¿Cómo evaluar expresiones matemáticas en el computador?*

MSc. Alexander Borbón Alpízar**
aborbon@itcr.ac.cr

2005

Resumen

En este artículo se muestra una forma de programar un evaluador de expresiones matemáticas en JAVA. El programa se construye paso a paso y se explican detalladamente las partes más importantes del mismo. El evaluador consta de dos partes o módulos, el primero se encarga de convertir la expresión digitada a notación postfija que es más sencilla para el computador; el segundo es el que evalúa la expresión que se obtuvo en un valor específico. Para poder comprender y reescribir este programa se necesita tener conocimientos básicos en la programación en JAVA, sin embargo, se explicará el uso de varias primitivas utilizadas y de algunos conceptos básicos de programación.

Palabras Clave: Programación, JAVA, funciones, expresiones matemáticas, software matemático.

1. Introducción

El objetivo principal de este artículo es mostrar una manera de programar un evaluador de expresiones matemáticas en JAVA. En el artículo se explicará tanto la teoría matemática sobre la evaluación de expresiones como la creación del programa mismo.

El evaluador se realiza en JAVA [4], ya que es un lenguaje de programación muy accesible (es "open source", que significa que puede ser utilizado de forma gratuita) y permite acoplarse fácilmente con un navegador de Internet. Además, este programa tiene clases con muchas funciones que permiten trabajar con texto, lo que hace más sencillo el trabajo. Sin embargo, las ideas pueden ser trasladadas a otros lenguajes de programación (de hecho, al final del artículo se encuentra la versión en JAVA y un programa similar para Visual Basic).

El evaluador consta de dos módulos, el primero se encarga de revisar que la expresión esté bien digitada y la "traduce" a una expresión más simple para que la computadora pueda "entenderla". En este caso, la expresión se pasará a notación postfija[1], esta notación lo que hace es escribir el operador después de los números que opera; por ejemplo, una expresión como $x + 1$ el programa la traduciría como "$x\ 1\ +$".

El segundo módulo lo que hace es evaluar un valor dado en la expresión en notación postfija (ya sea la última que se digitó o cualquier expresión en notación postfija). Este módulo también tomará en cuenta cuando se evalúa en un valor en donde la expresión no tiene sentido en los números reales (división por

*Fecha de recibido: Octubre, 2005. Fecha de aceptación: Abril, 2006.
**Profesor de matemáticas en el Instituto Tecnológico de Costa Rica.
[1]Conocida también como notación polaca invertida.

cero o raíz par de un número negativo); en estos casos se lanzará una excepción (un error no muy grave o una conclusión anormal en JAVA).

Para la programación de estos módulos se va a suponer que la ecuación que digita el usuario está bien escrita y en la última sección se explicará la idea general para poder verificarlo.

Este artículo está dirigido a profesores de matemáticas y programadores que quieran aprender a realizar un evaluador de expresiones matemáticas para sus proyectos o que les gustaría conocer un poco sobre la técnica que hay atrás de la programación de un evaluador.

Aunque en este momento el programador ansioso debe querer iniciar programando de una vez, se necesita primero comprender bien la teoría básica, así que antes de empezar a trabajar en JAVA se va a explicar la notación postfija, cómo se evalúa en ella y la procedencia de los operadores.

2. Notación Postfija

El primer módulo del programa se encargará de traducir la expresión que digita el usuario (en notación infija) en una expresión más simple, esta segunda expresión se dice que está en notación postfija; esta forma de escribir una expresión matemática tiene la ventaja que se evalúa de forma lineal por lo que es más sencillo para una computadora "entenderlo".

Al inicio talvez se ve un poco complejo, pero no es así; lo que hace el lenguaje es colocar primero los números con los que va a operar y luego escribe la operación, por ejemplo $a + b$ se escribiría "$a\ b\ +$"; $(5 - 8) * 4$ se escribiría "5 8 - 4 *", otros ejemplos son:

Notación infija	Notación postfija
$\text{sen}(x)$	x sen
$1 + 3 * 4$	1 3 4 * +
$\text{sen}(9 + x)$	9 x + sen

Notación infija	Notación postfija
$3 * 6$	3 6 *
$4 - \text{sen}(x)$	4 x sen -
$\text{sen}(x^2) + 4 * x$	x 2 ^ sen 4 x * +

Para evaluar una expresión en notación infija en un valor específico para x se deben seguir las reglas matemáticas para la prioridad de las operaciones:

1. Las potencias tienen prioridad sobre cualquier operación
2. La multiplicación y la división tienen prioridad sobre la suma y la resta.
3. Si se presenta un paréntesis, se deben realizar primero las operaciones dentro de éste. Si hay un paréntesis dentro de otro tiene prioridad el paréntesis interno.

Por el contrario, en la notación postfija siempre se trabaja de izquierda a derecha; note además que la notación postfija no tiene paréntesis por la forma lineal en que se lee. Comparemos un ejemplo en donde se evalúa un valor en una expresión utilizando estas dos notaciones.

2.1. Evaluación en notación infija y postfija

Tomemos un ejemplo sencillo como $4 + x^3$ cuando $x = 2$, esta expresión se escribe en la notaciones anteriores de la siguiente manera:

Notación infija: $4 + x$ ^ 3

Notación postfija: 4 x 3 ^ +

Para evaluar $4 + x^3$ en notación infija primero se debe tomar el 2 (en vez de la x), elevarlo al cubo y luego sumarle 4; se nota que esto no está en forma lineal, es decir, se debe ir primero a la segunda parte de la expresión, evaluarla y luego sumarle al resultado el cuatro.

Es muy complejo hacer que un programa evalue de esta forma. Si la expresión "$4 + x^3$" se traduce a la forma "4 x 3 ^ +", entonces se convierte en una expresión más sencilla; para evaluar esta expresión en $x = 2$, el algoritmo es leer el texto de izquierda a derecha, si se encuentra un número lo apila[2] y las operaciones se irán realizando conforme aparezcan.

