Una clase Parser en Java para evaluar expresiones algebraicas

Recepción: Mayo de 2006 / Aceptación: Junio de 2006

(1) Edgar Ruiz Lizama
(2) Eduardo Raffo Lecca

RESUMEN

El artículo presenta una clase parser en Java para evaluar expresiones algebraicas empleando algoritmos fundamentales para la construcción de compiladores, pasando por la conversión de expresiones de infija a postfija, la evaluación de expresiones en notación postfija, el algoritmo de evaluación por precedencia de operadores, el algoritmo parsing de precedencia y el algoritmo de construcción de funciones de precedencia. El objetivo del artículo es escribir un analizador léxico en un lenguaje convencional de programación de sistemas, utilizando las posibilidades de entrada y salida del lenguaje Java para leer las expresiones a evaluar desde la entrada; procesarlas y enviar los resultados a la salida.

Palabras Clave: Parser, analizador léxico, evaluación de expresiones postfijas, algoritmo de precedencia de operadores.

A PARSER CLASS IN JAVA TO EVALUATE ALGEBRAIC EXPRESSIONS
ABSTRACT

The article presents a Parser class in Java to evaluate algebraic expressions using fundamental algorithms for the construction of compilers, passing from the conversion of expressions from infix to postfix, the evaluation of postfix expressions, the evaluation for precedence of operators, parser algorithms of precedence and construction algorithm of precedence functions. The goal of the article is to write a lexical analyser in a conventional language of systems programming, using the possibilities of input and output of Java language in order to read expressions to evaluate from the input, to process them and to send the results to the output.

Key words: Parser, lexical analyser, postfix expressions evaluation, operators algorithm precedence.

INTRODUCCIÓN

Cuando se escriben programas en un lenguaje convencional de programación, que tienen que ver con la evaluación de expresiones, se tiene el problema que la expresión a evaluar desde la entrada no siempre es la misma o del mismo tipo. Por ello el programador debe escribir el código necesario para que la entrada de sus programas sea lo más general posible, permitiendo un comportamiento adecuado frente a expresiones y/o funciones en la entrada.

LA CLASE PARSER

Para evaluar expresiones, se hace uso de algunas técnicas utilizadas en el diseño de compiladores. Las expresiones según su notación pueden ser:

Notación infija.- operando-operador-operando. Ejemplo: A + B
Notación prefija.- operador-operando-operando. Ejemplo: + A B
Notación postfija.- operando-operando-operador. Ejemplo: A B +

Al hacer uso de las expresiones infijas, la expresión 1 + 5; consiste de un operador binario junto con sus operandos (argumentos). Una expresión más elaborada es: 1 + 5 * 2; la cual matemáticamente es evaluada a 11, debido a que, el operador * (de multiplicación) tiene mayor precedencia que el de suma. Si no se tiene en cuenta la precedencia, la respuesta será 12. Esto nos lleva a concluir que una simple evaluación de izquierda a derecha no basta.

En expresiones donde están presentes los operadores de resta y potencia ¿Cuál de ellos debe evaluarse primero? Para responder a la pregunta considere las expresiones

(1) 6 – 3 – 1,
(2) 2 ^ 3 ^ 2

Para la expresión (1) el resultado es 2; pues las restas se evalúan de izquierda a derecha. En el caso (2), las potencias se evalúan de derecha a izquierda; y la expresión reflejará 2^{3^2} en lugar de $(2^3)^2$.

Las expresiones en postfija, proporcionan un mecanismo directo de evaluación; existiendo algoritmos de pila para la resolución.

(1) Ingeniero Industrial. Profesor del Departamento de Ingeniería de Sistemas e Informática, UNMSM. E-mail: eruizl@unmsm.edu.pe
(2) Ingeniero Industrial. Profesor del Departamento de Ingeniería de Sistemas e Informática, UNMSM. E-mail: eraffol@unmsm.edu.pe

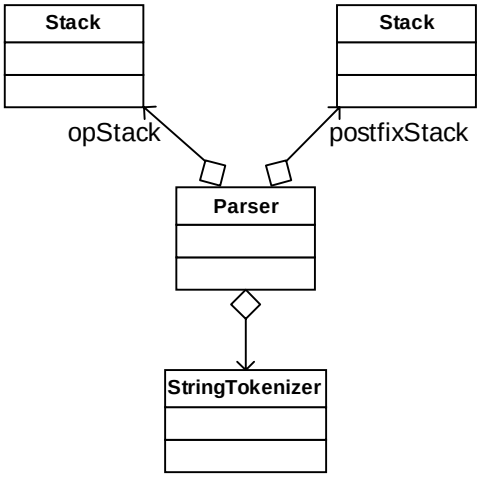

Figura 1. Diagrama UML de clases para la clase Parser

```
public class Parser
{
   private static final int EOL    = 0;
   private static final int VALUE  = 1;
   private static final int OPAREN = 2;
   private static final int CPAREN = 3;
   private static final int EXP    = 4;
   private static final int MULT   = 5;
   private static final int DIV    = 6;
   private static final int PLUS   = 7;
   private static final int MINUS  = 8;
   private static final int FUNCT  = 9;
   private static String[] function =
   {
          "sqrt","sin",
          "cos","tan",
          "asin","acos",
          "atan","log",
         "floor","eXp"
   };
   private String string;
    private Stack opStack;        // Operator stack for
conversion
    private Stack postfixStack;  // Stack for postfix
machine
     private StringTokenizer str; // StringTokenizer
stream
   // . . . continua codigo de los metodos de la clase .
. .
   public double getValue(double x)
   {
           return getValue(x,0,0);
   }
```

```
public double getValue(double x,double y)
{
        return getValue(x,y,0);
}
public double getValue(double x,double y,double z)
{
   // for each call
   opStack = new Stack( );
   postfixStack = new Stack( );
   str = new StringTokenizer(string,"+*-/^()xyz ",true);
   opStack.push( new Integer( EOL ) );

   EvalTokenizer tok = new EvalTokenizer(str,x,y,z);
    Token lastToken;
    do
    {
       lastToken = tok.getToken( );
       processToken( lastToken );
    } while( lastToken.getType( ) != EOL );

   if( postfixStack.isEmpty( ) )
    {
       System.err.println( "Missing operand!" );
       return 0;
    }
    double theResult = postFixTopAndPop( );
   if( !postfixStack.isEmpty( ) )
      System.err.println( "Warning: missing operators!"
);
    return theResult;
  }
}
// . . . continua codigo de los metodos de la clase . . .
```

