Converting Postfix Expressions to Infix

W. Michael Perkins, 17 Jun 2012 CPOL

4.79 (6 votes)

Rate this:

This article provides a general algorithm and a C# implementation for converting expressions written in postfix or reverse Polish notation to infix.

Introduction

Postfix notation, also known as reverse Polish notation, is a syntax for mathematical expressions in which the mathematical operator is always placed after the operands. For instance, the addition of 1 and 2 would be written in postfix notation as "1 2 +". Computer scientists and software developers are interested in postfix notation because postfix expressions can be easily and efficiently evaluated with a simple stack machine. Moreover, postfix notation has been used in some hand-held calculators because it enables arbitrarily complex expressions to be entered and evaluated without the use of parentheses and without requiring the user to keep track of intermediate results. In particular, Hewlett-Packard has produced a number of scientific and engineering calculators that use postfix notation.

Though postfix expressions are easily and efficiently evaluated by computers, they can be difficult for humans to read. Complex expressions using standard parenthesized infix notation are often more readable than the corresponding postfix expressions. Consequently, we would sometimes like to allow end users to work with infix notation and then convert it to postfix notation for computer processing. Sometimes, moreover, expressions are stored or generated in postfix, and we would like to convert them to infix for the purpose of reading and editing.

There are a number of articles on *The Code Project* and elsewhere that discuss methods for evaluating postfix expressions and for converting standard infix expressions to postfix. Discussions of methods for converting postfix to infix are not as common. In this article, I present a general algorithm for converting postfix to infix and provide an implementation in C#.

The General Approach

The most common algorithm for evaluating postfix expressions uses a stack to maintain intermediate results. The expression is scanned left to right. When numbers are encountered, they are pushed on the stack. When an operator is encountered, the operands are popped off the stack and the result is pushed back on the stack. When the scan is complete, the final value of the expression is left on the stack.

We will use a similar stack-based approach for converting postfix expressions to infix. The general algorithm will work the same, but instead of using the stack to store intermediate results, we will use it to store intermediate infix subexpressions. The basic idea is the following:

1. The postfix expression is scanned from left to right.
2. If a token is a number, the token is pushed onto the stack.
3. If a token is an operator, the top two infix subexpressions are popped from the stack and a new infix expression is constructed by combining these subexpressions with the operator in a normal infix manner. The resulting expression is then pushed onto the stack. Initially, we will place each subexpression in parentheses to ensure the proper order of evaluation in the final expression.

Table 1 illustrates the conversion of "1 2 * 3 4 * + 5 *".

Table 1. Converting "1 2 * 3 4 * + 5 *"

Token	Stack (| delimited)	Comment
1	1	Token is a number. It is pushed on the stack.
2	2 | 1	Token is a number. It is pushed on the stack.
*	1*2	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "1*2" is pushed on the stack.
3	3 | 1*2	Token is a number. It is pushed on the stack.
4	4 | 3 | 1*2	Token is a number. It is pushed on the stack.
*	3*4 | 1*2	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "3*4" is pushed on the stack.
+	(1*2)+(3*4)	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "(3*4)+(1*2)" is pushed on the stack.
5	5 | (1*2)+(3*4)	Token is a number. It is pushed on the stack.
+	5*((1*2)+(3*5))	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "5*((1*2)+(3*4))" is pushed on the stack. Since there are no more tokens, this is the final infix expression.

The Problem of Parentheses

Suppose that expression A is "3+2" and expression B is "5". If we construct A + B without adding additional parentheses, we would get "3+2*5". This is surely not what was intended. Since A evaluates to 5 and B evaluates to 5, we expected an expression that evaluates to 10. However, this expression evaluates to 13 because the normal rules of operator precedence require that we evaluate "2*5" first and then add 3.

We addressed this problem in the previous section by always placing parentheses around subexpressions. Taking this approach, A + B would generate "(3+2)+5" which is what we desired. The problem with this solution, however, is that it sometimes generates expressions with a lot of unnecessary parentheses. Indeed, the final expression generated in Table 1 is equivalent to "5*(1*2+3*4)" which has two fewer sets of parentheses. In cases of complex expressions, the extra parentheses can significantly reduce the readability of the expression.

