Infix to Postfix Conversion

Computer Algorithm

20072605 Tae-Wan, Kim

Intelligent Multimedia Lab.
Dept. of Computer & Communication Engineering
POSTECH, South Korea
taey16@postech.ac.kr

April 28, 2007

1. Algebraic Expression Evaluation

1.1 Problem

The main application of **stack and binary expression tree is a algebraic expression evaluation.** This is a classic and important problem. What makes this problem particularly interesting is that the core of the solution requires two stacks, each holding different types of data or a simple parsing tree to evaluate an expression. Then, why we translate infix notation into postfix notation? The reason is that **infix is the format normally used in representing an algebraic expression by human. But postfix interpreted by machine is a format that places an operator directly after two operands without ambiguity.** we ,therefore, should develop a java software application that takes an algebraic expression as an input string. An example of such an algebraic expression is (1 + 2) * 2 - 1$\hat{3}$ % 2. After numeric values are assigned to each operand(values for 1,2,2,1,3,2), the algorithm must compute the value of the algebraic expression.

Input : A string representing an algebraic expression involving n operands and an n-tuple representing the values for the operands(i.e., numeric values for each operand). The arithmetic operations allowed in an expression are addition, subtraction, multiplication and division as well as exponentiation and modulus. Parentheses ,of course, should be evaluated.

Output : The postfix notation w.r.t. an expression and the value of the expression for the particular n-tuple of input operand values.

1.2 Solution of Problem using stack

1.2.1 Conversion from infix to postfix

Let us think about the algorithm to convert the infix string ”a*b+c” to postfix format. The symbol a is appended to the postfix string. The operator symbol ”*” is pushed onto the operator stack. The symbol b is appended to the postfix string. The operator symbol ”*” representing top is compared in precedence to the newly read operator symbol ”+”. Since the ”*” is not of lower or equal precedence, it it popped from the stack and appended to the postfix string(which is now ab*). The ”+” is pushed onto the operator stack. The final symbol c is appended to the postfix string(since it is an operand symbol). With all symbols read, the operator stack is popped(until empty) and in this case only the operator symbols ”+” is appended to the postfix string producing the final result ab*c+.

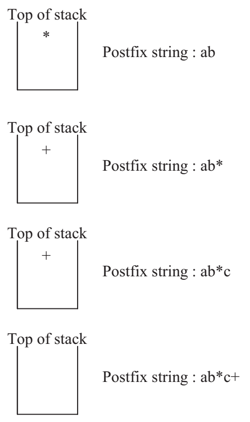

Figure 1: Use of the operator stack in converting from infix to postfix

The following algorithm shows a brief way to transform infix into postfix.

```
1. Tokenize the infix String into char \\
2. Loop for the size of infix
    if the current character in infix is a digit, copy it to the
    next element of postfix  \\
    if the current character in infix is a left parenthesis, push it
    on the stack.
    if the current character in infix is an operator
        pop operators at the top of the stack while they have equal
        or higher precedence than the current operator. and insert
        the popped operators in posfix
        push the current character in infix on the stack.
    if the current character in infix is a right parenthesis
        pop operators from the top of the stadck and insert them in
        postfix until a left parenthesis is at the top of the stack.
        pop(and discard) the left parenthesis from the stack
```

1.2.2 Evaluation of postfix expression

The following algorithm shows a brief way to evaluate postfix string

```
 1. Tokenize the postfix string into char
 2. Loop for the size of infix
    if the current character is a digit
```

```
        push its integer value on the stack
    otherwise, if the current character is an operator
        pop the two top elements of the stack into variables x and y.
        calculate y operator x.
        push the result of the calculation on the stack
```

1.2.3 Class diagram

Following class diagram shows our code very clearly.

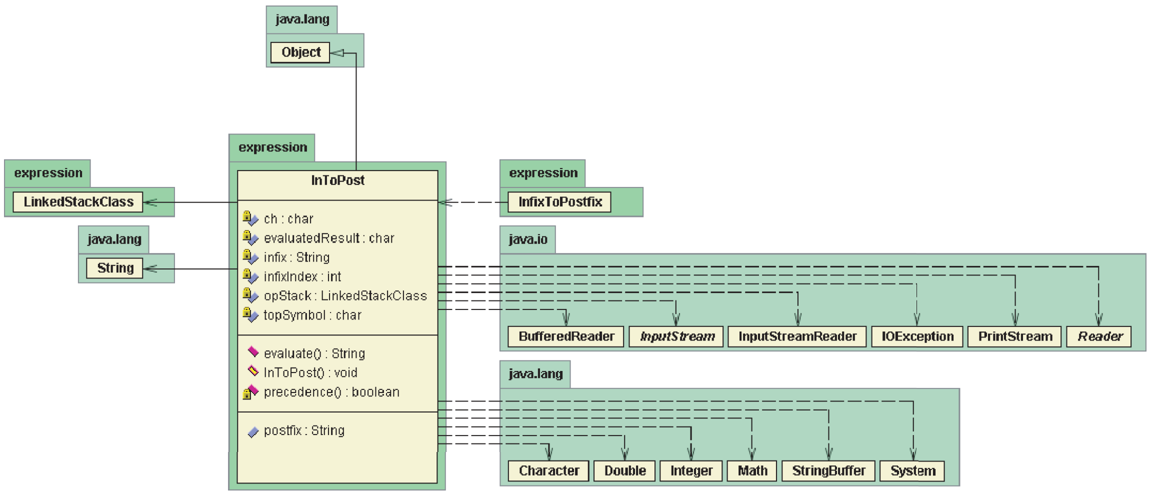

Figure 2: Class diagram for InfixToPostfix

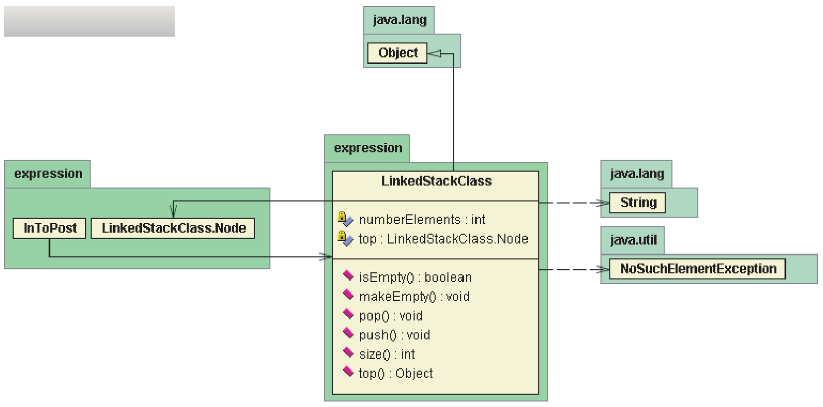

Figure 3: Class diagram for stack

1.2.3 Result

```
Input infix expression 2*3
Postfix : 23*
Value : 6
Input infix expression 4^(1+1)
Postfix : 411+^
Value : 16
Input infix expression 3+7*1%2
Postfix : 3712%*+
Value : 10
Input infix expression 6%8+4^(1+2)
Postfix : 68%412+^+
Value : 70
Input infix expression 7%6+2^(1+1)
Postfix : 76%211+^+
Value : 5
Input infix expression 2^(2+2)/(1-0)
Postfix : 222+^10-/
Value : 16
Input infix expression 9%3+6^1-3+(1-1)
Postfix : 93%61^3-11-++
Value : 3
Input infix expression (1+2)*3-4+1*2^(6/(2+1))
Postfix : 12+3*4-12621+/^*+
Value : 9
Input infix expression 1+(2+(3+(4+(5+(6^(1+2))))))
Postfix : 12345612+^+++++
Value : 231
Input infix expression 4-1%8+6%(3+1)*2+(4*(4+(2-5^2)))
Postfix : 418%631+%2*44252^-+*++-
Value : 75
```

1.3 Solution of Problem using binary tree

1.3.1 Building Expression Tree

Logic for *buildExpressionTree* is as follows. We iterate from left to right over the characters in *postfixString*. Each character in *postfixString* is assigned to *ch*. Character *ch* is either an operand or an operator as verified by supporting internal queries ch ≤ "9" && ch ≥ "0". Operands are encapsulated in instances of *SearchTreeNode* and pushed onto the stack *s*. Left and right subtrees of operand nodes are *null*. Operators are encapsulated in nodes with right and subtrees set sequentially to the top two nodes popped from stack *s*. This algorithm requires that *postfixString* be a valid postfix expression with no blank spaces. Successful completion of the algorithm leaves only a reference to the root node of a valid expression tree on stack *s*. Following figure shows how to build expression tree.

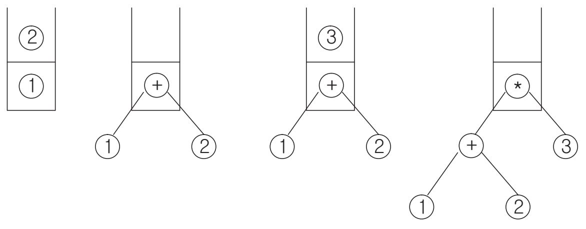

Figure 4: Converting "(1+2)*3" to 12+3x using binary expression tree and stack

1.3.2 Postorder Traversal

Following pseudo-code gives a chance to understanding the postorder traversal.

```
Algorithm postorder( Tree t )
    if t has a left child u then
        postorder( u );
    if t has a right child w then
        postorder( w );
    perform the "visit" action for node t
```