Figura 2. Declaración de la clase Parser

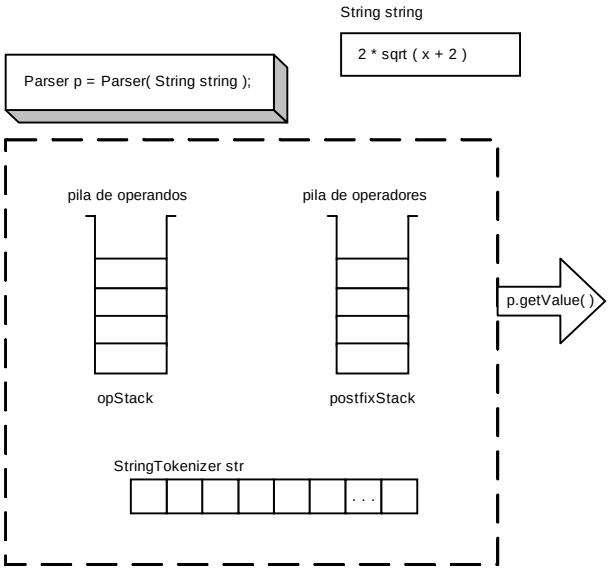

Figura 3. Objeto a la clase Parser

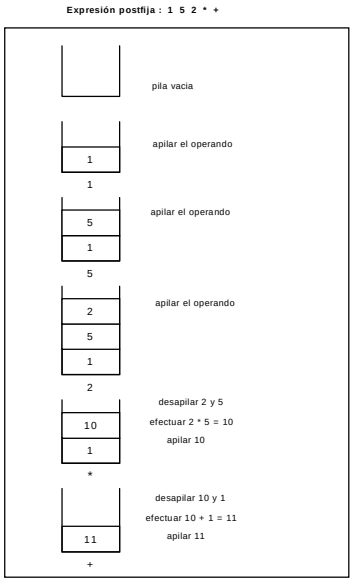

Figura 4. Máquina postfija para evaluar 1 5 2 * +

En la figura 1, se presenta el diagrama UML para la clase Parser.

La clase Parser, escrita en Java; evalúa expresiones algebraicas infijas que contengan operadores de adición, resta, multiplicación, división y de exponenciación, adicionalmente acepta paréntesis y llamada a funciones. Se utiliza un algoritmo de evaluación de expresiones con precedencia entre operadores. En la figura 2; se presenta una porción de la clase Parser.

En la figura 3, se muestra el objeto p, que es una instancia a la clase Parser.

MÁQUINAS POSTFIJAS

Una expresión postfija esta formada por una serie de operandos y operadores. Se evalúa usando una máquina postfija, en la forma siguiente: cuando se encuentra un operando, se apila en la pila; cuando se encuentra un operador, el número de operandos (según el operador) son sacados de la pila; se realiza la operación, y el resultado se apila de nuevo en la pila. Cuando la expresión postfija completa ha sido procesada, el resultado deberá de ser el único valor en la pila.

Considere la expresión 1 + 5 * 2; que en notación postfija es,

1 5 2 * +

La evaluación es como sigue: el 1, el 5 y el 2 son apilados en ese orden, en la pila de operandos. Al leerse el operador de multiplicación; el 2 y el 5 son desapilados, efectuándose el producto de 2*5, siendo el resultado 10; ahora, 10 es metido a la pila. Continuando con el algoritmo; se lee en la entrada el operador de suma, por lo que 10 y 1 son sacados de la pila, procediéndose a efectuar la suma entre estos; siendo el resultado 11; el cual es apilado nuevamente en la pila. En consecuencia el resultado de la evaluación es 11. La figura 4, ilustra este hecho.

En la figura 5, se presenta el código para el algoritmo de evaluación en postfija.

Conversión de notación infija a postfija

Cuando se encuentra un operando, podemos pasarlo inmediatamente a la salida. Cuando se encuentra un operador, no es posible pasarlo a la salida; pues se debe esperar a encontrar su segundo operando, por lo que se hace necesario una estructura de datos temporal. Una pila es la estructura de datos adecuada para almacenar operadores pendientes.

La expresión infija 2 ^ 4 – 1, tiene su expresión en postfija: 2 4 ^ 1 -. De acuerdo al procesamiento de un operador de entrada; se debe sacar de la pila aquellos operadores que deben ser procesados atendiendo a las reglas de precedencia y asociatividad. Este, es el principio básico en el algoritmo sintáctico de expresiones por precedencia de operadores. Al respecto ver la figura 6.

```
 /**
* Process an operator by taking two items off the
postfix stack, applying the operator, and pushing the
result. Print error if missing closing parenthesis or
division by 0.
    */
   private void binaryOp( int topOp )
   {
     if( topOp == OPAREN )
      {
        System.err.println( "Unbalanced parentheses"
);
         opStack.pop( );
         return;
      }
     if(topOp >= FUNCT )
      {
         double d=getTop();
                              postfixStack.push(new
Double(functionEval(topOp, d)));
         opStack.pop( );
         return;
      }
      double rhs = getTop( );
      double lhs = getTop( );
     if( topOp == EXP )
         postfixStack.push( new Double(pow(lhs,rhs)
) );
     else if( topOp == PLUS )
        postfixStack.push( new Double(lhs + rhs) );
     else if( topOp == MINUS )
        postfixStack.push( new Double(lhs - rhs) );
     else if( topOp == MULT )
        postfixStack.push( new Double(lhs * rhs) );
     else if( topOp == DIV )
        if( rhs != 0 )
          postfixStack.push( new Double(lhs / rhs) );
       else
       {
         System.err.println( "Division by zero" );
         postfixStack.push( new Double( lhs ) );
       }
    opStack.pop();
}
private double functionEval(int topOp, double d)
{
        double y = 0;
        switch (topOp) {
                case 9:
                        y = Math.sqrt(d); break;
                case 10:
                        y = Math.sin(d); break;
                case 11:
                        y = Math.cos(d); break;
                case 12:
                        y = Math.tan(d); break;
                case 13:
                        y = Math.asin(d); break;
                case 14:
                        y = Math.acos(d); break;
                case 15:
                        y = Math.atan(d); break;
                case 16:
                        y = Math.log(d); break;
                case 17:
                        y = Math.floor(d); break;
                case 18:
                        y = Math.exp(d);
        }
        return y;
}
```

Figura 5. Código para el algoritmo de evaluación en postfija

Análisis sintáctico por precedencia de operadores [AHO]

Sea la siguiente gramática para expresiones:

E → E A E
| (E)
| - E
| F(E)
| id
A → + | - | * | / |^

donde, F es el operador de llamada a función.

Un analizador sintáctico para gramáticas, tiene la propiedad que ningún lado derecho de la producción es ϵ, ni tiene dos no terminales adyacentes. A esto se denomina gramática de operadores.

Se observa que la producción E → E A E tiene dos no terminales consecutivos.La conversión de la gramática anterior a la gramática de operadores es la siguiente:

E → E + E
E - E
E * E
E / E
E ^ E
(E)
- E
F (E)
id

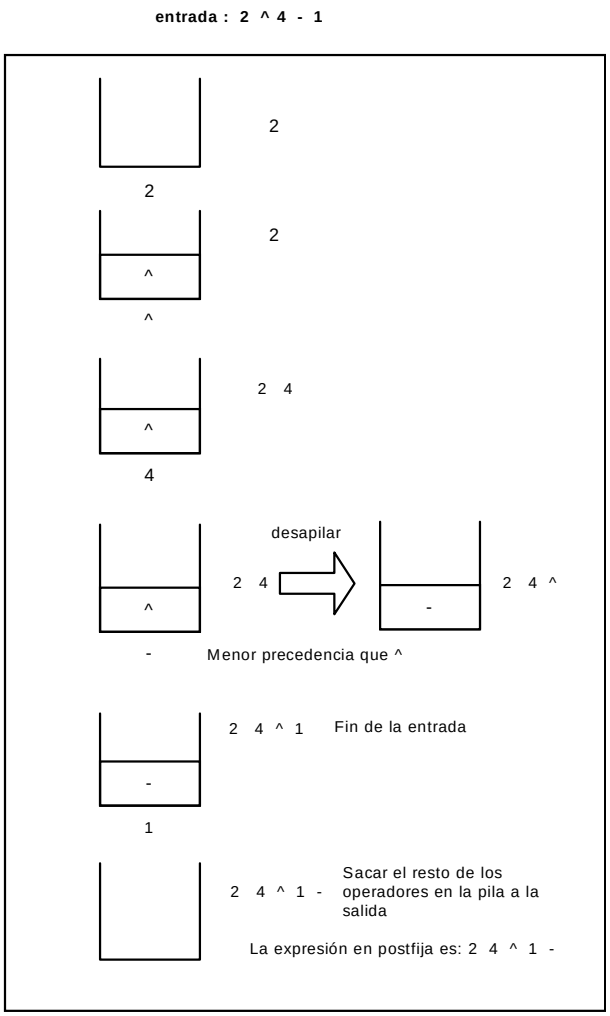

Figura 6. Conversión de infija a postfija

```
/**
* Construct an evaluator object.
* @param s the string containing the expression.    */
public Parser( String s )
{
   opStack = new Stack( );
   postfixStack = new Stack( );
    string = unary2Binary(s);
   str = new StringTokenizer(string,"+*-/^()xyz ",true);

   opStack.push( new Integer( EOL ) );
}
```

Figura 7. Constructor para la clase Parser

```
Algoritmo análisis sintáctico
    Apuntar al primer símbolo de s
    while true
        if $ se encuentra en la cima de la pila y la entra
          da apunta a fin de cadena then
          return
        else
            a = símbolo terminal en la cima de la pila
            b = símbolo que apunta a la entrada
        if a <. b or a   b then
            apilar b en la pila
            avanzar al siguiente símbolo en la entrada
        else
            if a .> b
            repeat
                  extraer el elemento de la cima de la pila
            until el terminal de la cima de la pila <.
            else
                  error ( )
  End Algoritmo análisis sintáctico
```

Figura 8. Algoritmo del parser por precedencia de operadores

El algoritmo de análisis sintáctico por precedencia de operadores, tiene como entrada una cadena de caracteres *s* y una tabla de relaciones de precedencia, y como salida, una estructura de árbol de análisis sintáctico. Inicialmente la pila contiene $ y el buffer de entrada, la cadena *s*. Ver la figura 7.

Para realizar el análisis sintáctico, se ejecuta algoritmo en pseudo código presentado en la figura 8.

En la figura 9, se presenta el código en java que implementa el algoritmo para evaluar expresiones.

Tabla Driven

Las gramáticas de operadores describen un lenguaje de propósitos especiales, o un subjuego de un lenguaje típico de programación, como por ejemplo snoboll.

Las relaciones de precedencia, son especificadas entre un conjunto de operadores, para esa gramática.

En la precedencia de operadores, el string es capturado de izquierda a derecha; existiendo dos estados esenciales:

Estado 1: esperando un operador
Estado 2: esperando un operando

El análisis sintáctico, propone dos pilas: una para operandos y otra para operadores; luego compara la precedencia entre operadores consecutivos. Cuando un operador de más alta precedencia, se encuentra en la cima (el top o tope de la pila), entonces el handle, llamado también mango; como lo llamaremos de aquí en adelante, consiste de dos operandos, combinados con este operador.

Se formaliza las relaciones de precedencia, por la introducción de 3 relaciones disjuntas:

$<\cdot$ $\quad (\doteq)$ $\quad \cdot>$

```
// The only publicy visible routine
    /**
     * Public routine that performs the evaluation.
     * Examine the  postfix machine to see if a single result is
     * left and if so, return it; otherwise print error.
     * @return the result.
     */
    public double getValue(double x)
    {
              return getValue(x,0,0);
    }
    public double getValue(double x,double y)
    {
              return getValue(x,y,0);
    }
    public double getValue(double x,double y,double z)
    {
    // for each call
    opStack = new Stack( );
    postfixStack = new Stack( );
    str = new StringTokenizer(string,"+*-/^()xyz ",true);
    opStack.push( new Integer( EOL ) );

    EvalTokenizer tok = new EvalTokenizer(str,x,y,z);
    Token lastToken;
    do
    {
      lastToken = tok.getToken( );
      processToken( lastToken );
    } while( lastToken.getType( ) != EOL );

    if( postfixStack.isEmpty( ) )
    {
      System.err.println( "Missing operand!" );
      return 0;
    }

    double theResult = postFixTopAndPop( );
    if( !postfixStack.isEmpty( ) )
    System.err.println( "Warning: missing operators!" );

    return theResult;
    }
    /**
     * Internal method that unary to binary.
    */
              private String unary2Binary(String s)
              {
        int i;
        s = s.trim();
        if(s.charAt(0) == '-')
              s = "0.0"+s;
        while((i = s.indexOf("(-")>=0)
            s = s.substring(0,i+1)+"0.0"+
           s.substring(i+1);
        return s;
    }
    /**
      * Internal method that hides type-casting.
     */
      private double postFixTopAndPop( )
    {
         return ((Double)(postfixStack.pop())).doubleValue( );
    }
    /**
      * Another internal method that hides type-casting.
     */
     private int opStackTop( )
    {
        return ( (Integer) ( opStack.peek( ) ) ).intValue( );
    }
```

```
  /**
    * After a token is read, use operator precedence parsing
    * algorithm to process it; missing opening parentheses
   * are detected here.
   */
   private void processToken( Token lastToken )
   {
       int topOp;
        int lastType = lastToken.getType();

        switch(lastType)
       {
        case VALUE:
           postfixStack.push( new Double( lastToken.getValue( ) ) );
          return;

        case CPAREN:
           while((topOp = opStackTop( ) ) != OPAREN && topOp
!= EOL)
              binaryOp( topOp );
          if( topOp == OPAREN )
             opStack.pop( );  // Get rid of opening parentheseis
          else
              System.err.println( "Missing open parenthesis" );
          break;

        default:    // General operator case
             int last=(lastType>=FUNCT?FUNCT:lastType);
            while(precTable[last].inputSymbol<=
precTable[opStackTop()>=FUNCT?FUNCT:opStackTop()].topOfStack)
              binaryOp( opStackTop() );
           if( lastType != EOL )
              opStack.push( new Integer( lastType ));
          break;
       }
   }
   /*
     * topAndPop the postfix machine stack; return the result.
     * If the stack is empty, print an error message.
    */
   private double getTop( )
   {
       if ( postfixStack.isEmpty( ) )
       {
           System.err.println( "Missing operand" );
          return 0;
       }
        return postFixTopAndPop( );
   }
   /**
     * Internal routine to compute x^n.
    */
   private static double pow( double x, double n )
   {
      if( x == 0 )
      {
         if( n == 0 )
             System.err.println( "0^0 is undefined" );
         return 0;
      }
      if( n < 0 )
      {
          System.err.println( "Negative exponent" );
         return 0;
      }
      if( n == 0 )
         return 1;
      if( n % 2 == 0 )
         return pow( x * x, n / 2  );
      else
         return x * pow( x, n - 1 );
   }
```

Figura 9. Código en Java para evaluar expresiones

donde $a <\cdot b$ significa, que *a* tiene precedencia menor que *b*, $a \doteq b$, significa que *a* tiene la misma precedencia que *b*, y $a \cdot> b$ significa, que *a* tiene mayor precedencia que *b*. Por ejemplo,

$$+ <\cdot * , (\doteq), \text{ y } * \cdot> +$$

En la entrada, el string:

(id + id)

las relaciones de precedencia entre los elementos del string:

$$\$ <\cdot (<\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot>) \cdot> \$$$

y haciendo uso del algoritmo parsing de precedencia de operadores, se tiene:

1. Explorar el primer $\cdot>$ y regresar al más cercano $<\cdot$

$$\$ <\cdot (<\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot>) \cdot> \$$$
$$\quad\quad | \quad\quad |$$

2. Identificar el primer **id** como el mango y utilizando la producción de $\mathbf{E} \rightarrow \mathbf{id}$ la gramática se reduce a **E**:

$$\begin{array}{c} E \\ | \\ \$ (\textbf{id} + \textbf{id}) \$ \end{array}$$

3. Ignorar el no terminal para reinsertar, las relaciones de precedencia

$$\$ <\cdot (<\cdot \textbf{id} \ + <\cdot \textbf{id} \cdot>) \cdot> \$$$
$$\quad\quad\quad\quad | \quad\quad |$$

con lo cual el siguiente identificador **id** es ahora el mango:

$$\begin{array}{c} E \quad E \\ | \quad\ | \\ \$ (\textbf{id} \ + \ \textbf{id}) \$ \end{array}$$

luego, la forma de la sentencia es:

$$\$ (E + E) \$$$

Repitiendo el algoritmo

$$\$ <\cdot (<\cdot + \cdot>) \cdot> \$$$
$$\quad\quad\quad | \quad\quad |$$

el mango es ahora E + E; es decir

$$\$ (E) \$$$

El proceso continua y como $E \rightarrow (E)$ se tiene,

$$\$ <\cdot (\doteq) \cdot> \$$$
$$\quad\quad | \quad\quad\quad |$$

$$\$ \ E \ \$$$

Finalizando; el árbol del parser es el de la figura 10.

Las precedencias para la gramática G se presenta en el cuadro 1.

Haciendo uso del cuadro 1 y del algoritmo de la figura 11 se procede a construir el árbol de la figura 10.

Construcción de la tabla de precedencia
Sean las operaciones Leading y trailing:

1. Leading (N), donde N es un no terminal, y corresponde al conjunto de todos los terminales que aparecen primeros en una forma sentencial derivable de N.
2. Trailing (N), donde N es un no terminal, y corresponde al conjunto de todos los terminales que aparecen últimos, en una forma sentencial derivable de N.

Usando la gramática G, del cuadro 1, se tiene:

Leading (E) = { + , * , (, id }
Trailing (E) = { + , * ,) , id }

Una producción de la forma: $N \rightarrow a\ A\ t\ B\ \beta$. Donde t, es un terminal y A y B son no terminales; cumple con la relación:

$$\text{Trailing (A)} \cdot> t <\cdot \text{Leading (B)}$$

Al respecto ver la figura 12.

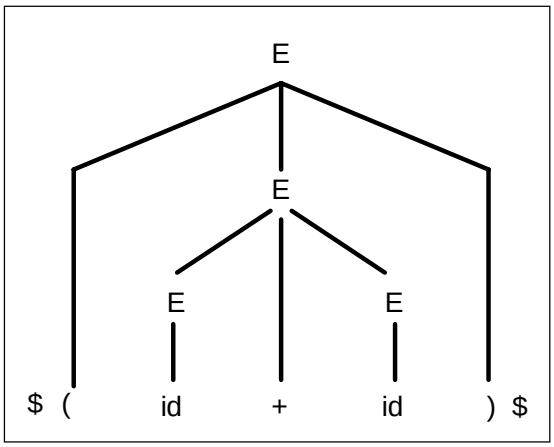

Figura 10. Árbol parser para la gramática G

Cuadro 1. Precedencias para la gramática G

Gramática G
E → E + E
E → E * E
E → (E)
E → id

	(	id	*	+	)	$
)			·>	·>	·>	·>
Id			·>	·>	·>	·>
*	<·	<·	·>	·>	·>	·>
+	<·	<·	<·	·>	·>	·>
(	<·	<·	<·	<·	$\doteq$	
$	<·	<·	<·	<·		$\doteq$

Algoritmo parsing de precedencia
Insertar relaciones de precedencia
while la entrada ? $ E $
    capturar el .> más a la izquierda
    capturar el primer <. cercano a la izquierda
    reducir el mango
    reinsertar las relaciones de precedencia ignorando no terminales
end while
End Algoritmo parsing de precedencia

Figura 11. Algoritmo parsing de precedencia

Se plantean las siguientes reglas:

Regla 1: t <· Leading (B)
Regla 2: Trailing (A) ·> t
Regla 3: Si t_1 y t_2, ambos aparecen en el lado de la mano derecha de la misma producción, entonces $t_1 \doteq t_2$

En la figura 13, se plantea el algoritmo para construir la tabla de precedencia. Aplicando el algoritmo de construcción de la tabla de precedencia a la producción E → E + E, se tiene la figura 14.

Funciones de precedencia

Los compiladores que utilizan parser por precedencia de operadores, utilizan funciones de precedencia antes que tablas de precedencia. Las funciones de precedencia *f* y *g* transforman símbolos terminales en enteros. Las funciones *f* y *g*, actúan ante los símbolos *a* y *b* como:

1. f (a) < g (b), si a <· b,
2. f (a) = g (b), si a $\doteq$ b, y
3. f (a) > g (b), si a ·> b

El algoritmo para encontrar las funciones de precedencia, usa como insumo la tabla de precedencia de operadores; y se denomina algoritmo de construcción de funciones de precedencia. En la figura 16, se presenta el grafo de funciones de precedencia aplicando el algoritmo a la gramática G siguiente.

Gramática G:
E → E + E
E → E - E
E → E * E
E → E / E
E → E ^ E
E → (E)
E → id

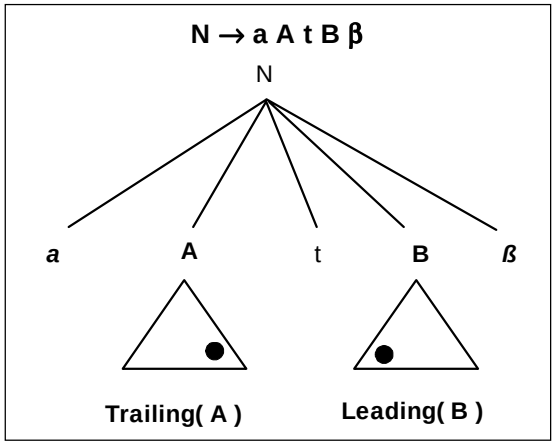

Figura 12. Trailing (A) .> t <.Leading (B)

Algoritmo construcción tabla de precedencia

for cada producción de la forma N → α A t B β
computar Leading (B)
computar Trailing (A)
aplicar las reglas 1, 2 y 3
$ <· otros terminales
end for
End Algoritmo construcción tabla de precedencia

Figura 13. Algoritmo de construcción de la tabla de precedencia

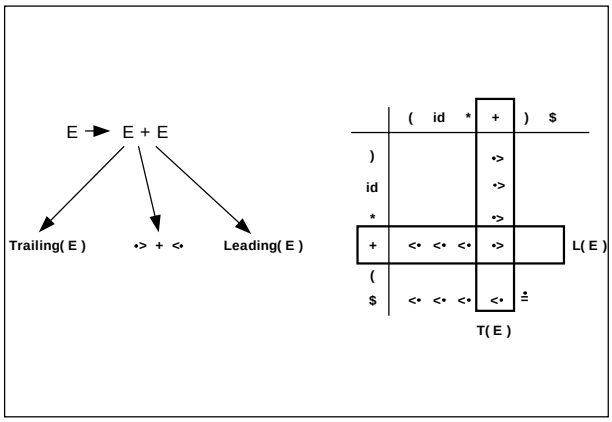

Figura 12. Aplicación del algoritmo de construcción para la producción: $E \rightarrow E + E$

Algoritmo de construcción de funciones de precedencia

1. Crear los símbolos fa y ga, para cada a que sea un terminal o $.
2. Crear un grafo dirigido, cuyos nodos sean los símbolos fa y ga. Si a <. b, colóquese una arista desde gb. a fa. Si a .> b; colóquese una arista de fa a gb
3. Si no hay ciclos, f(a) es la longitud del camino mas largo que comienza en fa

End Algoritmo de construcción de funciones de precedencia

Figura 15. Algoritmo de construcción de funciones de precedencia

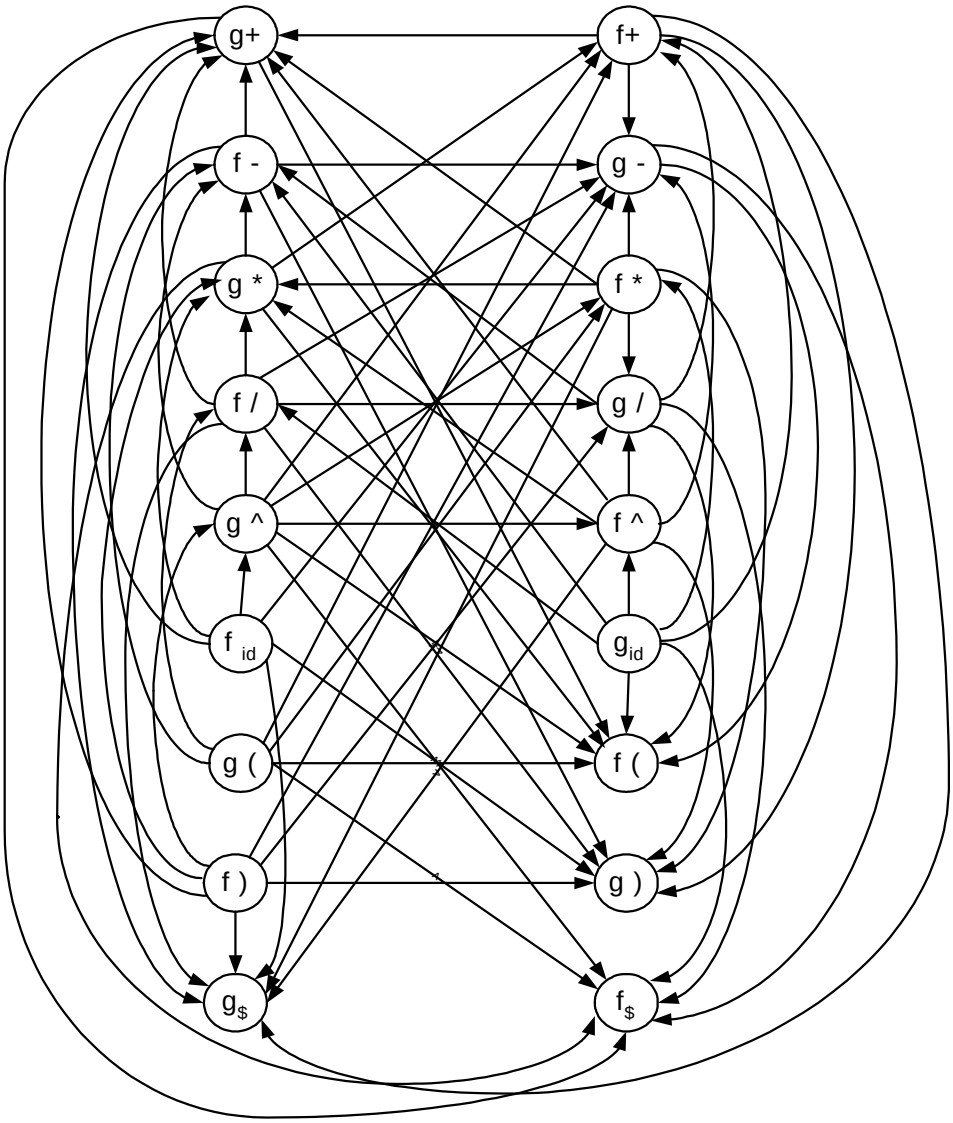

Figura 16. Grafo de funciones de precedencia

Cuadro 2. Funciones de precedencia

	+	-	*	/	^	(	)	Id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

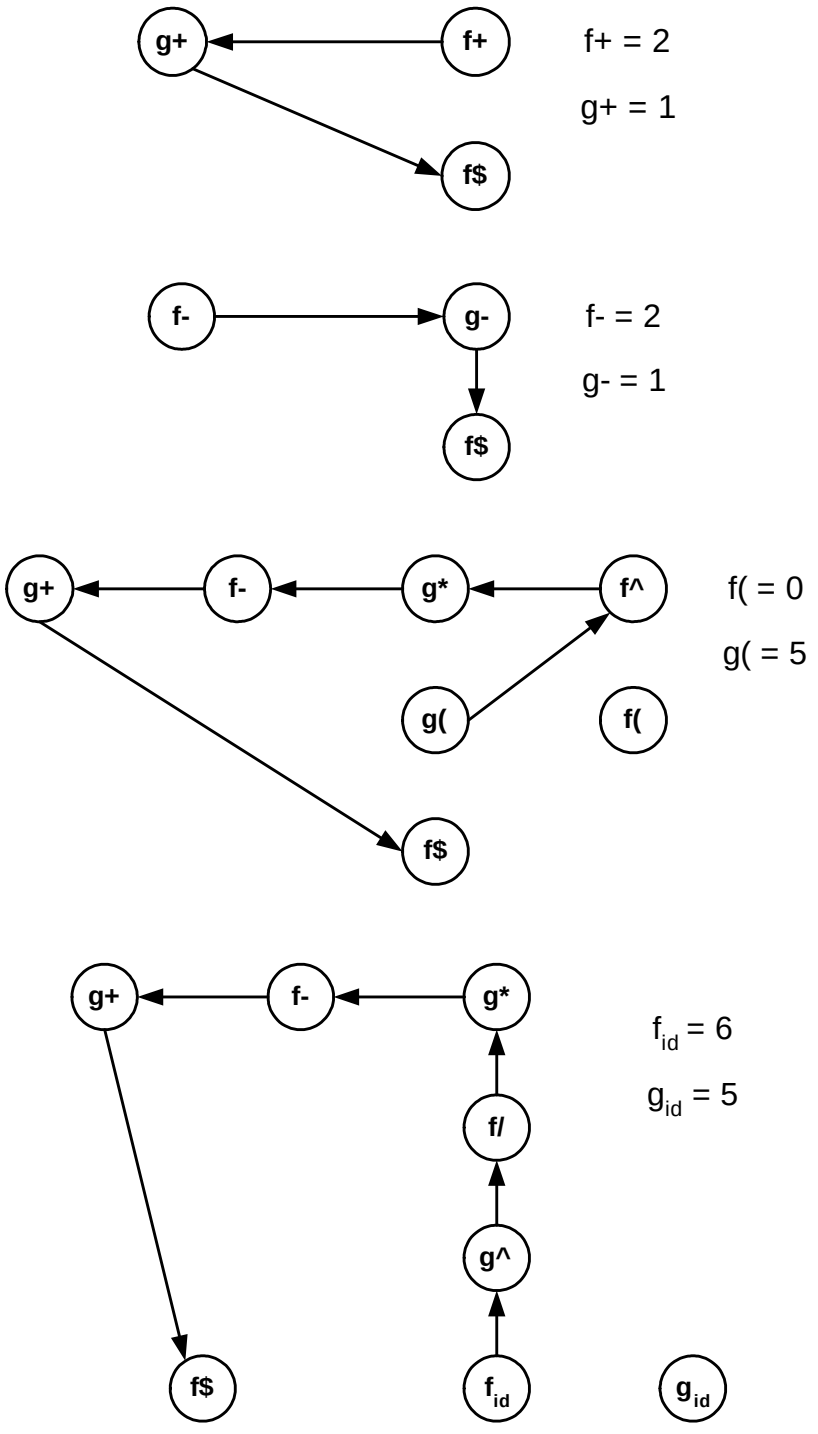

Figura 17. Algunos cálculos de funciones de precedencia

```
private static class Precedence
   {
       public int inputSymbol;
       public int topOfStack;

       public Precedence( int inSymbol, int topSymbol
)
       {
          inputSymbol = inSymbol;
          topOfStack  = topSymbol;
       }
   }
    // PrecTable matches order of Token enumeration
     private static Precedence [ ] precTable = new
Precedence[ ]
   {
      new Precedence(   0, -1 ),  // EOL
      new Precedence(   0,  0 ),  // VALUE
      new Precedence( 100,  0 ),  // OPAREN
      new Precedence(   0, 99 ),  // CPAREN
      new Precedence(   6,  5 ),  // EXP
      new Precedence(   3,  4 ),  // MULT
      new Precedence(   3,  4 ),  // DIV
      new Precedence(   1,  2 ),  // PLUS
      new Precedence(   1,  2 ),  // MINUS
      new Precedence(   7,  6 )   // FUNCT
   };
```

Figura 18. La clase Precedence

```
private static class Token
   {   public Token( )
       {
          this( EOL );
       }
       public Token( int t )
       {
          this( t, 0 );
       }
       public Token( int t, double v )
       {
          type = t;
          value = v;
       }
       public int getType( )
       {
          return type;
       }
       public double getValue( )
       {
          return value;
       }
       private int type = EOL;
       private double value = 0;
   }
```

Figura 19. La clase Token

```
private static class EvalTokenizer
   {   public EvalTokenizer( StringTokenizer is,double x,
                    double y,double z)
       {
          str = is;
          equis=x;
          ye=y;
          zeta=z;
       }
       /**
     * Find the next token, skipping blanks, and return it.
     * For VALUE token, place the processed value in
currentValue.
     * Print error message if input is unrecognized.
      */
       public Token getToken( )
       {
          double theValue;
          if(!str.hasMoreTokens( ))
             return new Token( );
          String s = str.nextToken();
          if( s.equals( " " ) ) return getToken( );
          if( s.equals( "^" ) ) return new Token(EXP);
          if( s.equals( "/" ) ) return new Token(DIV);
          if( s.equals( "*" ) ) return new Token(MULT);
          if( s.equals( "(" ) ) return new Token(OPAREN);
          if( s.equals( ")" ) ) return new Token(CPAREN);
          if( s.equals( "+" ) ) return new Token(PLUS);
          if( s.equals( "-" ) ) return new Token(MINUS);
       if( s.equals( "x" ) ) return new Token(VALUE,equis);
          if( s.equals( "y" ) ) return new Token(VALUE,ye);
         if( s.equals( "z" ) ) return new Token(VALUE,zeta);
          if(Character.isLetter(s.charAt(0)))
       {
          int i=searchFunction(s);
          if(i>=0)
                    return new Token(FUNCT+i);
          else
          {
          System.err.println( "Parse error" );
          return new Token();
          }
       }
       try
       {
          theValue = Double.valueOf(s).doubleValue();
       }
       catch( NumberFormatException e )
       {
          System.err.println( "Parse error" );
          return new Token( );
       }
       return new Token( VALUE, theValue );
   }
          public int searchFunction(String s)
          {
       for(int i=0;i<function.length;i++)
          if(s.equals(function[i]))
             return i;
       return -1;
   }
    private StringTokenizer str;
    private double equis;
    private double ye;
    private double zeta;
}
```

Figura 20. La clase EvalTokenizer

Como no hay ciclos, existen las funciones de precedencia. Debido a que $f_{\$}$ y $g_{\$}$ no poseen aristas de salida, $f(\$) = g(\$) = 0$.

El cuadro 2, es el resultado a aplicar el paso 3, del algoritmo de construcción de funciones de precedencia. Algunos cálculos de funciones de precedencia, se muestran en la figura 17. El código en Java que realiza los cálculos de las funciones de precedencia se presenta en la figura 18.

ANÁLISIS LEXICOGRÁFICO

Existen tres métodos generales de implantación de un analizador léxico. [AHO]

1. Utilizar un generador de analizadores léxicos, como el compilador LEX, utilizado para producir el analizador léxico a partir de una configuración basada en expresiones regulares. En este caso, el generador proporciona rutinas para la entrada y manejarla con buffers.
2. Escribir el analizador léxico en un lenguaje convencional de programación de sistemas, utilizando las posibilidades de entrada y salida de este lenguaje para leer la entrada.
3. Escribir el analizador léxico en lenguaje ensamblador y manejar explícitamente la lectura de la entrada.

Para el desarrollo del trabajo, se considera el segundo método y el lenguaje a utilizar es el JAVA, de Sun. Muchos compiladores son del tipo parser-driven, lo cual significa que el analizador sintáctico llama al analizador lexical.

Los token, son la unidad básica lexical; así como las palabras y la puntuación son las unidades básicas en una sentencia; sea en el lenguaje español o en el inglés. La figura 19, presenta la clase Token. Los token son descritos en dos partes; un tipo y un valor: Token = (type, value).

Por ejemplo:

```
Token ( MINUS );
Token ( VALUE, equiz);
Token ( VALUE, zeta);
```

Esto se presenta en la clase EvalTokenizer mostrada en la figura 20.

RESULTADOS

Las pruebas hechas a la clase parser presentada, indican que esta es robusta y de calidad frente a entradas como las siguientes:

Cuadro 3. Resultados de evaluación para una ecuación diferencial ordinaria

x_k	y_k
0.0	1.00000
0.5	0.50000
1.0	0.31250
1.5	0.31250
2.0	0.50781

a. 3.14159*(1+(x/2)^2)^2
Expresión ingresada para evaluar la integral de dicha función en el intervalo [0, 2] con n = 16, con el método de Trapecio; siendo el resultado 11.744961918792724, el cual es el esperado y correcto.

b. 1+eXp(0-x)*sin(4*x)
Expresión ingresada para evaluar la integral de dicha función en el intervalo [0, 1] con n = 3, aplicándole método de Simpson 3/8; se obtiene como resultado 1.3143968149336276, el cual es el esperado y correcto.

c. y*(x*x-1)
Expresión ingresada para evaluar la ecuación diferencial ordinaria del problema del valor inicial con y0=1.0, el intervalo [0, 2], n = 4; y evaluado con el método de Runge Kutta 2, se obtienen los siguientes puntos de evaluación.

CONCLUSIONES

El programa presentado consigue el objetivo de escribir un analizador léxico en un lenguaje convencional de sistemas para evaluar expresiones y/o funciones desde la entrada. Los resultados obtenidos con las pruebas realizadas, son los esperados y correctos teniendo muy buena precisión y exactitud.

REFERENCIAS BIBLIOGRAFICAS

1. Aho, Alfred V.; Sethi, Ravi; Ulman, Jeffrey D. 1990. “Compiladores: Principios, técnicas y herramientas” 1ra.edición, Addison-Wesley Iberoamericana S.A. USA.

2. Deitel, Harvey & Deitel, Paul. 2003. “Java How to program” Fith Edition, USA.

3. Palmer, Grant. 2003. “Technical Java Developing Scientific and Engineering Applications” First edition, Pearson Education Inc., USA.

4. Weiss, Mark Allen. 2000. “Estructura de datos en Java™” 1ra. edición, Pearson Educación S.A.