precedencia es si el tope de la pila tiene mayor precedencia que el elemento actual.

Elemento que está siendo analizado (actual)								
Elemento que está en el tope de la pila (previo)		+	-	*	/	^	()
	+	True	True	False	False	False		
	-	True	True	False	False	False		
	*	True	True	True	True	False		
	/	True	True	True	True	False		
	^	True	True	True	True	False		
	(

Completemos la siguiente tabla con el agregado de paréntesis. Lo que representa cada celda es la precedencia es si el tope de la pila tiene mayor precedencia que el elemento actual.

Elemento que está en el tope de la pila (previo)	Elemento que está siendo analizado (actual)							
		+	-	*	/	^	()
	+	True	True	False	False	False	False	True
	-	True	True	False	False	False	False	True
	*	True	True	True	True	False	False	True
	/	True	True	True	True	False	False	True
	^	True	True	True	True	False	False	True
	(False	False	False	False	False	False	false

Ejercicio

Implementar en Java el parser de precedencia de operadores que transforme una expresión de notación infija a postfija usando la tabla de precedencia discutida, incorporando también el manejo de paréntesis.

		Elemento que está siendo analizado (actual)						
			+	-	*	/	^	(
Elemento que está en el tope de la pila (previo)	+	True	True	False	False	False	False	True
	-	True	True	False	False	False	False	True
	*	True	True	True	True	False	False	True
	/	True	True	True	True	False	False	True
	^	True	True	True	True	False	False	True
	(False	False	False	False	False	False	false

```

private String infijaToPostfija()
{
    Scanner auxi = new Scanner(System.in).useDelimiter("\\n");
    String expLine= auxi.nextLine();

    Scanner scannerLine = new Scanner(expLine).useDelimiter("\\s+");

    Stack<String> theStack= new Stack<>();
    String postfijaOutput="";

    while( scannerLine.hasNext() )
    {
        String currentToken= scannerLine.next();

        if (isOperand(currentToken))
        {
            postfijaOutput+= String.format("%s ", currentToken);
        }
        else
        {
            while (! thestack.empty() && getPrecedence(theStack.peek(), currentToken))
            {
                postfijaOutput += String.format("%s ", theStack.pop() );
            }

```

Agregado 1

```

        theStack.push(currentToken);
    }
}

while (! thestack.empty())
{
    postfijaOutput+= String.format("%s ", theStack.pop() );
}

return postfijaOutput;

```

Agregado 2

Más aún

Podríamos considerar una nueva extensión del algoritmo, donde los operandos no sean solo constantes sino variables previamente definidas.

(hacerlo Uds completando el TP)

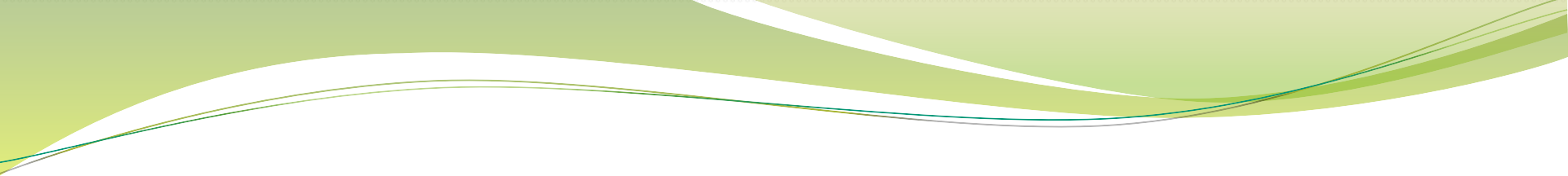
Caso de Uso:

```
private static Map<String, Integer> vbles = new HashMap<String, Double>()  
{ { put("nro1", 0.2); put("x", -2.0); put("y", 2.0) ; } };
```

Se invoca igual que antes.

Si ingreso

$(nro1 + 3) * (x - -2 + y)$ se obtendría el valor 6.4



Completar Evaluator para que maneje variables en las expresiones.

Tip:

En el `infijaToPostija` además del método `isOperand(currentToken)` codificar el método `isVariable(currentToken)` y devolver el valor del binding o error si no fue la variable predefinida



Posible implementación

P

```
while( scannerLine.hasNext() )
{
    String currentToken= scannerLine.next();

    if (isOperand(currentToken))
    {
        postfijaOutput+= String.format("%s ", currentToken);
    }
    else
    {
        while (! theStack.empty() && getPrecedence(theStack.peek(), currentToken))
        {
            postfijaOutput += String.format("%s ", theStack.pop() );
        }
        if (currentToken.equals("("))
            // quedo un ( en la pila?
            if (! theStack.empty() && theStack.peek().equals("(") )
                theStack.pop();
            else
                throw new RuntimeException("( missing");
        else
            theStack.push(currentToken);
    }
}
etc.
```