Así, en el ejemplo "4 x 3 ^ +" se toma el 4 y se mete en una pila de números, el segundo valor que se toma es la x, en vez de ésta introducimos el 2 (que es el valor que estamos evaluando), luego se toma el 3 y se mete nuevamente en la pila de números, es decir, tenemos una pila de números como sigue

3
2
4

Ahora sigue una potencia, por lo que se toman los dos últimos números de la pila (recuerde que tomamos los dos de arriba: 2 y 3) y se realiza la potencia $2^3 = 8$, así, el 2 y el 3 se sustituyen en la pila por 8, se obtiene la pila

8
4

Por último, sigue un signo de suma, éste también se realiza con dos números, por lo que se toman el 4 y el 8, el resultado es $4 + 8 = 12$ y en la pila queda un 12; como ya se acabó la expresión entonces el resultado es 12.

Aunque a primera vista esta forma de evaluar parece más compleja, para un programa no lo es, ya que se siguen los pasos en forma lineal (la expresión se lee de izquierda a derecha), la algoritmo estaría compuesto por tres reglas:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce el resultado en la pila.
3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Con este ejemplo se observa que evaluar una expresión en notación postfija es más sencillo que hacerlo en notación infija, por lo que lo primero que se hará el programa es "traducir" la expresión que digite el usuario a dicha notación. Veamos primero la teoría sobre cómo se hará esto.

[2]En computación, existe una estructura de datos que utiliza la metáfora de una pila de objetos, por ejemplo, piense en una pila de documentos, si uno agrega un nuevo documento lo coloca encima de la pila y si uno toma un documento lo toma de arriba de la pila; es decir, una pila es una estructura de datos en donde el último objeto en entrar es el primero en salir. Como referencia ver [1].

3. Traducción a notación postfija

Para la traducción a notación postfija se utilizarán dos pilas, una en donde se guardarán los números y otra para los operadores y los paréntesis[3]. Aunque se dice que una de las pilas maneja números, esto no es muy cierto, en realidad las dos pilas se trabajarán con texto (*String*), esto permite concatenar varias tiras fácilmente y se pueden manejar expresiones que no son números como si lo fueran.

Para la traducción del lenguaje natural a notación postfija es fundamental manejar la prioridad de las operaciones y el uso de paréntesis.

Como se dijo al inicio del artículo, para evaluar una expresión matemática se deben seguir las siguientes reglas:

1. Primero se evalúan las potencias.
2. A continuación siguen las multiplicaciones, las divisiones y el resto de la división entera (%).
3. Por último se realizan las sumas y las restas.
4. Si hay paréntesis, se hace primero la expresión que está dentro del paréntesis interno.

Por esto, para la prioridad se les dará a las operaciones los siguientes valores:

Operador	Prioridad
+, -	0
*, /, %	1
^	2

Por lo que en nuestro programa ocupamos una función como la que sigue:

```
private int prioridad(char s) {
   if (s=='+' || s=='-')
      return 0;
   else if (s=='*' || s=='/' || s=='%')
      return 1;
   else if (s=='^')
      return 2;

   return -1;
} //Fin de la funcion prioridad
```

Esta función recibe un caracter *s*, dependiendo de la operación que represente este caracter se devuelve el valor de la prioridad correspondiente, si recibe un operador que no corresponde con las operaciones se devuelve -1.

Otro punto importante es que todas las funciones que reciben un parámetro (seno, coseno, tangente,...) se manejan como un paréntesis que abre "(", el usuario debe digitarlas con ese paréntesis, es decir: "sen(", "cos(", "tan(", ... Se manejan así porque cuando se digita el paréntesis que cierra ")", este paréntesis

[3]Existe un algoritmo para este programa que utiliza una sola pila, sin embargo, consideramos que es más sencillo y didáctico con dos.

saca todo lo que hay en la pila hasta que encuentra el de apertura o una función (lo que quiere decir que en términos prácticos funcionan igual con una ligera diferencia que luego se verá).

A continuación se muestran varios ejemplos para observar el algoritmo que se debe seguir para traducir una expresión de notación infija a postfija.

3.1. Una expresión simple

Como primer ejemplo se va a traducir una expresión simple como 3*x+4, la prioridad indica que primero se tiene que hacer la multiplicación y luego la suma; para esto, recuerde que la función le asigna prioridad 0 a la suma y 1 a la multiplicación y se dará la regla que una prioridad inferior saca de la pila cualquier operación con prioridad igual o superior a ella.

Dado esto, se pueden notar que:

1. Los operadores "+" y "-" son los que tienen menor prioridad (0), por lo que siempre sacarán todos los operadores precedentes.
2. Los operadores "*", "/" y "%" sacan a la potencia "^" y a ellos mismos.
3. El operador "^" es el que tiene mayor prioridad, no saca a nadie, ni siquiera a sí mismo ya que la prioridad para realizar las potencias es de derecha a izquierda (contrario a todos las demás operaciones matemáticas), es decir: $2^{3^4} = 2^{(3^4)}$ o de manera escrita 2^3^4=2^(3^4)

Por lo tanto, se van a sacar los elementos de la expresión 3*x+4; primero se toma el 3 y se introduce en la pila de números, luego se toma el * y se mete en la pila de operadores, se obtiene

Números
3

Operadores
*

Luego se saca la x, ésta se debe manejar como un número pues cuando se evalúe así será, esta se mete en la pila de números