The solution is to add parentheses to subexpressions only when they are needed. Parentheses are only required around a subexpression if the subexpressions main operator has a lower precedence than the operator being used to combine it with another subexpression. Otherwise, the proper order of evaluation is maintained without any additional parentheses.

More formally, suppose that we are constructing a new expression out of subexpressions A and B using operation opNew. Note that A was itself constructed using some operation operA, and B was constructed using some operation operB. At the time we construct the new expression, we can determine whether we need to place parentheses around A or B by comparing the precedence of operA and operB with operNew. If their precedence is less than or greater than operNew, then no parentheses are required. Otherwise, we will place parentheses around the subexpression at the time we construct the new expression.

Suppose operA is +, operB is * and operNew is *. Since operNew has greater precedence than operA, we will need to place parentheses around A. Since operNew has the same precedence as operB, we do not have to place parentheses around B. So our new expression will have the form (A)*B. We can make this more concrete by supposing that A is "1+2" and B is "3*4". Applying the parentheses rule, our new expression would be "(1+2)*3*4" which has parentheses only around those subexpressions that require them to ensure the proper order of evaluation.

In Table 2, we work through the same example as Table 1, but this time, we use the parentheses rule to determine when to place parentheses around subexpressions.

Table 2. Converting "1 2 * 3 4 * + 5 *" Using Parentheses Rule

Token	Stack (| delimited)	Comment
1	1	Token is a number. It is pushed on the stack.
2	2 | 1	Token is a number. It is pushed on the stack.
*	1*2	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "1*2" is pushed on the stack.
3	3 | 1*2	Token is a number. It is pushed on the stack.
4	4 | 3 | 1*2	Token is a number. It is pushed on the stack.
*	3*4 | 1*2	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "3*4" is pushed on the stack.
+	1*2+3*4	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "3*4+1*2" is pushed on the stack. Note that neither of the subexpressions are placed in parentheses because * has a higher precedence than +.
5	5 | 1*2+3*4	Token is a number. It is pushed on the stack.
+	5*(1*2+3*5)	Token is an operator. The top two elements are popped off the stack and combined with the operator in infix manner. The resulting expression "5*(1*2+3*4)" is pushed on the stack. The right-hand subexpression is placed in parentheses because + has a lower precedence than *. Since there are no more tokens, this is the final infix expression.

An Implementation in C#

Listing 1 contains a complete C# implementation of a postfix to infix conversion function. Class Intermediate is used to represent intermediate subexpressions on the stack. Member expr stores the subexpression string, and member oper stores the operator that was used to construct the subexpression. oper is used to determine whether parentheses need to be added when expr is combined into a larger expression. It turns out that the only time the parentheses need to be added is when a new expression is being constructed with * or / and the subexpression was constructed with + or -.

Listing 1.

