Arithmetic with cards

Generating arithmetic equations and solving them: it's what all the cool kids are doing. And your PC can do it too, if you're clever about it...

What follows is a simple card game than will exercise your mental arithmetic skills. However, once you think about it in terms of finding all possible solutions by writing a program, you realize that there is a lot more to it than you think. We'll learn about expression trees, generating all possible binary trees, evaluating expressions, and RPN.

Now, I don't know about you, but I keep a large number of RSS subscriptions in my news reader. The blogs I subscribe to cover computer-related and programming topics as well as political subjects, scientific themes, comics and so on. Recently a couple of posts from widely different blogs caught my eye because, although they were on completely different topics, they nevertheless meshed with each other. Let's start with the game first – I saw this on The Daily WTF.

Take a deck of cards and discard the face cards and any jokers. You'll be left with 40 cards, from ace to 10 in the four different suits. Shuffle them, and then deal out five cards followed by a single card, a little separated from the rest. View the cards as pure numbers from 1 to 10, ignoring the suits. The game, then, is to use the four main arithmetic operators (addition, subtraction, multiplication and division), together with as many parentheses as needed, to create an equation between the five numbers on the left (without rearranging them) and the single result card on the right.

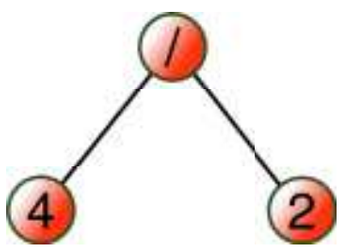
For example, if you've dealt the following sequence of cards:

2♣ 3♥ 9♣ 7♠ A♦ 8♠

An answer you might come up with could be (and there are others):

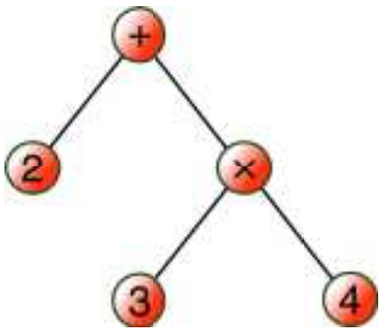
((2 * 3) + (9 - 7)) * 1 = 8

If you've ever watched Channel 4 in the afternoons during the week, the programme *Countdown* has a similar puzzle. *Countdown* uses six numbers drawn from a field of four 'large' numbers and 20 'small' numbers, and the contestants do not need to use all six to obtain the random value as the result, nor do they have to use them in the order drawn. In our game here, you have to use them all and in the given order.



▲ **Figure 1: The binary expression tree for the simple calculation 4 /2.**

Now, I'd say it's hard enough to do this in your head, but writing a program to do it? Where would you start? Somehow you'd have ▶



▲ **Figure 2: The expression tree for 2 + 3 * 4, using normal operator precedence.**

► to generate every single possible arithmetic expression using the numbers on the left-hand side. And then how would you evaluate each expression to make sure that the result equalled the value on the right? There would seem to be some nice chunky programming involved.

The possibilities

First task then: generate every possible expression. It seems difficult. There is the precedence of the arithmetic operators to take into account (so that 2 + 3 * 4 is equal to 14 and not 20), so there's some need for parentheses to group sub-expressions.

In order to attack this problem, the first thing to realise is that the four operators are all binary operators: they take two operands, usually known as the left-hand side (lhs) and the right-hand side (rhs). We could visualise this as a tiny binary tree as in Figure 1, which shows 4 / 2. The lhs is shown as the left child of the operator node and the rhs as the right child, as you'd expect. With another operator, we can expand this as a bigger binary tree: Figure 2 shows 2 + 3 * 4 (using the standard precedence rules) and Figure 3 shows (2 + 3) * 4. Notice that there are two types of nodes: the operator nodes that always have two children, and the number nodes that have none.

