Project files on your SuperDisc
For the full contents of your cover disc, see page 9

Evaluating expressions

We look at how to use numerical evaluation on expressions for use in parsing

What's covered

- Way back in our early schooldays, given an arithmetic expression, we learnt how to read it and calculate the answer. We also learned about precedence rules and how to change them using parentheses. It's all very well using a calculator, but how can we program the evaluation algorithm?

Nugget

In the main article, we state that the individual parsing routines would return a Boolean value to indicate success or failure and also in the first case an RPN expression. Since, in most languages, methods can only have one return value, we would have to use something like an out or a byref parameter. I've always found that type of coding pattern very hard to read, and so generally I create a special class to hold both the success indicator and the resulting value instead and return an instance of that. ■

Early on in my career when programming the IBM PC, I wrote a library in Turbo Pascal for creating Lotus 1-2-3 worksheets. Already I can see you checking Wikipedia to see how long ago this was, given all the hints I've given in one sentence, but let me tell you it was going to make my fortune. Unfortunately, in the end it all fizzled out due to a contract I was working on not paying up. It never got released, thankfully, because in the next couple of years, everyone was using Excel.

One of the interesting parts of this library was the expression parser. It would take an expression in a cell and parse it into Lotus' internal postfix form, and, given some values, would then use that converted expression to calculate its value.

How is this done? The first step is to calculate the reverse Polish form of the expression. Before we see the notation, a bit of history: why reverse and why Polish? It was named after a Polish mathematician called Jan Lukasiewicz who was the first to propose such a form in 1951. He proposed a 'prefix notation' in which the operator appears before its operands.

Maintaining standards

The standard way of writing an expression, say (1+2)*3, is an 'infix notation': the operators are found in between their operands. So, in our example, for the addition operator the two operands are 1 and 2, and for the multiplication operator the two operands are (1+2) and 3.

The biggest problem with infix notation is that we sometimes need parentheses in order to force certain calculations to be performed before others. Since multiplication is assumed to be done before addition (we say multiplication has higher precedence than addition), we need to use parentheses around the 1+2 part to signal to the reader that this has to be calculated first before we do the multiplication. We have to scan the expression first to identify what subcalculations have to be done first and sometimes we have to write down the results of some sub-expressions. Anyone who's used a cheap or older algebraic calculator that doesn't have parentheses keys is well used to this implicit scanning and jotting down process.

```
expression => term (add term)*
term => factor (multiply factor)*
factor => number | '(' expression ')'
add => '+' | '-'
multiply => '*' | '/'
number => single digit
```

▲ **Figure 1: A simple expression grammar**

Lukasiewicz got annoyed at all this and so devised a notation where the operator preceded its operands. (Our example expression would be written as *+123 in this form.) When engineers were first building calculators (both the older mechanical ones and the early electronic ones, especially from Hewlett-Packard), they worked out that they would be easier to build if they abandoned the infix notation and reversed Lukasiewicz's prefix notation to a postfix one, where the operator is found *after* its two operands. So you'd enter the first number into your calculator, enter the second number and then press the operator key (say, +). This is how most adding machines work to this day and Reverse Polish Notation (RPN) was born. Our example expression in RPN form would be 12+3*.

The nice thing about RPN is that there is no concept of operator precedence and no parentheses are needed, but, even better, RPN expressions are easily evaluated using a standard stack. In essence it goes like this: read the RPN expression from left to right. For a number (or a variable, come to that), push it onto the stack. For an operator, pop off two operands from the stack (we assume that all operators are binary and have two operands with the first one popped as the right operand and the second one the left operand), apply the operator to them (that is, evaluate the expression 'leftoperand operator rightoperand', and push the result onto the stack. When you reach the end of the expression, there should be one value on the stack and it's the result of evaluating the expression. (Figure 2 (right) shows this algorithm with 12+3*, where we assume that values are only one digit.)

Expressing ourselves

So how do we get the RPN form of an ordinary expression? We use what's known as a 'recursive descent parser' on the grammar of an expression. The simple grammar we'll use for our expressions is the one shown in Figure 1 (above).

The way you read this grammar is to start at the top and work your way down. Words in

Nugget

Of course, numbers generally are not single digits. What problems would occur if you wanted to use numbers of the form 1.23? Changing the parser is pretty simple (in the code, there's already a method to do that). The bigger issue is the RPN expression; after all, understanding 1.2+3.4 is easier than understanding 1.23.4+. The easiest way is to make the RPN expression be a linked list of tokens (numbers and operators) rather than a simple string. ■

Spotlight on… Extending the grammar

The grammar shown only allows for the standard four arithmetic operators. What if you wanted to have the 'raising to a power' operator (known as exponentiation, usually shown as '^')? What if you wanted calls to functions like sin, cos or log? You have to extend the grammar.