Números
x
3

Operadores
*

Ahora se debe sacar un + que se tendría que introducir en la pila de operadores, pero tiene prioridad cero que es menor que la prioridad de la multiplicación (que es 1), por lo que la multiplicación se debe hacer antes que la suma (la multiplicación tienen prioridad sobre la suma). Para este caso se sacan dos números de la pila y los "multiplicamos" obteniendo "3 x *" (recuerde que en realidad manejamos texto, cuando se dice que lo multiplicamos se quiere decir que se escribe la multiplicación en notación postfija). El resultado se introduce en la pila como un número (ahora se maneja como si toda esta expresión fuera un número porque es como si ya se hubiera evaluado) y la suma se mete en la pila de operadores, se obtienen las pilas

Números
3 x *

Operadores
+

Ahora se saca el número 4 y se mete en la pila de números

Números
4
3 x *

Operadores
+

Aquí ya se acaba la expresión por lo que se saca todo lo que queda en la pila. Para cada operador se deben tomar los valores necesarios de la pila de números; en este caso sólo hay una suma y dos números, al realizar la operación se sacan los dos números y se escribe en postfijo "3 x * 4 +" pues se colocan los dos números primero y luego la operación.

3.2. Una expresión con funciones

Se tomará como segundo ejemplo una expresión con más elementos, por ejemplo:

$$1+3*\tan(2*(1+x)-1)$$

En este caso se toma el 1, el + y el 3

Números
3
1

Operadores
+

Sigue la multiplicación, pero esta sólo saca la potencia y las de su mismo nivel por lo que no saca la suma, simplemente se agrega a los operadores

Números
3
1

Operadores
*
+

Sigue la función tangente que se maneja dentro de los operadores como un paréntesis que abre (en la pila le quitamos el paréntesis que en postfijo no se necesita)

Números
3
1

Operadores
tan
*
+

Ahora se deben incluir el 2 y el *. A la hora de incluir *, tangente no funciona como un operador sino como el inicio de otra expresión por lo que * no lo puede sacar; aquí no hay problema con la función prioridad porque a tan le asignaría un -1 que no lo saca nadie. También se agrega el paréntesis de apertura

Números
2
3
1

Operadores
(
*
tan
*
+

Ahora incluimos el 1, el + y la x

Números
x
1
2
3
1

Operadores
+
(
*
tan
*
+

Viene el cierre de paréntesis, este sacaría todos los elementos hasta que haya una apertura (un paréntesis o una función). En este caso solo debe sacar el + y se quita el paréntesis de la pila.

Números
1 x +
2
3
1

Operadores
*
tan
*
+

Sigue un - que tiene menor prioridad que el *, por lo que tenemos que sacar la multiplicación antes de meter el menos.

Números
2 1 x + *
3
1

Operadores
-
tan
*
+

Sigue un uno y luego un cierre de paréntesis que sacaría hasta tangente. En este caso, se debe sacar el menos que queda; la tangente, contrario al paréntesis de apertura, se debe agregar al final del texto para que se evalúe en la expresión.

Números
2 1 x + * 1 - tan
3
1

Operadores
*
+

Aquí se acaba la expresión por lo que se debe sacar todo lo que queda, obteniendo como resultado “1 3 2 1 x + * 1 - tan * +”

4. Programa que traduce de notación infija a postfija

Recuerde que para esta parte del programa se va a suponer que la expresión que digita el usuario no tiene errores, luego se darán consejos para detectarlos.

En los ejemplos de la sección anterior se observó que para realizar la traducción de la expresión se necesitan dos pilas de *Strings* (texto), una para los números y otra para los operadores. JAVA ya posee una clase que maneja pilas, esta clase se llama *Stack* que se encuentra dentro del paquete *java.util*, lo primero que tenemos que hacer es llamar a esta librería y hacer nuestra clase *Parseador*[4]; el inicio de nuestro programa se debe ver como:

```
import java.util.*;

public class Parseador{
...
}//fin de Parseador
```

Para crear una nueva pila se debe definir como un nuevo objeto:

```
Stack nuevaPila = new Stack();
```

La clase *Stack* se maneja con objetos (introduce objetos y saca objetos); para introducir un nuevo objeto dentro de la pila se utiliza la instrucción

```
nuevaPila.push(Objeto);
```

Para sacar un objeto de la pila (recuerde que una pila saca el último objeto que se introdujo) utilizamos

```
nuevaPila.pop();
```

Para "mirar" un objeto de la pila sin sacarlo se usa

```
nuevaPila.peek()
```

Además se puede preguntar si la pila está vacía con la instrucción

```
nuevaPila.empty()
```

que devuelve *True* si está vacía o *False* si no lo está.

Para empezar con la clase *Parseador*, definimos la variable global *ultimaParseada* como sigue:

```
private String ultimaParseada;
```

[4]Aquí se usa la palabra Parseador como un nombre simbólico para el programa, en realidad esta palabra no existe en español sino que se hace referencia a la palabra parser en inglés que tiene un significado similar a "traductor", por supuesto el usuario puede utilizar la palabra que guste.

Esta guarda un registro de la última expresión parseada (o traducida) en notación postfija, la expresión se guarda por si alguien quiere evaluar sin tener que dar la expresión en donde se evalúa.

El constructor de nuestra clase lo único que hace es poner *ultimaParseada* en 0.

```
public Parseador(){
   ultimaParseada="0";
}
```

La función parsear se define de tal forma que recibe un texto con la expresión en notación infija y devuelve otro texto con la expresión en notación postfija. La función lanza una excepción (*SintaxException*) si encuentra que la expresión está mal digitada.

```
public String parsear(String expresion) throws SintaxException{
   Stack PilaNumeros=new Stack(); //Pila de números
   Stack PilaOperadores= new Stack(); //Pila de operadores
   String fragmento;
   int pos=0, tamano=0;
   byte cont=1;
   final String funciones[]={"1 2 3 4 5 6 7 8 9 0 ( ) x e + - * / ^ %",
         "pi",
         "ln(",
         "log( abs( sen( sin( cos( tan( sec( csc( cot( sgn(",
         "rnd() asen( asin( acos( atan( asec( acsc( acot( senh( sinh( cosh( tanh(
              sech( csch( coth( sqrt(",
         "round( asenh( acosh( atanh( asech( acsch( acoth("};
   final private String parentesis="( ln log abs sen sin cos tan sec csc cot asen asin
              acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt round";
   final private String operadoresBinarios="+ - * / ^ %";

   //La expresión sin espacios ni mayúsculas.
   String expr=quitaEspacios(expresion.toLowerCase());
```

La función necesita dos pilas: *PilaNumeros* y *PilaOperadores*.