Hide Shrink Copy Code

```
//
// class used to represent intermediate expressions on the stack.
//
public class Intermediate
{
    public string expr;     // subexpression string
    public string oper;     // the operator used to create this expression

    public Intermediate(string expr, string oper)
    {
        this.expr = expr;
        this.oper = oper;
    }
}

//
// PostfixToInfix
//
```

```
static string PostfixToInfix(string postfix)
{
    // Assumption: the postfix expression to be processed is space-delimited.
    // Split the individual tokens into an array for processing.
    var postfixTokens = postfix.Split(' ');

    // Create stack for holding intermediate infix expressions
    var stack = new Stack<Intermediate>();

    foreach(string token in postfixTokens)
    {
        if (token == "+" || token == "-")
        {
            // Get the left and right operands from the stack.
            // Note that since + and - are lowest precedence operators,
            // we do not have to add any parentheses to the operands.
            var rightIntermediate = stack.Pop();
            var leftIntermediate = stack.Pop();

            // construct the new intermediate expression by combining the left and right
            // expressions using the operator (token).
            var newExpr = leftIntermediate.expr + token + rightIntermediate.expr;

            // Push the new intermediate expression on the stack
            stack.Push(new Intermediate(newExpr, token));
        }
        else if (token == "*" || token == "/")
        {
            string leftExpr, rightExpr;

            // Get the intermediate expressions from the stack.
            // If an intermediate expression was constructed using a lower precedent
            // operator (+ or -), we must place parentheses around it to ensure
            // the proper order of evaluation.

            var rightIntermediate = stack.Pop();
            if (rightIntermediate.oper == "+" || rightIntermediate.oper == "-")
            {
                rightExpr = "(" + rightIntermediate.expr + ")";
            }
            else
            {
                rightExpr = rightIntermediate.expr;
            }

            var leftIntermediate = stack.Pop();
            if (leftIntermediate.oper == "+" || leftIntermediate.oper == "-")
            {
                leftExpr = "(" + leftIntermediate.expr + ")";
            }
            else
            {
                leftExpr = leftIntermediate.expr;
            }

            // construct the new intermediate expression by combining the left and right
            // using the operator (token).
            var newExpr = leftExpr + token + rightExpr;

            // Push the new intermediate expression on the stack
            stack.Push(new Intermediate(newExpr, token));
        }
        else
        {
            // Must be a number. Push it on the stack.
            stack.Push(new Intermediate(token, ""));
        }
    }
```

```
    // The loop above leaves the final expression on the top of the stack.
    return stack.Peek().expr;
}
```

License

Share

About the Author

W. Michael Perkins

Prosper Technologies
United States

Michael Perkins is a software developer, philosopher and musician with special interests in discrete mathematics, the history and philosophy of computer programming, Buddhist philosophy and modern jazz. He is a graduate of Georgia State University where he studied music and philosophy and The Ohio State University where he studied philosophy and computer science. He completed a Ph.D. in Philosophy at The Ohio State University in 1983.

For over 25 years, Michael has developed sophisticated software systems for some of the world's leading software vendors. He has designed and implemented special-purpose programming languages, data management tools, business application generators, cross-platform networking software, and IT systems management software. He has programmed professionally in 370 Assembly, I86 Assembly, Cobol, C, C++, C#, Pascal, VB, Perl, PHP, Java, Javascript, Python and R. He has developed software for IBM mainframes, most flavors of Unix, Vax VMS, AS/400 and Windows.

Currently, Michael is Chief Scientist for Prosper Technologies where he creates software systems for integrating, analyzing and visualizing market research and economic data. Prior to joining Prosper Technologies, Michael held senior technical and management positions at Goal Systems, Computer Associates, Symix Systems, ECNext and Quest Software.

Michael is married to the poet Paula J. Lambert and has five beautiful children.

You may also be interested in...

Converting Postfix Expressions to Infix

How 5 Companies Maintain Optimal .NET Performance

Converting InFix to PostFix using C#

The Basics of Inputs and Outputs, Part 2: Understanding Protocols

Top 5 .NET Metrics, Tips & Tricks

Simple Obstacle Avoidance Robot

Comments and Discussions

You must Sign In to use this message board.

Search Comments Go

First Prev Next

How do i evaluate the infix expression generated at last Nitesh56 6-Aug-14 22:07	1
Incorrect parentheses algorithm? Hynek Syrovtka 11-Jul-13 0:58	1
Thanks you Member 9998057 20-Apr-13 19:29	1
Check out this version of InFix to PostFix Conversion :) xtreme performer 27-Feb-13 3:51	1
My vote of 5 Kanasz Robert 6-Nov-12 2:28	1
Typo cadebabu 27-Sep-12 15:15	1
I would make the interpreter more generic... Andreas Gieriet 19-Jun-12 14:09	2
My vote of 5 Wooters 18-Jun-12 7:08	1
My vote of 5 fredatcodeproject 18-Jun-12 1:38	1

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin