Reverse Polish Notation

Ian McLoughlin (ianmcloughlin.github.io)

Reverse Polish notation is a syntax for writing expressions involving operators and operands. When the number of operands that each operator takes is fixed, reverse Polish notation does not require any brackets, or precedence of operators, to unambiguously represent an expression.

Example

Let's start with an example. Consider the following bracketed expression.

$$((5 + 4) \times 9) \div (6 - 3)$$

This expression is written in what we call infix notation – the (binary) operators are written in-between their (two) operands. The operators in this case are the common addition, subtraction, multiplication and division. The operands are the numbers to which these operators are to be applied: 5, 4, 9, 6 and 3. In reverse Polish notation (RPN), this expression is written as:

$$5 \quad 4 \quad + \quad 9 \quad \times \quad 6 \quad 3 \quad - \quad \div$$

The difference between the RPN form and the infix form is simply that the operator is written after its two operands, rather than between them.

Benefits of RPN

We are so used to using the infix notation, why would we use RPN instead? There are three main benefits.

Firstly, RPN does not require bracketing. In infix notation, brackets are needed in expressions such as $(3 + 4) \times 5$. Without them, 4 would be multiplied 5 giving 20, which in turn would be added to 3. This is following the usual convention of giving multiplication a higher precedence than addition. That would give the result as 23, rather than the correct answer of 35. In fact, RPN does not require any hierarchy of operators at all.

Secondly, RPN expressions can be evaluated using a simple algorithm. The algorithm involves a single pass of reading the expression from left to right. The intermediate calculations can be done as they arise. Contrast this with expressions in infix notation such as $3 + 4 \times 5$. In this case the expression must be fully read from left to right before any intermediate calculation can be performed and the expression can be interpreted.

Finally, when an RPN expression is to be evaluated by a computer, less memory and less instructions are used as compared with infix expressions. Less memory is needed, because the computer can apply the operators when they are read, as noted above. With infix expressions, the computer must read and store the expression before parsing it. Less instructions are needed with RPN for the same reason.

Evaluation and the stack

Let's now evaluate the above RPN expression:

$$5 \quad 4 \quad + \quad 9 \quad \times \quad 6 \quad 3 \quad - \quad \div$$

Evaluation can be done using a single stack. A stack is a simple data structure that permits only two operations: pushing and popping.

Pushing involves placing an item on the top of the stack. This is the only way to put a new item into the stack, to place it on top. Popping involves removing the item on the top of the stack. This is the only way to remove an item from the stack, to take the top item. In this way, a stack is a last-in-first-out mechanism – the last item in must be the first item out.

In evaluating an RPN expression, a stack is used to store numbers. These numbers can be either the original operands in the expression, or the result of applying some of the operators in the expression to some of the operands.

An RPN expression is read item by item from left to right. Two simple rules are used as each item is read. The first is applied when the item is a number: push the number to the top of the stack. The second is applied when the item is an operator: pop the first element off the stack, pop the next element off the stack, apply the operator to them, and finally push the result to the top of the stack. So, the rules are:

Number: push number to stack.

Operator: pop twice from stack, apply operator, push result.

The following table depicts the stack as each symbol of our example RPN expression is read.

	5	4	$+$	9	$\times$	6	3	$-$	$\div$
							3		
Stack		4		9		6	6	3	
	5	5	9	9	81	81	81	81	27

So, when the first item, 5 is read, it is a number so it's pushed to the empty stack. The second item, 4, is a number too so it is pushed to the stack. The third is the operator $+$, so the top item on the stack (4) is popped, then the next item (5) is available for popping and is popped. The numbers are added to give 9, and then the 9 is pushed to the stack. The final answer is what is left on the stack once the full expression has been read, and this must be a single number.

Order of the operands

One technicality to watch out for is the order of the operands. With addition and multiplication, the order does not matter. For instance, $4 + 5 = 5 + 4 = 9$. However, with subtraction and division order does matter: $5 - 4 \neq 4 - 5$.

In RPN, using the everyday operators like plus and minus, the convention is for $5\ 4\ -$ to mean $5 - 4$, and not $4 - 5$. It doesn't really matter, so long as we agree on what we mean and are consistent. Consider the following idea: creating a new operator $\ominus$ that is defined as $5 \ominus 4 = 4 - 5$. Then we can use $\ominus$ in our RPN expressions. So, $5\ 4\ \ominus$ in RPN would evaluate as $5 \ominus 4 = 4 - 5$ in infix.

Valid RPN expressions

An interesting question to ask is *what combinations of operators and operands constitute valid RPN expressions?* An expression like $+\ -\ 5$ is clearly not valid. Neither is an expression like $+\ 5\ 4$.

Note that valid RPN expressions always have one more number than operators. Every operator needs two operands, and so two operators need three operands, three need four, and so on. Also, note that the stack algorithm needs to work. It only works if the following two conditions hold:

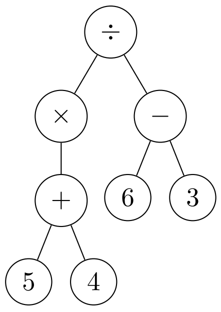
1. There are at least two numbers on the stack whenever an operator is read.
2. There is exactly one number on the stack when the end of the expression is reached.

The first rule implies that every RPN expression must begin with two numbers. The second rule implies that every RPN expression must end with an operator.

While necessary, these last two stipulations are not sufficient. For example, the expression $1\ 2\ 3\ +\ +\ +\ 4\ 5\ +$ both starts with two numbers and ends with an operator. It even has the correct number of operators relative to numbers. However, upon reading the third $+$ symbol, the stack has only a single item on it and the operator cannot be applied. We must read the expression and keep track of the size of the stack to confirm that the stack contains two numbers for every operator read. Only then we will have a valid RPN expression.

Evaluation trees

RPN expressions can be decomposed into evaluation trees. An evaluation tree for an RPN expression has operators for internal vertices, where their operands are their children, and the numbers in the expression are the leaves. The first operand for an operator is drawn as the left-most child, and last is the right-most. So, for instance, the evaluation tree for the above expression is as follows.



References

Jan Lukasiewicz. *Aristotle's Syllogistic From the Standpoint of Modern Formal Logic*. Garland, 1951.