This visual representation of an expression is great, because evaluating it is simple and recursive. We start at the root and say: 'Evaluate the lhs, evaluate the rhs, and then combine them according to the operator and return the result. To evaluate the lhs (and the rhs), we say: 'If it's a number, the result is the number itself; otherwise proceed as before (evaluate the lhs, the rhs, combine), and return the result.' Check this out yourself with Figures 2 and 3.

So once we have generated a binary tree for an expression, evaluating it is simple indeed. The next problem to solve is how to generate all binary trees that contain four internal nodes. (Think about it: the numbers in our binary tree appear as leaf nodes (nodes without children), and there is a well-known result that a binary tree of n internal nodes, where every internal node has exactly two children, has n+1 leaves. Such a binary tree is called a full binary tree.)

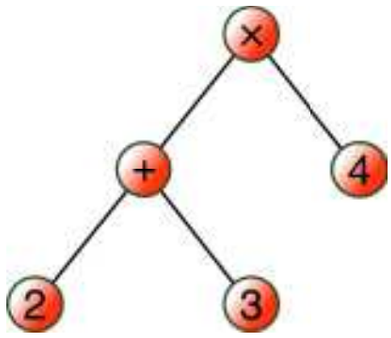
Enter the second blog post, Every Binary Tree There Is, which is from Eric Lippert's blog.

Why 'reverse' in 'RPN'?

The reason for this rather weird term is to do with a Polish mathematician called Jan Łukasiewicz. In the 1920s, he was investigating propositional logic, when he came across the idea of placing the binary logic operators he was using (which he called functors) before their operands. Using this notation meant he could avoid having to use parentheses, further meaning that expressions were easier to read and evaluate. In the 1950s, computer scientists worked out that reversing this notation, so that the operator appeared after its operands, meant that only a simple stack would be necessary to evaluate an expression. Hence, Reverse Polish Notation. Since that time, RPN has even appeared on calculators, mostly from the Hewlett-Packard company. ■

Again – as is the case with many binary tree algorithms – the solution is recursive. From inspection you can work out that there is only one possible tree with a single internal node, and two possible trees with two internal nodes. To extend that to generating all the trees with n internal nodes where n is greater than 2, we reason as follows. Take the example of n = 3. We know that one of the internal nodes will be the root. The root's left child will have either two nodes (and the right, none), one node (and the right one also) or no nodes (and the right will have two). Since we know how to generate all trees with two, one or no nodes, we can therefore easily generate all the trees with three nodes. For n = 4, the reasoning is similar: the left will have three, two, one, or no nodes (and the right will have no, one, two or three nodes, respectively).

The number of trees generated in this manner for differing numbers of internal nodes form the 'Catalan' number sequence, named after the Belgian mathematician Eugène Catalan. The Catalan numbers are 1, 2, 5, 14, 42, 132, 429, and so on, and grow rapidly. The tenth Catalan number is 16,796, which we can interpret as the number of unique binary trees with 10 internal nodes.

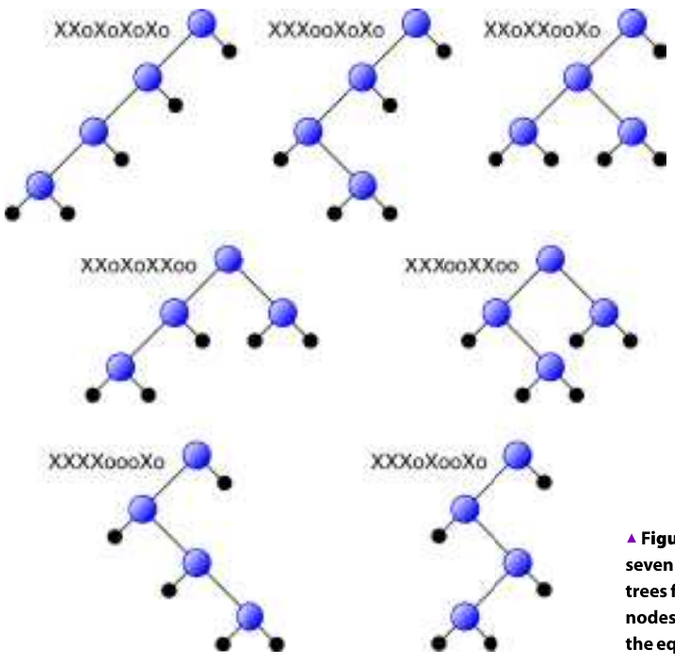


▲ **Figure 3 : The binary expression tree for the calculation (2 + 3) * 4.**

For our purposes with four internal nodes, the number of unique variants of binary trees is still relatively small: just 14. We could generate these at runtime using this recursive definition, but for such a small number it's possibly worth just predefining them in some kind of data structure (Figure 4 shows the first seven, the other seven are mirror reflections in the vertical axis through the root. Underneath each is the equivalent RPN expression – see the 'Reverse Polish Notation' box on page 83.)

At this point, let's step back from everything a moment to see where we're at. We've seen that arithmetic expressions can easily be represented by full binary trees and that evaluating such a binary tree is simple and recursive. We've also seen how to generate all the binary trees for a certain number of internal nodes (that is, operators).

The next step is to plug each of the four operators into each of the four internal nodes



▲ **Figure 4 : The first seven unique binary trees for four internal nodes, each showing the equivalent RPN.**

Other game variants

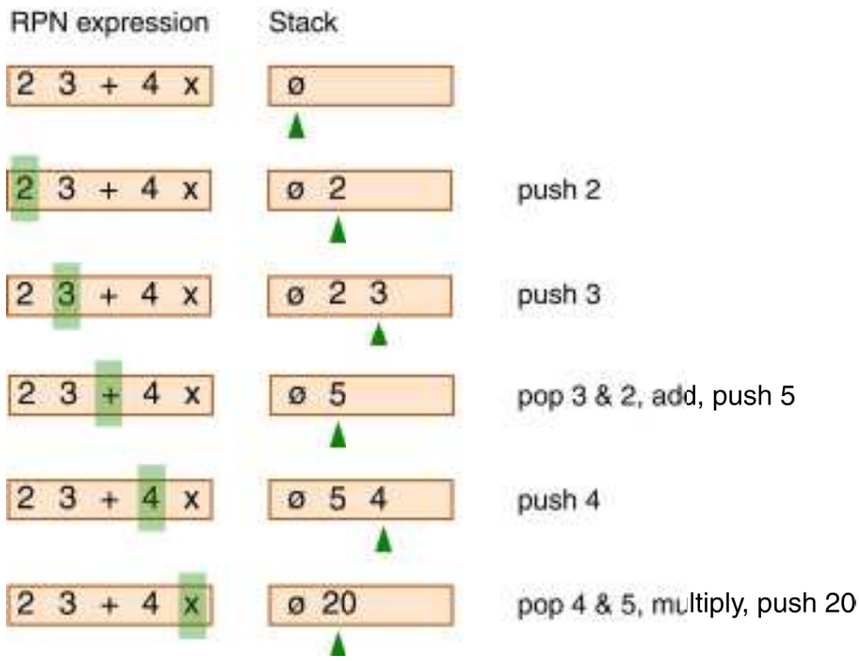
The *Countdown* variant of this game is more complicated, since you can use the numbers in any order and even omit some if you don't need them. However, the approach outlined in this article is still valid, with the proviso that you have to permute the numbers instead of accept them in the original order. Generating permutations is another interesting algorithm, one which we have no space to elaborate on here.

Another variant I've investigated is a school problem presented to my brother-in-law as a mental arithmetic exercise some 10 years ago. Given the first four prime numbers (2, 3, 5, 7), insert operators and parentheses where necessary in order to derive expressions that evaluate to the values 1 to 100. ■

for each of the 14 possible binary trees (which makes 3,584 possible expressions) and evaluate each expression. (Remember that the numbers themselves remain in the order dealt, so we don't have to permute them during the process.) We make a note of each expression whose evaluated result is equal to the sixth value: there could be none, one or several, of course.

Showing your results

Now that we've identified one or more expressions that satisfy the rules, how do we display them? After all, we don't usually view our arithmetic expressions as binary trees. The simplest answer is again recursive, but unfortunately it's not the best. We start off at the root node. Output a left parenthesis, evaluate the left child, output the operator for the root node, evaluate the right child, and finish off with a right parenthesis. To evaluate a child, we look to see if it is a number or not. If so, we output the number. If not, we perform the left parenthesis, left child, operator, right child, right parenthesis thing again.



▲ **Figure 5: Evaluating the RPN expression 2 3 + 4 * using a stack.**

I say this solution is not the best because although it produces a valid expression that evaluates correctly without any problems due to operator precedence, it also produces one that has an overabundance of parentheses, most of which are not needed. As an example, Figure 2 would produce this expression: (2 + (3 * 4)) and as we've seen using the standard precedence rules, we can uniquely specify it without any parentheses at all.

So, let's refine the recursive step a little. What we will do is to define a 'precedence number' with each operator. For addition and subtraction we'll use 1, and for multiplication and division we'll use 2. We'll then use the rule that if we go from an operator with a higher precedence to one with a lower precedence, we'll output parentheses to surround the subexpression containing the lower precedence operator. If the precedence number is equal or lower, we don't output any parentheses.

> **"The simplest answer is recursive, but unfortunately it's not the best solution"**

The recursive step now becomes this: If the node is a number, output the number. Otherwise, the node is an operator. If the previous operator has a higher precedence than ours, output a left parenthesis, the result of the recursion on the left child (passing the precedence number of our operator), our operator, the result of the recursion on the right child (passing our precedence number), and then a right parenthesis. If the previous operator has a lower or equal precedence to ours, do the same except without outputting any parentheses.

We kick the whole thing off by calling the recursive step on the root of the expression tree with a precedence number of 0, forcing the overall expression to have no enclosing parentheses. Using this recursion on Figure 2, we'll output 2 + and then call the recursive step on the right child. Since the precedence for addition is 1, which is lower than the precedence for multiplication (2), the right child will not output any parentheses. It produces 3 * 4, making the whole output 2 + 3 * 4.

For Figure 3, we'll be asked to evaluate the left child, passing the precedence for multiply, 2. Since this is greater than the precedence for addition, 1, the left child will output (2 + 3). The final expression would complete to (2 + 3) * 4. At this point, we have in place enough information and algorithms in order to fully solve the original card game. PCP

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

Spotlight on… Reverse Polish Notation

In the main text I rather blithely stated that for the game, it would be possible to store the 14 different binary trees in some kind of data structure. But how do we do that without writing explicit recursive routines? The answer is to use RPN (Reverse Polish Notation).

Look back to how to evaluate the binary tree: we evaluate the lhs, evaluate the rhs, and then combine them using the operator. This is RPN in a nutshell. In practice we use a stack: if the next token is a number, push it onto the stack; if it's an operator, pop the rhs, pop the lhs, combine them and push the result. The final result is the number on top of the stack. 2 3 + 4 * would work out as 20 (and is the RPN representation of Figure 3), whereas 2 3 4 * + would evaluate to 14 (that is, Figure 2). Figure 5 demonstrates how you can evaluate the first RPN expression we gave (2 3 + 4 *) using a stack in this way.

If I represent a number with X and an operator with an 'o', we can specify the 14 RPN strings as follows for the four-operator game: XXXXXoooo, XXXXoXooo, XXXoXXooo, XXXooXoo, XXXoXoXoo, XXoXXXooo, XXoXXoXoo, XXXooXXoo, XXoXoXXoo, XXXXoooXo, XXXoXooXo, XXoXXooXo, XXXooXoXo, XXoXoXoXoo.

Using these RPN templates, we can easily substitute the actual cards for the Xs in order, and then construct the actual expressions by replacing each o with each operator in turn. ■