For example, exponentiation has a higher precedence than multiplication; that is 2^3*4 is calculated as 32 rather than 4096. For us to get the desired result we would have to insert another grammar rule which defines another item. So, for example:

```
term => factor (multiply factor)*
factor => powerfactor ('^' powerfactor)*
powerfactor => number | '(' expression ')'
```

For a function, it's a little more complicated, but in essence the factor rule that introduced the parenthesised expression would be extended to include another option, something like:

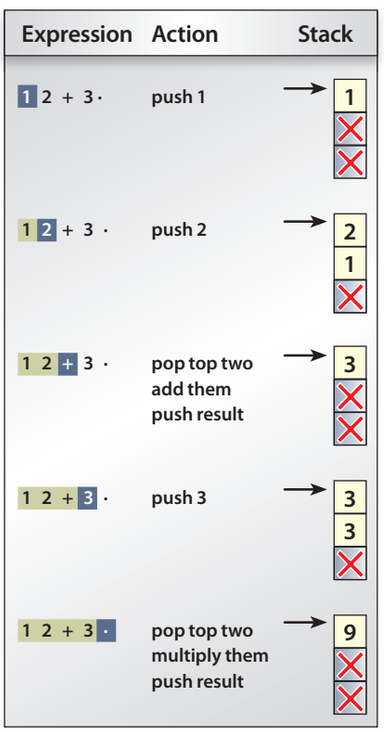
```
function => identifier '(' expression ')'
```

In Lotus 1-2-3, life was much simpler since all function names began with an @ sign. ■

italics are defined within the grammar, literal values are in quotes or have a description. The arrow means 'is defined as'. The pipe character (the vertical bar) means 'or'. When part of a rule is placed in parentheses with an asterisk, it means 'is repeated zero or more times'. So, the first 'production' or rule in the grammar, says that an 'expression' is a 'term', followed by zero or more 'adds' followed by other terms. An 'add' is merely an addition operator, either + or -. A term, on the other hand, is a factor possibly followed by a multiply followed by another factor. And so on and so forth.

This grammar is recursive since a factor can be defined as an expression within literal parentheses.

Let's look at the expression (1+2)*3 and see how the grammar works. Since the expression has no addition operators outside the parentheses, it's counted as a term. We look at the next rule: is it a multiplication? Yes, with the first factor being (1+2) and the second, 3. The second factor is a simple number and we're done with it, but the first can be parsed further as a parenthesised expression. This time we have an addition with the first term being 1 and the second, 2. Both of these can be further parsed into single factors and then as simple numbers.



▲ **Figure 2: Evaluating 12+3* with a stack**

The practical bit

But how do we do this in code? The easiest way is to make each of the rules a routine that returns a Boolean and that parses that rule. If it returns true, the parsing was successful; if false, not. Let's start at the bottom and work our way up. We assume that we have the expression string and we know where we're at in the string. 'ParseNumber', then, reads a single digit from the expression string. If there was no digit, the routine returns false. If there was, we do something with the digit (we haven't discussed what yet) and we move the current position on by one. 'ParseMultiply' reads either a * or a / in the same manner; 'ParseAdd' does the same with + or -. 'ParseFactor' has a choice: first we try and read a number by calling ParseNumber and if that returns false, we try the alternate, that is, try and read a left parenthesis, then parse an expression (by calling 'ParseExpression'), then read a right parenthesis. If, in the alternate, any of these three separate parsing steps fails, we return false.

'ParseTerm' is a weird one. First we call ParseFactor. If that returns true, we call ParseMultiply. If that returns false, it's all right, we just return true, assuming that there is only one factor. Otherwise we call ParseFactor again and expect it to return true. We repeat these two calls to ParseMultiply and ParseFactor in a loop until ParseMultiply returns false. If ParseFactor ever returns false, so does ParseTerm. In a similar manner we can write ParseExpression.

Now, by calling ParseExpression on the original expression string, we run a recursive descent parser. 'Recursive' because the parser can call ParseExpression from within itself; 'descent' because we start at the top, with a complex expression and work our way down through the expression – in essence building the 'expression tree' – until we get a series of simple tokens that we can deal with.

And how do we deal with these leaf tokens? Or, to put it another way, how can we assemble the RPN expression from these tokens as we parse the original string?

Actually there's nothing arcane about it. For the first rule in our grammar, we parse the left term (save it) and then we parse a possible add operator (and save it) and its associated right operand. At this point we can assemble an RPN sub-expression as left operand, right operand, operator. Ditto for the second rule in the grammar. Each parsing routine would then return this RPN expression if it didn't fail.

A full implementation of this recursive descent parser and RPN evaluator, written in C#, is available on this issue's SuperDisc. The evaluate method looks like this:

```
public static double Evaluate(string rpn) {
Stack<double> stack = new Stack<double>();
foreach (char token in rpn) {
if (char.IsDigit(token))
stack.Push(Convert(token));
else
stack.Push(Evaluate(token, stack.Pop(), stack.Pop()));
}
return stack.Pop();
}
}
```

with each evaluate method following this pattern:

```
private static IParseResult ParseAdd(ParserState state) {
char current = state.Current;
switch (current) {
case '+':
case '-':
state.Advance();
return new SuccessfulParse(current.ToString());
default:
return new FailedParse();
}
}
```

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk