IBM

developer**Works**®

Evaluate mathematical expressions quickly and accurately

With this handy applet, you can evaluate mathematical expressions step by step

Nikola Stepan (nikola.stepan@vz.tel.hr)
Software Engineer
ABIT Ltd.

10 September 2001

Computer science's traditional methods for evaluating mathematical expressions are awkward and difficult for the untrained user; software engineer Nikola Stepan aims to change all that. His applet W3Eval evaluates expressions using the same sequence of steps you would on with a paper and pencil -- but with much faster and with no mistakes in arithmetic. Read on to learn about the challenges involved in transforming human-readable mathematics into Java code.

Remember struggling with reverse Polish notation on your first scientific calculator? The W3Eval applet can't make your trusty HP-41 easier to use -- as its name implies, it's an expression evaluator that runs only on the Web. But it does offer one method for evaluating expressions in a step-by-step fashion that's easier for human beings to follow.

W3Eval's approach is different from that of a conventional calculator and is more in tune with the way people calculate. When you calculate on a conventional calculator, each time you enter a new number you lose sight of the previous one. And if you make a mistake in the middle of a long expression, you have to start all over. With W3Eval, you can see everything you are calculating and easily edit expressions. Its unique ability to evaluate expressions step by step is very handy, because the user can see each step of evaluation, together with temporary results.

This article will walk you through the highlights of W3Eval's functionality; you'll be able to look at some of the code it uses to evaluate expressions. First, however, we'll look at the classical algorithms for evaluating expressions, so you can see how W3Eval's approach differs.

Classic algorithms for expression evaluation

The classic approach to writing code to evaluate arithmetic expressions was described by Donald Knuth in 1962 (see Resources). Knuth outlines three steps:

- Parsing an infix expression
- Converting infix expression to a postfix expression
- Evaluating the postfix expression

Note that our discussion of this classic approach will be somewhat simplified: arithmetic expressions will contain only operands, binary operators, and one kind of parentheses. Additionally, all operands and operators will be represented by a single character, making parsing trivial.

Expression notations

The three most common forms of notation in arithmetic expressions are *infix, prefix,* and *postfix* notations. Infix notation is a common way of writing expressions, while prefix and postfix notations are primarily used in computer science.

Infix notation

Infix notation is the conventional notation for arithmetic expressions. It is called *infix* notation because each operator is placed between its operands, which is possible only when an operator has exactly two operands (as in the case with binary operators such as addition, subtraction, multiplication, division, and modulo). When parsing expressions written in infix notation, you need parentheses and precedence rules to remove ambiguity.

```
Syntax: operand1 operator operand2
Example: (A+B)*C-D/(E+F)
```

Prefix notation

In prefix notation, the operator is written before its operands. This notation is frequently used in computer science, especially in compiler design. It is also called *Polish notation* in honor of its inventor, Jan Lukasiewicz (see Resources).

```
Syntax  : operator operand1 operand2
Example : -*+ABC/D+EF
```

Postfix notation

In postfix notation, the operator comes after its operands. Postfix notation is also known as *reverse Polish notation* (RPN) and is commonly used because it enables easy evaluation of expressions.

```
Syntax  : operand1 operand2 operator
Example : AB+C*DEF+/-
```

Prefix and postfix notation have three common features:

- The operands are in the same order that they would be in the equivalent infix expression.
- Parentheses are not needed.
- The priority of the operators is irrelevant.

Converting infix notation to postfix notation

To convert an expression in an infix expression to its equivalent in postfix notation, we must know the precedence and associativity of operators. *Precedence* or operator strength determines order

of evaluation; an operator with higher precedence is evaluated before one of lower precedence. If the operators all have the same precedence, then the order of evaluation depends on their *associativity.* The associativity of an operator defines the order in which operators of the same precedence are grouped (right-to-left or left-to-right).

```
Left associativity  : A+B+C = (A+B)+C
Right associativity : A^B^C = A^(B^C)
```

The conversion process involves reading the operands, operators, and parentheses of the infix expression using the following algorithm:

1. Initialize an empty stack and empty result string variable.
2. Read the infix expression from left to right, one character at a time.
3. If the character is an operand, append it to the result string.
4. If the character is an operator, pop operators until you reach an opening parenthesis, an operator of lower precedence, or a right associative symbol of equal precedence. Push the operator onto the stack.
5. If the character is an opening parenthesis, push it onto the stack.
6. If the character is a closing parenthesis, pop all operators until you reach an opening parenthesis and append them to the result string.
7. If the end of the input string is found, pop all operators and append them to the result string.

Postfix expression evaluation

Evaluating a postfix expression is simpler than directly evaluating an infix expression. In postfix notation, the need for parentheses is eliminated and the priority of the operators is no longer relevant. You can use the following algorithm to evaluate postfix expressions:

1. Initialize an empty stack.
2. Read the postfix expression from left to right.
3. If the character is an operand, push it onto the stack.
4. If the character is an operator, pop two operands, perform the appropriate operation, and then push the result onto the stack. If you could not pop two operators, the syntax of the postfix expression was not correct.
5. At the end of the postfix expression, pop a result from the stack. If the postfix expression was correctly formed, the stack should be empty.

W3Eval: A new approach

W3Eval's approach differs from the classic algorithms outlined above. It does not convert infix expressions to postfix notation; rather, it evaluates the infix expression directly. This approach is slightly more complicated than the traditional one, but it provides step-by-step evaluation, and you can see each step as it executes. The process of evaluation is similar to calculating by hand: while the expression contains parentheses, the subexpression inside the deepest nested pair of parentheses is evaluated. When all parenthetical subexpressions have been evaluated, the rest of the expression is evaluated.

The process of evaluation is divided into three steps:

1. Expression parsing
2. Expression checking
3. Step-by-step evaluation

Expression parsing

Mathematical expressions in W3Eval are built of numbers, variables, operators, functions, and parentheses. In addition to the default decimal base, W3Eval also supports binary, octal, and hexadecimal bases. Numbers in these other bases must be preceded by the character `#` followed by `b`, `o`, or `h` for binary, octal, or hexadecimal, respectively.

A variable in W3Eval is an unlimited-length sequence of uppercase letters and digits, the first of which must be a letter. W3Eval has some predefined variables, but it also supports user-defined variables.

W3Eval supports functions with fixed and variable number of arguments. Functions are divided into following groups:

- Trigonometric (sin, cos, tan, cot, sec, csc)
- Inverse trigonometric (asin, acos, atan, atan2, acot, asec, acsc)
- Hyperbolic (sinh, cosh, tanh, coth, sech, csch)
- Inverse hyperbolic (asinh, acosh, atanh, acoth, asech, acsch)
- Exponential (log, log2, log10, exp, exp2, exp10, sqrt, cur)
- Combinatoric (comb, combr, perm, permr, var, varr)
- Statistical (sum, avg, min, max, stddev, count)
- Other (abs, ceil, fact, floor, pow, random, rint, round, sign, frac, hypot, deg, rad, trunc, int)

When we say that W3Eval *parses* an expression, we mean that it recognizes the expression's arithmetical components, transforms them into tokens, and puts them in a vector. Once the expression is in this state, it is ready for the next two steps: expression checking and evaluation.

Tokens are the constituent parts of an arithmetical expression in W3Eval; *marks* are individual characters used by the applet as internal flags identifying kinds of tokens. Each kind of token has its unique mark. Expressions in W3Eval are built from the tokens shown in Table 1.

Table 1. Tokens in W3Eval

Token	Mark	Class
Decimal number	D	`Double`
Binary number	B	`String`
Hexadecimal number	H	`String`
Octal number	O	`String`
Variable	V	Variable
Function	F	`Function`
Operator	P	`Operator`
Open parenthesis	(	`String`

| Close parenthesis |) | String |
| Comma | Z | String |

Listing 1 shows the definitions of the classes used for representing functions, operators, and variables:

Listing 1. The Function, Operator, and Variable class definitions

```
public class Function
   {
   public String function;
   public int number_of_arguments;

   public Function( String function, int number_of_arguments )
      {
      this.function=function;
      this.number_of_arguments=number_of_arguments;
      }

   public String toString()
      {
      return function;
      }
   }

public class Operator
   {
   public String operator;
   public byte priority;

   public Operator( String operator, byte priority )
      {
      this.operator=operator;
      this.priority=priority;
      }

   public String toString()
      {
      return operator;
      }
   }

public class Variable
   {
   public String variable;
   public double value;

   public Variable( String variable, double value )
      {
      this.variable=variable;
      this.value=value;
      }

   public String toString()
      {
      return variable;
      }
   }
```

The `Token` class is shown in Listing 2.

Listing 2. Token class

```
public class Token
```

```
   {
   public Object token;
   public char mark;
   public int position;
   public int length;

   public Token ( Object token, char mark, int position, int length )
      {
      this.token=token;
      this.mark=mark;
      this.position=position;
      this.length=length;
      }

   public String toString()
      {
      return token.toString()+" ; "+mark+" ; "+position+" ; "+length+"
";
      }
   }
```

Expression checking

All of the code that checks formal expression validity is in a single class. A detailed expression check can determine the precise type and location of an error. There are seven categories of error checking:

Parentheses check. Expressions in W3Eval can contain three kind of parentheses: standard parentheses, square brackets, and curly braces. An expression's parenthesis syntax is correct if the expression contains equal number of opening and closing parentheses, with each opening parenthesis matching a corresponding closing parenthesis of the same type. All three kind of parentheses are semantically equivalent and are shown in the snippet below.

Listing 3. Three kinds of parentheses

```
import java.util.Stack;

public class Parentheses_check
   {

   public static boolean is_open_parenthesis( char c )
      {
      if ( c=='(' || c=='[' || c=='{' )
         return true;
      else
         return false;
      }

   public static boolean is_closed_parenthesis( char c )
      {
      if ( c==')' || c==']' || c=='}' )
         return true;
      else
         return false;
      }

   private static boolean parentheses_match( char open, char closed )
      {
      if ( open=='(' && closed==')' )
         return true;
      else if ( open=='[' && closed==']' )
         return true;
      else if ( open=='{' && closed=='}' )
```

```
          return true;
      else
          return false;
      }

  public static boolean parentheses_valid( String exp )
      {
      Stack       s = new Stack();
      int         i;
      char        current_char;
      Character   c;
      char        c1;
      boolean     ret=true;

      for ( i=0; i < exp.length(); i++ )
          {

          current_char=exp.charAt( i );

          if ( is_open_parenthesis( current_char ) )
              {
              c=new Character( current_char );
              s.push( c );
              }
          else if ( is_closed_parenthesis( current_char ) )
              {
              if ( s.isEmpty() )
                  {
                  ret=false;
                  break;
                  }
              else
                  {
                  c=(Character)s.pop();
                  c1=c.charValue();
                  if ( !parentheses_match( c1, current_char ) )
                      {
                      ret=false;
                      break;
                      }
                  }
              }
          }

      if ( !s.isEmpty() )
          ret=false;

      return ret;
      }
  }
```

Token check. Checks the syntax of the expression. Makes sure that all parts of the expression are recognized as legal tokens.

Beginning of expression check (see Listing 4)**.** Makes sure that the expression begins with a legal token. Expressions must not begin with an operator, comma, or closing parenthesis.

End of expression check. Makes sure that the expression ends with a legal token. Expressions must not end with an operator, function, comma, or opening parenthesis.

Token sequence check. Checks the sequence of tokens in an expression. In the table below, the token on the X-axis can follow the token on the Y-axis if their meeting place is marked with an X.

Table 2. Legal token sequences

D	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
B	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
H	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
O	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
V	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
F	Â	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â
P	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â	Â Ã#	Â	Â
(	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â	Â Ã#	Â	Â
)	Â	Â	Â	Â	Â	Â	Â Ã#	Â	Â Ã#	Â Ã#
Z	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â Ã#	Â	Â Ã#	Â	Â

Function check. Ensures that all functions in an expression have the correct number of arguments.

Comma check. A comma should only be used to separate function arguments. If it is found in any other place in an expression, it is illegal.

Step-by-step evaluation

W3Eval only evaluates expressions that successfully pass all the checks outlined above. This ensures that the assumptions built into W3Eval will not cause problems. The following algorithm is used to perform a single step of the evaluation of an expression:

1. Find the deepest nested pair of parentheses.
2. Inside that pair of parentheses, find the operator with highest priority.
3. If there are no operators inside that pair of parentheses:
   - If the expression doesn't contain any more parentheses, the process of evaluation is finished.
   - If the expression contains parentheses but no operators, there is a function here. Evaluate the function and go to step 5.
4. Get operands and perform the operation.
5. Remove used tokens from the vector and put the result in their place.
6. Remove redundant parentheses.
7. Join remaining tokens in the vector into the string and display the result on the screen.

We'll now go through each step of the algorithm in more detail, examining most of the interesting code snippets along the way.

Step 1: To avoid dealing with parentheses, W3Eval determines which subexpression lies within the deepest nested pair of parentheses. Two steps are required for that task. First, W3Eval must find the first closing parenthesis:

Listing 5. Find the first closing parenthesis

```
public static int pos_first_closed_parenthesis( Vector tokens )
   {
   Token   t;

   for ( int i=0; i<tokens.size(); i++ )
      {
      t=(Token)tokens.elementAt( i );
      if ( t.mark==')' )
         return i;
      }
   return 0;
   }
```

Next, it finds the opening parenthesis that matches the previously found closing parenthesis, as shown in .

Step 2: To perform a single step of an evaluation, W3Eval finds the operator with the highest priority inside the deepest nested pair of parentheses. (The priorities of operators are hardcoded into the applet.)

Listing 7. Find operator with highest priority

```
public static int pos_operator( Vector tokens, Range r )
   {
   byte     max_priority=Byte.MAX_VALUE;
   int      max_pos=0;

   byte     priority;
   String   operator;
   Token    t;

   for ( int i=r.start+2; i<=r.end-2; i++ )
      {
      t=(Token)tokens.elementAt( i );
      if ( t.mark!='P' )
         continue;
      priority=((Operator)t.token).priority;
      operator=((Operator)t.token).operator;

      if ( priority < max_priority || ( operator.equals("^") ||
         operator.equals("**") ) && priority == max_priority )
         {
         max_priority=priority;
         max_pos=i;
         }
      }
   return max_pos;
   }
```

Step 3: If the expression doesn't contain any more parentheses, the process of evaluation is finished. If the expression contains parentheses but no operators, there is a function here that needs to be evaluated.

Listing 8. Check if any more operators

```
...
int poz_max_op=pos_operator( tokens, range );
// if there are no operators
if ( poz_max_op==0 )
   {
```

```
   if ( no_more_parentheses )
      {
      return false;
      }
   else
      {
      double    result;
      result=function_result( tokens, range.start-1 );
      function_tokens_removal( tokens, range.start-1 );

      t = new Token ( new Double(result), 'D', 0, 0 );
      tokens.setElementAt( t, range.start-1 );

      parentheses_removal( tokens, range.start-1 );
      return true;
      }
   }
...
```

Step 4: All operators are binary, which means that the first operand is positioned before the operator, and the second operand is positioned after the operator.

Listing 9. Get operands and perform the operations

```
...
double operand1, operand2;

// first operand is before...
t=(Token)tokens.elementAt( poz_max_op-1 );
operand1=operand_value( t );

// ...and second operand is after operator
t=(Token)tokens.elementAt( poz_max_op+1 );
operand2=operand_value( t );

// operator
t=(Token)tokens.elementAt( poz_max_op );
String op=((Operator)t.token).operator;

double result=operation_result( operand1, operand2, op );

tokens.removeElementAt( poz_max_op+1 );
tokens.removeElementAt( poz_max_op );

t = new Token ( new Double(result), 'D', 0, 0 );
tokens.setElementAt( t, poz_max_op-1 );

parentheses_removal( tokens, poz_max_op-1 );
...
```

Operands can be variables or decimal, hexadecimal, octal, or binary numbers.

Listing 10. Get operands

```
public static double operand_value( Token t )
   {
   if ( t.mark=='V' )
      return ((Variable)t.token).value;
   else if ( t.mark=='D' )
      return ((Double)t.token).doubleValue();
   else if ( t.mark=='H' )
      return base_convert( ((String)t.token).substring(2), 16 );
   else if ( t.mark=='O' )
      return base_convert( ((String)t.token).substring(2), 8 );
   else if ( t.mark=='B' )
      return base_convert( ((String)t.token).substring(2), 2 );
   }
```

The next method converts numbers from the various numeric bases into decimal numbers.

Listing 11. Convert numbers to decimal numbers

```
public static long base_convert( String s, int base )
   {
   long r=0;
   int i, j;

   for ( i=s.length()-1, j=0; i>=0; i--, j++ )
      r=r+digit_weight( s.charAt( i ) )*(long)Math.pow( base, j );
   return r;
   }

public static int digit_weight( char c )
   {
   if ( Character.isDigit( c ) )
      return c-48;
   else if ( 'A'<=c && c<='f' )
      return c-55;
   else if ( 'a'<=c && c<='f' )
      return c-87;
   return -1;
   }
```

When we determine the operands and operator, we can perform the operation, as shown in Listing 12.

Step 5: In this step, W3Eval removes used tokens from the vector and puts the result in their place. In case of a function evaluation, the function, parentheses, arguments, and commas are removed; in case of an operator evaluation, the operands and operator are removed.

Step 6: At this point of the evaluation, W3Eval removes redundant parentheses from the expression.

Listing 13. Remove redundant parentheses

```
private static void parentheses_removal( Vector tokens, int pos )
   {
   if (
      pos>1 &&
      ((Token)tokens.elementAt( poz-2 )).mark!='F' &&
      ((Token)tokens.elementAt( poz-1 )).mark=='(' &&
      ((Token)tokens.elementAt( poz+1 )).mark==')'
      ||
      pos==1 &&
      ((Token)tokens.elementAt( 0 )).mark=='(' &&
      ((Token)tokens.elementAt( 2 )).mark==')'
      )
      {
      tokens.removeElementAt( poz+1 );
      tokens.removeElementAt( poz-1 );
      }
   return;
   }
```

Step 7: In the last step of the evaluation, the remaining tokens from vector are joined in a string and displayed on the screen.

Listing 14. Join tokens and display the result

```
public static String token_join( Vector tokens )
   {
   String   result=new String();
   Token    t;

   for ( int i=0; i < tokens.size(); i++ )
      {
      t=(Token)tokens.elementAt( i );

      if ( t.mark=='D' )
         {
         double n=((Double)t.token).doubleValue();
         result=result + formated_number( n );
         }
      else
         result=result + t.token;

      if ( result.endsWith( ".0" ) )
         result=result.substring( 0, result.length()-2 );
      result=result + " ";
      }
   return result;
   }
```

Conclusion

This article examined an applet that has the ability to evaluate arithmetic expressions step by step. We reviewed the most interesting code snippets along the way and discussed two different approaches to expression evaluation.

There are various potential enhancements for the next version of W3Eval, including the ability to add user-defined functions; support for fractions, complex numbers, and matrices; an improved GUI; size and speed optimization; and security enhacements. I encourage you to offer your own ideas for enhancements.

I hope you will find W3Eval to be a useful online tool for evaluating expressions in a way that is more natural than the classical method. I also hope the code and algorithms discussed here offered some insight into the ways that the Java language can help you deal with matematical problems.

Resources

- Read a biography of Polish mathematician Jan Lukasiewicz.
- Donald Knuth, a preeminent scholar of computer science, has written and lectured extensively in the design and analysis of algorithms. His home page offers links to recently published papers and information on his books.
- You may find this Java FAQ helpful in your programming endeavors.
- There's more good information on applets in *Just Java 2,* Peter Van Der Linden (Prentice Hall PTR/Sun Microsystems Press, December 1998).
- *The Java Programming Language* by Ken Arnold, James Gosling, and David Holmes (Addison Wesley, December 2000) contains good information on collections.
- Find more Java technology resources on the *developerWorks* Java technology zone.

About the author

Nikola Stepan

Nikola Stepan is a software engineer at ABIT Ltd., where he works on banking software design and development. He has extensive academic background in information systems and a broad range of programming experience, from low-level programming to information systems. His special interests are object-oriented programming languages, relational databases, Internet programming, and system programming. He received a B.S. in information systems from Faculty of Organisation and Informatic in Varazdin, Croatia, in 1999. He speaks Croatian, English, and a bit of German. Contact Nikola at nikola.stepan@vz.tel.hr.