La variable *fragmento* se encargará de guardar el fragmento de texto (*String*) que se esté utilizando en el momento (ya sea un número, un operador, una función, etc.). La variable *pos* va a marcar la posición del caracter que se está procesando actualmente en el *String*.

La variable *funciones* contiene un arreglo de textos (*String*), en la primera posición tiene todas las expresiones de un caracter que se aceptarán, en la segunda posición están las expresiones de dos caracteres y así hasta llegar a la posición seis.

La variable *parentesis* contiene a todas las expresiones que funcionarán como paréntesis de apertura.

En *operadoresBinarios* se guardan todos los operadores que reciben dos parámetros.

Todas estas definiciones se hacen como texto (*String*) para después poder comparar si la expresión que el usuario digitó concuerda con alguno de ellos.

Se define también el *String* en donde se guarda la expresión sin espacios ni mayúsculas (*expr*); la función *toLowerCase()* ya está implementada en JAVA en la clase *String* mientras que *quitaEspacios* es una función que se define de la siguiente manera

```
private String quitaEspacios(String expresion){
   String unspacedString = ""; //Variable donde guarda la función

   //Le quita los espacios a la expresión que leyó
   for(int i = 0; i < expresion.length(); i++){
      if(expresion.charAt(i) != ' ')
         unspacedString += expresion.charAt(i);
   }//for

   return unspacedString;
}//quitaEspacios
```

Esta función va tomando cada uno de los caracteres de la expresión, si el caracter no es un espacio lo agrega al texto sin espacios *unspacedString*.

La excepción que lanza el programa también se define como una clase privada

```
private class SintaxException extends ArithmeticException{
   public SintaxException(){
      super("Error de sintaxis en el polinomio");
   }

   public SintaxException(String e){
      super(e);
   }
}
```

Volviendo a la función parsear, lo que seguiría es realizar un ciclo mientras no se acabe la expresión, es decir

```
   try{
      while(pos<expr.length()){
```

Lo que se haría dentro del *while* es ver si lo que sigue en la expresión es algo válido y tomar decisiones dependiendo si es un número, un operador, un paréntesis o una función.

El código:

```
   tamano=0;
   cont=1;
   while (tamano==0 && cont<=6){
      if(pos+cont<=expr.length() &&
           funciones[cont-1].indexOf(expr.substring(pos, pos+cont))!=-1){
         tamano=cont;
      }
      cont++;
   }
```

Hace que el contador vaya de 1 a 6 que es la máxima cantidad de caracteres que tiene una función y se inicializa *tamano* en cero. Luego se pregunta si la posición actual (*pos*) más el contador (*cont*) es menor

de la longitud del texto y si el fragmento de texto que sigue está en alguna de las funciones, si esto pasa el siguiente texto que se tiene que procesar es de tamaño *cont*.

Ahora se van tomando algunos casos con respecto al tamaño encontrado.

```
if (tamano==0){
   ultimaParseada="0";
   throw new SintaxException("Error en la expresión");
```

Si *tamano* continúa siendo cero quiere decir que el fragmento de texto que sigue no coincidió con ninguna función ni con algo válido por lo que se lanza una excepción y se pone la última expresión parseada en 0.

```
}else if(tamano==1){
```

Pero si el tamaño es uno tenemos varias opciones, la primera es que sea un número

```
if(isNum(expr.substring(pos,pos+tamano)){
   fragmento="";
do{
   fragmento=fragmento+expr.charAt(pos);
   pos++;
}while(pos<expr.length() && (isNum(expr.substring(pos,pos+tamano)) ||
      expr.charAt(pos) == '.' || expr.charAt(pos) == ','));
   try{
      Double.parseDouble(fragmento);
   }catch(NumberFormatException e){
      ultimaParseada="0";
      throw new SintaxException("Número mal digitado");
   }
   PilaNumeros.push(new String(fragmento));
   pos--;
```

En la primera línea, para preguntar si el caracter es un número se utiliza la función *isNum* definida por

```
private boolean isNum(char s) {
   if (s >= '0' && (s <= '9'))
      return true;
   else
      return false;
} //Fin de la funcion isNum
```

Que devuelve *True* si el caracter que recibe es un número (está entre 0 y 9) o *False* si no lo es.

Si el caracter actual es un número existe un problema, no se sabe cuántos dígitos tenga este número, por lo que se sacan todos los caracteres que sigan que sean números, puntos o comas.

En la variable *fragmento* se va a guardar el número, por lo que al inicio se debe vaciar (*fragmento=" "*). Luego se hace un ciclo mientras no se acabe la expresión y el caracter que sigue sea un número, un punto o una coma: *while(pos<expr.length() && (isNum(expr.substring(pos,pos+tamano)) || expr.charAt(pos) == '.' || expr.charAt(pos) == ','))*{ .

Luego, la variable *fragmento* se trata de convertir a *double* y se mete en la pila de números; si no la puede convertir el programa lanza una excepción en el bloque *try-catch*.