Fig5. shows exact order of postorder traversal

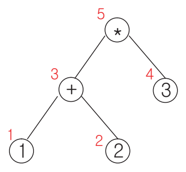

Figure 5: Postorder traversal on expression tree

1.3.3 Expression Tree Evaluation

Algorithm evaluateExpression, given in fig.5, evaluates the expression associated with the subtree rooted at a node v of an arithmetic expression tree T by performing a postorder traversal of T starting at v. In this case, the "visit" action consists of performing a single arithmetic operation. Note that we use the fact that an arithmetic expression tree is a proper binary tree.

```
Algorithm evaluateExpression(T,v)
  if (root.isInternal())
    Character op = (Character)root.contents;
    int x = evaluateExpression( root.left );
    int y = evaluateExpression( root.right);
    return x o y
  else
    return the value stored at v
```

1.3.3 Class Diagram

Following class diagram shows our code very clearly.

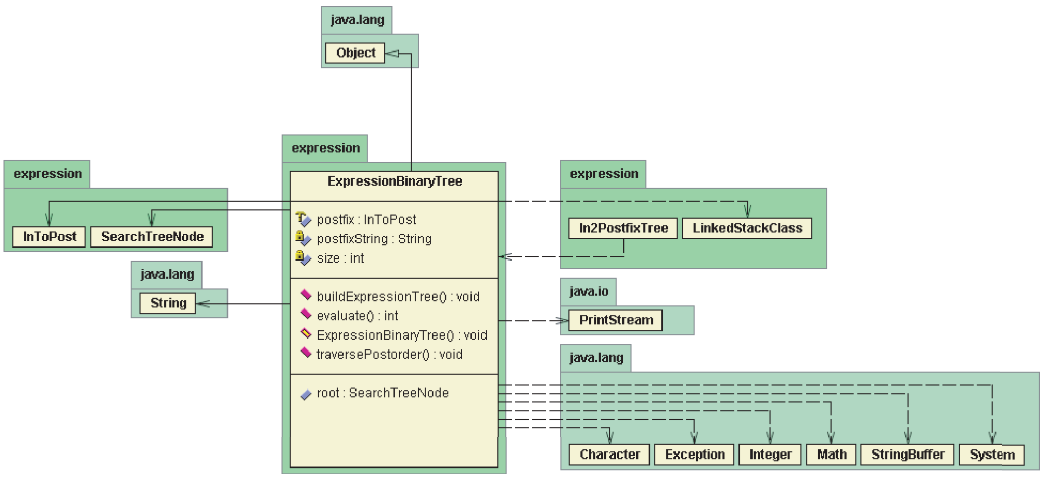

Figure 6: Class diagram for InfixToPostfix using binary expression tree

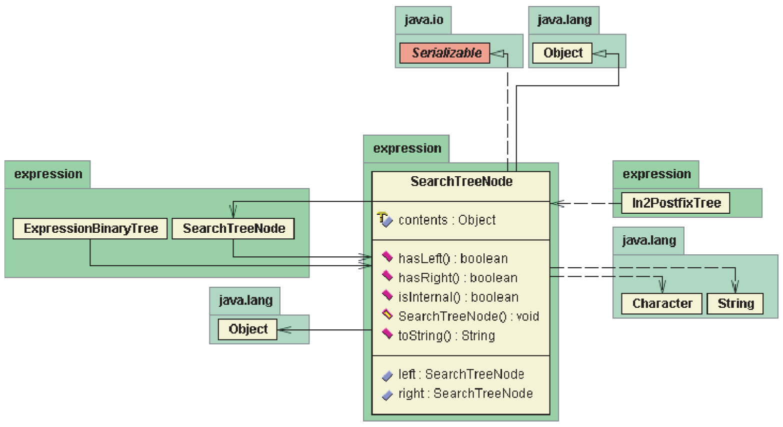

Figure 7: Class diagram for binary expression tree

1.4 Result

```
Input infix expression 5^8-5
5 8 ^ 5 -
Value : 390620
Input infix expression 6+5/5+9
6 5 5 / 9 + +
Value : 16
Input infix expression (6+5)/(5+9)
6 5 + 5 9 + /
Value : 0
Input infix expression 5%(6-1)^5
5 6 1 - % 5 ^
Value : 0
Input infix expression 5%6-1^5
5 6 % 1 5 ^ -
Value : 4
Input infix expression 8/(2^2)
8 2 2 ^ /
Value : 2
Input infix expression (5/2)^2
5 2 / 2 ^
Value : 4
Input infix expression 5+6^3+(9-1)
5 6 3 ^ 9 1 - + +
Value : 229
Input infix expression 1+(2-(3+(4-(5+(6-7)))))
1 2 3 4 5 6 7 - + - + - +
Value : 0
Input infix expression 1+2-3+4-5+6-7+8-9+0
1 2 + 3 - 4 + 5 - 6 + 7 - 8 + 9 - 0 +
Value : -3
```