Estructura de Datos y Algoritmos

ITBA 2024-Q2

Parser de Precedencia de Operadores

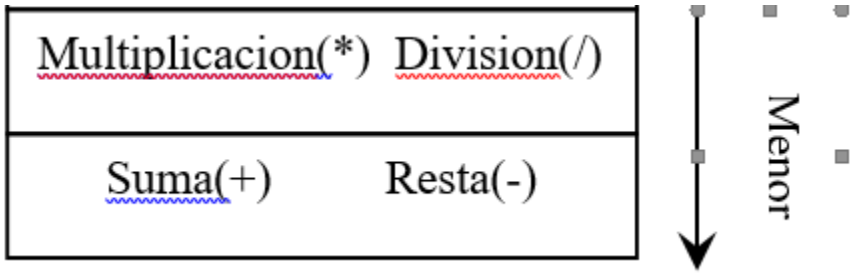
Algoritmo para transformar expresión en notación infija a expresión en notación postfija.

Idea:

- cada vez que aparezcan varios operadores se consultará un tabla que indique cuál se evalúa primero.

- Si dos operadores tienen la misma precedencia, se utiliza la regla de asociatividad para saber cuál se evalúa primero.

Ejemplo:



La asociatividad de esas 4 operaciones es de izquierda a derecha.

Por lo tanto si tenemos A - B + C en la expresión postfija primero tiene que aparecer el – (ya que se evaluará primero A – B).

Eso es debido a que +- tienen la misma precedencia, pero como es asociativo a izquierda si llega un "+" y había previo un "-", entonces el previo – se evalúa antes que el +.

Ejemplo:

Si tenemos A + B * C / D entre el “+” y el “*”, el segundo debe aparecer antes que el “+” en la expression postfija

Pero entre el “*” y el “/” el “*” también tiene que aparecer antes porque aunque tiene igual precedencia, la asociatividad a izquierda indica que se evaluará primero el “*” y luego el “/”.

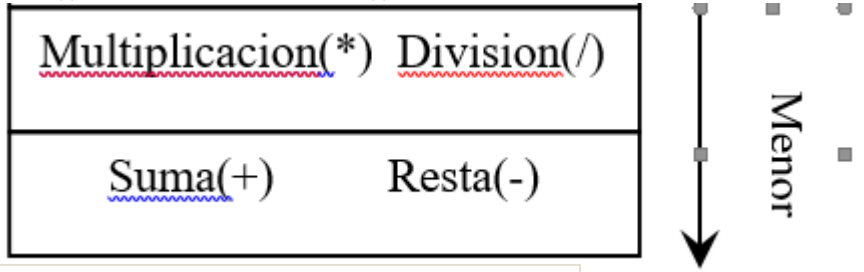
Resumiendo, deberían aparecer en la expression postfija: primero el *, luego el / y finalmente el +

La expression original es equivalente a : A + ((B * C) / D) .

Salvo que se quisiera cambiar este comportamiento, no hace falta colocar paréntesis en la expresión infija.

Completemos la siguiente tabla sabiendo que esas operaciones son asociativas a izquierda. Lo que representa cada celda es la precedencia de si el tope de la pila tiene mayor precedencia que el elemento actual.

A * C - D



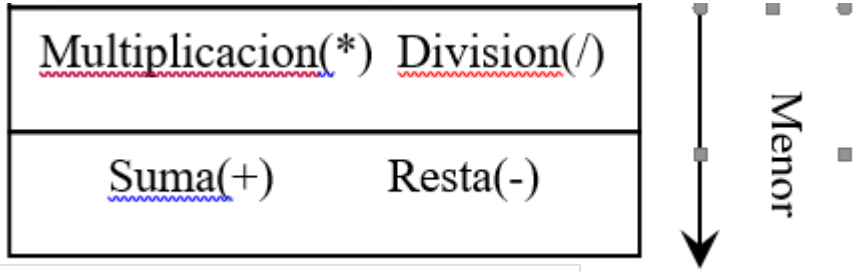
Elemento que está siendo analizado (actual)

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

	+	-	*	/
+				
-				
*	true	true		
/	true	true		

Completemos la siguiente tabla sabiendo que esas operaciones son asociativas a izquierda. Lo que representa cada celda es la precedencia de si el tope de la pila tiene mayor precedencia que el elemento actual.