De esta manera se maneja un número cualquiera. Ahora la segunda posibilidad con tamaño uno es que el caracter sea x o e. Estos dos casos se manejan como un número que se mete en la pila.

```
   else if (expr.charAt(pos)=='x' || expr.charAt(pos)=='e'){
      PilaNumeros.push(expr.substring(pos,pos+tamano));
```

Si el caracter que sigue es uno de los operadores +, *, / y % (el menos '-' se maneja igual, pero se recomienda ponerlo en un caso aparte para manejar posteriormente los menos unarios) se deben sacar todos los operadores con prioridad mayor o igual a ellos y meter el operador en la pila

```
   }else if (expr.charAt(pos)=='+' || expr.charAt(pos)=='*' ||
         expr.charAt(pos)=='/' || expr.charAt(pos)=='%'){
      sacaOperadores(PilaNumeros, PilaOperadores, expr.substring(pos,pos+tamano));
```

En este caso se declara una función (ya que se utilizará mucho al manejar detalles posteriores) que realice el trabajo de sacar operadores de la pila, esta función se define de la siguiente manera

```
private void sacaOperadores(Stack PilaNumeros, Stack PilaOperadores, String operador){
   final String parentesis="( ln log abs sen sin cos tan sec csc cot sgn asen asin
         acos atan asec acsc acot senh sinh cosh tanh sech csch coth sqrt round asenh
         asinh acosh atanh asech acsch acoth";
   while(!PilaOperadores.empty() &&
         parentesis.indexOf((String)PilaOperadores.peek( ))==-1 &&
         ((String)PilaOperadores.peek()).length()==1 &&
         prioridad(((String)PilaOperadores.peek()).charAt(0))>=
         prioridad(operador.charAt(0))){
      sacaOperador(PilaNumeros, PilaOperadores);
   }
   PilaOperadores.push(operador);
}
```

Aquí se vuelve a declarar la variable *parentesis* para el *while*. En este *while* se indica que se deben sacar operadores mientras haya algo en la pila, lo que siga en la pila no sea un paréntesis, sea de un solo caracter y cuya prioridad sea mayor o igual a la prioridad del operador que se está procesando en ese momento; luego se guarda el operador en la pila correspondiente.

Observe que el *while* llama a *sacaOperador* que es una función definida posteriormente cuyo código es

```
private void sacaOperador(Stack Numeros, Stack operadores) throws EmptyStackException{
   String operador, a, b;
   final String operadoresBinarios="+ - * / ^ %";
   try{
      operador=(String)operadores.pop();

      if(operadoresBinarios.indexOf(operador)!=-1){
         b=(String)Numeros.pop();
```

```
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+b+" "+operador));
      }else{
         a=(String)Numeros.pop();
         Numeros.push(new String(a+" "+operador));
      }
   }catch(EmptyStackException e){
      throw e;
   }
}//sacaOperador
```

Esta función se encarga de sacar dos números de la pila correspondiente, esto si es un operador binario o un número si es unario; luego introduce el nuevo número en notación postfija en la pila. Esta función recibe las dos pilas y no devuelve nada, si se le acaban los elementos a alguna de las pilas se lanza una excepción. Esta es la última función externa que se va a necesitar para traducir la expresión.

Con la potencia (^) es más sencillo ya que este operador no saca a nadie, el código quedaría de la siguiente manera.

```
   }else if (expr.charAt(pos)=='^'){
      PilaOperadores.push(new String("^"));
```

En este caso, simplemente se mete el operador a su pila correspondiente.

Si el caracter fuera un paréntesis de apertura simplemente se mete en la pila de operadores.

```
   }else if (expr.charAt(pos)=='('){
      PilaOperadores.push(new String("("));
```

Y si el caracter es un paréntesis de cierre se deben sacar operadores hasta encontrar una apertura de paréntesis en la pila.

```
   }else if (expr.charAt(pos)==')'){
      while(!PilaOperadores.empty() &&
            parentesis.indexOf(((String) PilaOperadores.peek()))==-1){
         sacaOperador(PilaNumeros, PilaOperadores);
      }
      if(!((String)PilaOperadores.peek()).equals("(")){
         PilaNumeros.push(new String(((String)PilaNumeros.pop()) + " " +
               PilaOperadores.pop()));
      }else{
         PilaOperadores.pop();
      }
   }
```

Si el paréntesis de apertura no era el caracter "(" (es decir, era una función) entonces la concatena al final del texto en la notación postfija, si era un paréntesis simplemente lo desecha (recuerde que en notación postfija no hay paréntesis).

Aquí se terminan de procesar todos los elementos posibles de un solo caracter, ahora se pasará al caso de dos o más caracteres

```
   }else if(tamano>=2){
      fragmento=expr.substring(pos,pos+tamano);
      if(fragmento.equals("pi")){
         PilaNumeros.push(fragmento);
      }else if(fragmento.equals("rnd()")){
         PilaNumeros.push("rnd");
      }else{
         PilaOperadores.push(fragmento.substring(0,fragmento.length()-1));
      }
   }
   pos+=tamano;
}//Fin del while
```

En este caso se toma la expresión en *fragmento*, si *fragmento* es igual a "*pi*", lo metemos en la pila de números; lo mismo hacemos si es *rnd()*.

Cualquier otra posibilidad de tamaño mayor que dos es que sea una función por lo que se mete en la pila de operadores quitándole el paréntesis. Al final se aumenta la posición de acuerdo al tamaño del texto procesado para pasar al siguiente caracter en la expresión.

Ya cuando se acabó la expresión se debe procesar todo lo que quedó en las pilas en el proceso final, esto se hace con un while que saque todos los operadores que quedaron

```
      //Procesa al final
      while(!PilaOperadores.empty()){
         sacaOperador(PilaNumeros, PilaOperadores);
      }

   }catch(EmptyStackException e){
      ultimaParseada="0";
      throw new SintaxException("Expresión mal digitada");
   }

   ultimaParseada=((String)PilaNumeros.pop());

   if(!PilaNumeros.empty()){
      ultimaParseada="0";
      throw new SintaxException("Error en la expresión");
   }

   return ultimaParseada;
}//Parsear
```

Si hubo algún error con las pilas en el proceso se lanza una excepción y se pone a *ultimaParseada* en cero. Al final se devuelve *ultimaParseada*.