A - C + D



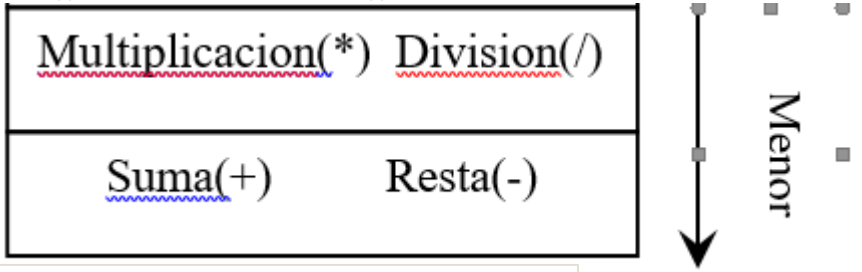
Elemento que está siendo analizado (actual)

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

	+	-	*	/
+				
-				
*	true	true		
/	true	true		

Completemos la siguiente tabla sabiendo que esas operaciones son asociativas a izquierda. Lo que representa cada celda es la precedencia de si el tope de la pila tiene mayor precedencia que el elemento actual.

A + C / D



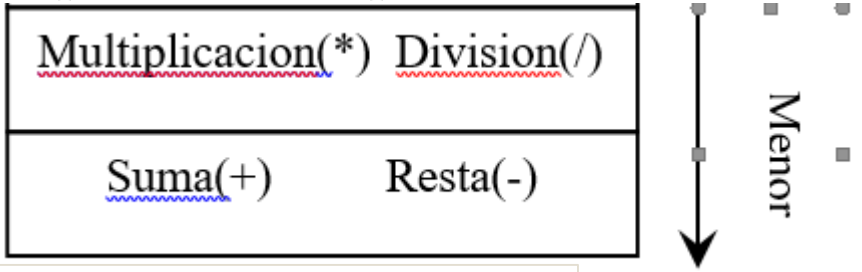
Elemento que está siendo analizado (actual)

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

	+	-	*	/
+	true	true		
-	true	true		
*	true	true		
/	true	true		

Completemos la siguiente tabla sabiendo que esas operaciones son asociativas a izquierda. Lo que representa cada celda es la precedencia de si el tope de la pila tiene mayor precedencia que el elemento actual.

A * C / D

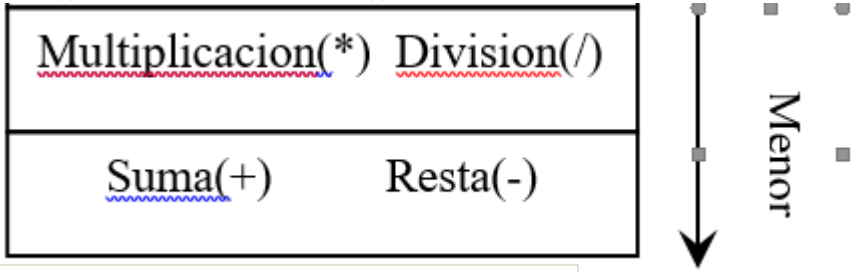


Elemento que está siendo analizado (actual)

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

	+	-	*	/
+	true	true	False	false
-	true	true	False	false
*	true	true		
/	true	true		

Completemos la siguiente tabla sabiendo que esas operaciones son asociativas a izquierda. Lo que representa cada celda es la precedencia de 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
-	true	true	False	false
*	true	true	True	True
/	true	true	True	true

Algoritmo infija=>postfija

Cada operando de la expresión infija se copia en la expresión postfija.

Cuando aparece un operador hay que analizar precedencia respecto del resto de los previos operadores, por lo tanto los casos se reducen a chequear la precedencia entre el **tope de la pila** y el **operador current**:

- Si la pila está vacía, se "**pushea" el operador current** ya que no se lo puede comparar con nada porque es el primero de la subexpresión.

- Si la pila no está vacía:
  - Si el **tope de la pila tiene mayor precedencia** que el **operador curren**t, entonces se realizar "**pop**" del operador en la pila y se lo copia en la expresión postfija **hasta que** se acabe la pila o quede en ella uno de menor precedencia que el **operador current**. **Se pushea al operador current**, ya que hay que postergar su acción hasta que aparezca otro operador.
  - Si el **tope de la pila tiene menor precedencia que** el **operador current no se puede ir todavía… Se pushea al operador current**, ya que hay que postergar su acción hasta que aparezca otro operador.

- Cuando se terminó de analizar la expresión infija, se **"popean"** todos los operadores de la pila y se copian en la expresión postfija.

Ejemplo: Supongamos que tenemos la expresión infija **3 + 10 * 2 / 1**

El algoritmo funciona así:

Pila	*Entrada a analizar*	*Accion a tomar*
	**3** + 10 * 2 / 1 ↑	Concatenar 3 ***Posfija*= "3"**

3 + 10 * 2 / 1
↑

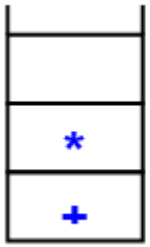
Pushear +
Posfija= "3"

3 + 10 * 2 / 1
↑

Concatenar 10
Posfija= "3 10"

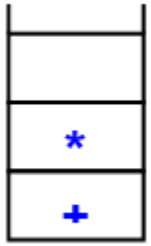
3 + 10 * 2 / 1
↑

Pushear *
Posfija= "3 10"



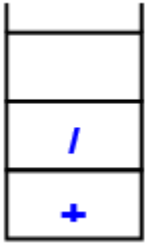
3 + 10 * 2 / 1
↑

Concatenar 2
***Posfija*= "3 10 2"**



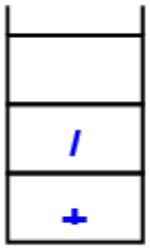
3 + 10 * 2 / 1
↑

Pop * (concatenarlo)
Pushear /
***Posfija*= "3 10 2 *"**



3 + 10 * 2 / 1
↑

Concatenar 1
***Posfija*= "3 10 2 * 1"**



Fin de la entrada

Pop / (concatenarlo)
Pop + (concatenarlo)
*Posfija= "3 10 2 * 1 / +"*

O sea,
3 + 10 * 2 / 1 => 3 10 2 * 1 / +

Aplicar el algoritmo (en papel) para pasar a notación postfija las siguientes expresiones infijas:

Notación Infija	*Notación Postfija*
A + B * C	
A + B - C	

Notar que el pasaje de infija a postfija no detecta muchos errores. La mayoría se detectan en tiempo de ejecución al **evaluar** la expresión.
Ej: 4 * / 2 no da error, genera 4 * 2 /

Ejercicio

¿Cómo implementar en Java la tabla que utiliza el parser de precedencia de operadores ?

Posibilidad Opcion 1

```
// opcion 1
 private static Map<String, Integer> mapping = new HashMap<String, Integer>()
 {   { put("+", 0); put("-", 1); put("*", 2); put("/", 3);  }
 };

private static boolean[][] precedenceMatriz=
{   { true, true, false, false},
    { true, true, false, false},
    { true, true, true, true},
    { true, true, true, true},
};

private boolean getPrecedence(String tope, String current)
{
    Integer topeIndex;
    Integer currentIndex;

    if ((topeIndex= mapping.get(tope))== null)
        throw new RuntimeException(String.format("tope operator %s not found", tope));

    if ((currentIndex= mapping.get(current)) == null)
        throw new RuntimeException(String.format("current operator %s not found", current));

    return precedenceMatriz[topeIndex][currentIndex];
}
```

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

Posibilidad Opcion 2

```
// opcion 2: asumo que _ no es parte de ningun operador posible
private static Map<String, Boolean> precendeceMap= new HashMap<String, Boolean>()
{   {
        put("+_+", true); put("+_-", true); put("+_*", false); put("+_/", false);
        put("-_+", true); put("-_-", true); put("-_*", false); put("-_/", false);
        put("*_+", true); put("*_-", true); put("*_*", true); put("*_/", true);
        put("/_+", true); put("/_-", true); put("/_*", true); put("/_/", true);
    } };

private final static String extraSymbol= "_";

private boolean getPrecedence(String tope, String current)
{
    Boolean rta= precendeceMap.get(String.format("%s%s%s", tope, extraSymbol, current));
    if (rta == null)
        throw new RuntimeException(String.format("operator %s or %s not found", tope, current));

    return rta;
}
```

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