Con esto ya acabamos el programa que convierte una expresión matemática del lenguaje natural a la notación postfija.

5. Evaluación de expresiones postfijas

Ahora que se tiene el traductor de expresiones en notación postfija, lo que hace falta es el segundo módulo, es decir, el que evalúa una expresión en notación postfija en un número dado.

Al igual que para el módulo anterior, se va a iniciar mostrando varios ejemplos que realizan esta evaluación para observar cuál es el procedimiento que se debe seguir.

5.1. Evaluación en notación postfija: primer ejemplo

En este caso vamos a iniciar evaluando en la expresión "x 2 ^ sen 4 x * +" cuando $x = 3$. Recuerde que para evaluar en postfijo se leía la expresión de izquierda a derecha y se seguían tres reglas a saber:

1. Si lo que sigue en la expresión es un número, se agrega a la pila de números.
2. Si sigue una operación que ocupa dos números (como la suma y la resta) se sacan los dos últimos números de la pila, se realiza la operación y se introduce en la pila el resultado.
3. Si es una función que ocupa un solo número (como seno o coseno) entonces se saca un número de la pila, se evalúa y se guarda el resultado.

Por lo tanto en este caso se inicia tomando la x y se introduce un 3 en la pila (es el valor que se evalúa en vez de la x), luego se toma el 2 y también se introduce en la pila obteniendo

2
3

Luego sigue un ^ por lo que se toman los dos valores y se obtiene $3^2 = 9$, nos queda una pila de un elemento

9

Ahora sigue sen, esta función se realiza sobre un solo número (el único que hay en la pila), se obtiene $\text{sen}(9) = 0{,}412118485241757$, este es el nuevo valor en la pila

0.412118485241757

Sigue agregar a la pila un 4 y un 3 (en vez de la x)

3
4
0.412118485241757

Sigue una multiplicación, por lo que tomamos el 4 y el 3 de la pila y le metemos $4 \cdot 3 = 12$ a la pila, nos queda

12
0.412118485241757

Por último, queda una suma, es decir $0{,}412118485241757 + 12 = 12{,}412118485241757$ que es el resultado final.

5.2. Evaluación en notación postfija: segundo ejemplo

Un segundo ejemplo antes de programar el módulo será evaluar la expresión

$$\operatorname{asen}\left(\tan\left(\ln\left(x+\pi\right)\right)\right)+x^{x^2+3\cdot x+5}$$

que en notación postfija se escribe "x pi + ln tan asen x x 2 ^ 3 x * + 5 + ^ +" cuando $x=-1$.
Se toma la x, es decir, se introduce el -1 en la pila, luego pi

3.14159265358979
-1

Ahora sigue la suma de estos términos

2.14159265358979

Ahora se le saca logaritmo natural a este valor

0.761549782880893

Sigue el cálculo de la tangente

0.953405606022648

Ahora se calcula el arcoseno

1.26433002209559

Sigue introducir dos veces -1 (en vez de x) en la pila y un dos

2
-1
-1
1.26433002209559

Ahora sigue una exponente, $(-1)^2$=1

1
-1
1.26433002209559

Se agrega un 3 y un -1

-1
3
1
-1
1.26433002209559

Se multiplica $3 \cdot -1 = -3$

-3
1
-1
1.26433002209559

Se suma $1 + -3 = -2$

-2
-1
1.26433002209559

Se agrega un 5

5
-2
-1
1.26433002209559

Se suma, $-2 + 5 = 3$

3
-1
1.26433002209559

Sigue un exponente $(-1)^3 = -1$

-1
1.26433002209559

Por último, se suman $1{,}26433002209559 + -1 = 0{,}26433002209559$, que es el resultado final.

6. Función que evalúa expresiones

Para hacer este módulo se inicia declarando la función para evaluar, la cual recibe la expresión en notación postfija y el número que se va a evaluar (que es un *double*); la función devuelve el resultado de la evaluación (que también es un *double*). En este caso la función se llamará *f* para simular que se evalúa en una función *f(x)*. Esta función lanzará una excepción (*ArithmeticException*) si encuentra un error en la expresión.

```
public double f(String expresionParseada, double x) throws ArithmeticException{
   Stack pilaEvaluar = new Stack(); //Pila de double's para evaluar
   double a, b;
   StringTokenizer tokens=new StringTokenizer(expresionParseada);
   String tokenActual;
   ...
```

En esta función se declaran algunas variables. En un principio se declara la pila que guardará los números (*pilaEvaluar*), luego se declaran dos números a y b en donde se guardarán los elementos que se sacan de la pila para ser operados.

El *StringTokenizer* es otra clase que se define en la librería *java.util*, ésta toma un texto y lo separa en unidades lexicográficas (llamadas lexemas o "tokens" en inglés), cada palabra separada por un espacio es un lexema; por ejemplo, el texto "2 3 + sen" tiene cuatro lexemas: "2", "3", "+" y "sen". El *tokenActual* guarda la unidad lexicográfica que se procesa en un momento dado.

Para saber si hay más lexemas en una expresión se utiliza *tokens.hasMoreTokens()* que devuelve *True* o *False* y para sacar el siguiente lexema se usa *tokens.nextToken()*. Así, lo que falta en nuestra función es ir pasando por los lexemas e ir haciendo cálculos o guardando números según corresponda, el código sería el siguiente

```
try{
   while(tokens.hasMoreTokens()){
      tokenActual=tokens.nextToken();
      ...
   }//while
}catch(EmptyStackException e){
   throw new ArithmeticException("Expresión mal parseada");
}catch(NumberFormatException e){
   throw new ArithmeticException("Expresión mal digitada");
}catch(ArithmeticException e){
   throw new ArithmeticException("Valor no real en la expresión");
}

a=((Double)pilaEvaluar.pop()).doubleValue();

if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");

return a;

}//funcion f
```

Los bloques *try-catch-throw* son los que lanzan las excepciones si en algún momento se acabaron los elementos en la pila (*EmptyStackException*), si hubo algún número que no pudo entender (*NumberFormatException*) o si hubo algún cálculo que no corresponde a un número real (*ArithmeticException*).

El proceso de analizar lexemas se repetirá mientras hallan más lexemas *while(tokens.hasMoreTokens()){* y se saca el siguiente lexema con la instrucción *tokenActual=tokens.nextToken();*.

Al final del proceso, sacamos el último valor de la pila con la instrucción *a=((Double)pilaEvaluar.pop()).doubleValue();* que lo que hace es sacar el elemento con *pilaEvaluar.pop()*, luego lo convierte a un objeto

Double (recuerde que la pila trabaja con objetos) y, por último, calcula su valor double primitivo (con *doubleValue()*).

Si este no era el último valor en la pila, quiere decir que hubo un error al evaluar (faltaron operadores) y se lanza una excepción con el bloque

```
if(!pilaEvaluar.empty())
   throw new ArithmeticException("Expresión mal digitada");
```

Si no hubo ningún error entonces devuelve el valor encontrado.

Ahora veamos el bloque de código que hace falta dentro del *while*.

Si el lexema actual es alguno de los números "e", "pi" o "x", simplemente lo metemos en la pila, tomemos por ejemplo el número "e":

```
   if(tokenActual.equals("e")){
      pilaEvaluar.push(new Double(Math.E));
```

Al meter el número *Math.E* en la pila (este número es un *double*) se debe introducir como un objeto *Double*, por eso se utiliza el código *new Double*.

Cuando se tiene que meter "*x*" lo que se hace es introducir en la pila el valor que recibe la función. En todos los siguientes casos de este *if* utilizamos *elseif*; en este caso se utilizó *if* por ser el primero.

Por otro lado, si el lexema siguiente es un operador que recibe dos números, entonces primero se saca los dos números de la pila, y se guarda el resultado de la operación en la pila, tomando como ejemplo la suma.

```
   }else if(tokenActual.equals("+")){
      b=((Double)pilaEvaluar.pop()).doubleValue();
      a=((Double)pilaEvaluar.pop()).doubleValue();
      pilaEvaluar.push(new Double(a+b));
```

De igual manera se hace para funciones con un solo número, como logaritmo.

```
   }else if(tokenActual.equals("ln")){
      a=((Double)pilaEvaluar.pop()).doubleValue();
      pilaEvaluar.push(new Double(Math.log(a)));
```

Por último, si no fue nada de lo anterior, la única posibilidad que queda es que sea un número, este caso se toma el lexema y se trata de convertir en un *double* con *Double.parseDouble(tokenActual)*, si no puede automáticamente se lanza la excepción; el número se convierte en un nuevo objeto *Double* y se mete en la pila. El código quedaría

```
   }else{
      pilaEvaluar.push(new Double(Double.parseDouble( tokenActual)));
   }
```

Agregando los casos correspondientes para cada una de las funciones que se admiten, ya se tiene un evaluador de expresiones en notación postfija funcional.

En el programa de ejemplo que se muestra al final del artículo, se admite lo siguiente:

- Números "especiales": e, pi, x.
- Operadores: +, -, *, /, %, ^.
- Funciones: ln, log, abs, sen, sin, cos, tan, sec, csc, cot, sgn, asen, asin, acos, atan, asec, acsc, acot, senh, sinh, cosh, tanh, sech, sech, coth, sqrt, asenh, asinh, acosh, atanh, asech, acsch, acoth, round.
- Números reales, por ejemplo 2,15, -20,5, etc.

El evaluador se puede modificar para poder admitir la variable y o la z por si queremos evaluar funciones de varias variables, además se pueden agregar otras funciones con modificaciones mínimas (otra variable se manejaría igual que la x, para otra función habría que manejarla como las demás funciones con su cálculo correspondiente).

Este evaluador también lo sobrecargamos para que pueda admitir la forma *f(x)* que evaluaría la función en la última expresión parseada (recuerde que la última expresión parseada se guardaba en el módulo anterior en una variable global llamada *ultimaParseada*).

```
public double f(double x) throws ArithmeticException{
   try{
      return f(ultimaParseada,x);
   }catch(ArithmeticException e){
      throw e;
   }
}//Fin de la funcion f
```

Esta función se puede probar introduciendo cualquier expresión en notación postfija y un número para evaluar.

Uniendo estos dos programas, se tiene una poderosa herramienta para evaluar expresiones y funciones matemáticas.

Para poder usar esta clase dentro de un programa se debe crear un objeto (*Obj*) de tipo *Parseador*.

```
Parseador miparser = new Parseador();
```

Para parsear una expresión *expr* se escribe *miparser.parsear(expr)*, la función devuelve un *String* con *expr* en notación postfija, además el programa también guarda de manera automática la última expresión parseada. Para evaluar el número x en la expresión se utilizar *miparser.f(x)* para evaluar en la última expresión o se puede pasar una expresión en notación postfija escribiendo *miparser.f(exprEnPostfija, x)*.

Así, por ejemplo, el siguiente es el código de un applet[5] en donde se utiliza la clase *Parseador*

```
import java.applet.*;
import java.awt.*;

public class PruebaParseador extends java.applet.Applet {
  //Constructor del parseador
  Parseador miparser=new Parseador();
  //Expresión a parsear
```

[5] Una aplicación hecha para poner en una página de Internet

```
String expresion=new String();
//Valor en el que se va a evaluar
double valor=0;
//Textfield donde se digita la expresión a parsear
TextField inputexpresion = new TextField("x + 5");
//Textfield donde se digita el valor a evaluar en la expresión
TextField inputvalor = new TextField("0",5);
//Botón para evaluar
Button boton= new Button("Evaluar la expresión");
//Resultado de parsear la expresión
TextField outputparseo = new TextField("          ");
//Resultado de la evaluación en la expresión
TextField outputevaluar = new TextField("         ");
//Label donde se dan los errores
Label info = new Label("Información en extremo importante           ", Label.CENTER);

public void init(){ //Todo se pone en el applet
  add(inputexpresion);
  add(inputvalor);
  add(boton);
  add(outputparseo);
  add(outputevaluar);
  add(info);
}//init

public boolean action(Event evt, Object arg){
  if (evt.target instanceof Button){ //Si se apretó el botón
    try{
      info.setText(""); //Se pone el Label de los errores vacío
      expresion=inputexpresion.getText(); //Se lee la expresión
      //Se lee el valor a evaluar
      valor=Double.valueOf(inputvalor.getText()).doubleValue();
      //Se parsea la expresión
      outputparseo.setText(miparser.parsear(expresion));
      //Se evalúa el valor y se redondea
      outputevaluar.setText(""+redondeo(miparser.f(valor),5));
    }catch(Exception e){ //Si hubo error lo pone en el Label correspondiente
      info.setText(e.toString());
    }
  }//if del botón
  return true;
}//action

/*
*Se redondea un número con los decimales dados
*/
private double redondeo(double numero, int decimales){
  return ((double)Math.round(numero*Math.pow(10,decimales)))/Math.pow(10,decimales);
}
```

```
}//PolCero
```

El applet se puede ver bajar y ver funcionando en la sección correspondiente a los programas, si se copia el texto se debe pegar en un archivo que se llame PruebaParseador.java y se debe tener una página HTML de prueba para verlo, la más sencilla es una página que tenga por código:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<CENTER>
<APPLET
  code = "PruebaParseador.class"
  width = "500"
  height = "300"
  >
</APPLET>
</CENTER>
</BODY>
</HTML>
```

Este applet tiene dos cajas de texto (*TextField*), una es donde se digita la expresión a evaluar (*inputexpresion*) y la otra es donde se digita el valor a evaluar(*inputvalor*), por defecto estas cajas iniciar con ´´x+5" y "0" respectivamente. El applet tiene un botón (*boton*), al ser presionado, se lee lo que el usuario escribió en las cajas de texto y se escribe el resultado de parsear la expresión con el texto *outputparseo.setText(miparser.parsear(expresion));*, observe que el código que hace la traducción es *miparser.parsear(expresion)* y el resultado se escribe en la etiqueta *outputparseo*; algo similar se hace con el resultado de la evaluación, sólo que antes se redondea el resultado con el código *redondeo(miparser.f(valor),5)*

7. Comentarios finales

Para finalizar, se comentará un poco las ideas para verificar si una expresión que digitó el usuario está bien escrita, lo cual no verificamos en el artículo pero sí está en el programa final. Para esto, lo que se hizo fue declarar una variable global *anterior* que se encarga de guardar un número dependiendo si lo último que tradujo en una expresión fue un número, un operador, etc. de la siguiente manera:

Valor	Última subexpresión traducida	Significa
0	nada	nada
1	Número, pi, e, x	Números
2	+, -, *, /, ^, %	Operadores
3	(, sen(, cos(, etc.	Paréntesis de apertura
4	)	Paréntesis de cierre

De esta forma se deben cumplir las siguientes reglas:

- Si no se había traducido nada (*anterior=0*) entonces se puede admitir cualquier cosa menos (+ * / ^ %).

- Si lo anterior fue un número puede seguir cualquier cosa.
- Si lo anterior fue un operador puede seguir cualquier cosa menos otro operador (con excepción de -).
- Si lo anterior fue un paréntesis de apertura puede seguir cualquier cosa menos (+ * / ^ %)
- Si lo anterior fue un cierre de paréntesis debe seguir un operador, un número (en cuyo caso hay una multiplicación oculta), un paréntesis de apertura (también hay una multiplicación oculta) u otro paréntesis de cierre, estas multiplicaciones ocultas deben ser agregadas antes de poner el número o el paréntesis.
- Para un menos unario se debe agregar un -1 en la pila de números y un "por" en la de operadores (sacando los correspondientes operadores con mayor prioridad); el menos unario se da si no había nada anterior al menos o si era otro operador o un paréntesis de apertura.

El código completo con su correspondiente archivo, el archivo de prueba y la página HTML se encuentran al final de este artículo junto con el código similar en un módulo de Visual Basic que puede ser usado en cualquier programa de este lenguaje.

Otro programa recomendado que es similar al que se desarrolló en el artículo y que puede ser estudiado, bajado con su código completo y utilizado en otros programas libremente es: JEP[5]

8. Programas

AQUI VAN LOS PROGRAMAS PARA QUE SEAN BAJADOS Y EL APPLET

JAVA.ZIP y VISUAL BASIC.ZIP

Referencias

[1] Aho, Alfred; Hopcroft, John; Ullman, Jeffrey. Data structures and algorithms. Massachusetts : Addison-Wesley, 1983.

[2] Deitel H. y Deitel P. (1998) *Cómo programar en Java*. [Traducción del libro *Java how to program*] Primera edición. Prentice-Hall, México.

[3] Murillo, M.; Soto, A. y Alfredo, J. (2000) *Matemática básica con aplicaciones*. EUNED. San José, C.R.

[4] Sitio Web de JAVA: www.java.sun.com

[5] Sitio Web de JEP - Java Expresion Parser: http://www.singularsys.com/jep