Reverse Polish Notation:
A Gentle Introduction to Stack Machines

Patrick M. Elsen <pelsen@xfbs.net>

v0.1.0

Contents

Introduction

Stack Machines are a good way to get acquainted with computing concepts because they are very accessible and typically simpler to work with than register machines. Reverse Polish Notation is not only historically interesting due to its use in a lot of early calculators, but also in practise due to the simplicity with which they can be implemented.

This article gives the reader an introduction to both by providing examples and easy challenges that can be completed. It assumes some background in computer science, knowledge of using command-line utilities and some programming knowledge in C++.

Following the example of poc||gtfo, this PDF is also a valid ZIP file, containing code discussed here and the source code to this very article. Use the `unzip` command to extract this information.

```
$ unzip rpn.pdf
```

Find the latest version of this at github.com/xfbs/rpn and feel free to redistribute as you like.

1 Overview

Reverse Polish Notation is a way to write down equations without needing to use brackets or precedence rules. This greatly simplifies parsing. For this reason, reverse polish notation has been quite popular with Casio calculators.

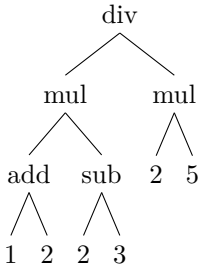
Take a simple mathematical equation with the four primitive operators — addition, subtraction, multiplication and division.

$$\frac{(1+2)*(7-3)}{2*6} \tag{1}$$

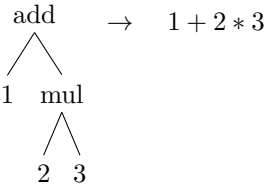
The rules dictate the order in which the operations should be executed, and brackets can be used to override those rules where necessary. For example, in the expression $1+2*7-3$, the multiplication of 2 and 7 would be evaluated first, and only then would the additions and multiplications be evaluated, because multiplication has a higher precedence.

This precedence can be overridden by using parentheses, to force addition to be evaluated before multiplicatiom. The PEDMAS mnemonic is a good way to remember the order of precedence: *parentheses, exponentiation, division, multiplication, addition, subtraction*.

In our mind, we can parse this equation into a tree that represents the order of operations.



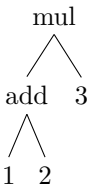
If we think about trees, then the way we write down equations normally is called the *inorder* notation, meaning that we write down the nodes in the order in which they appear in the tree, from left to right.



$\rightarrow \quad 1 + 2 * 3$

This is somewhat intuitive for humans to parse because there is higher locality. But this format also has some issues with it, in that it needs to be parsed and precedence to be applied before it can be executed. Imagine parsing an equation like $1+2*3$. You have to parse the *entire* equation first, because there might always be a higher-precedented operation after.

When using reverse polish notation, you simply rearrange the equations to something called the *postorder*. This means that the child nodes are put first, and then the parent nodes.



$\rightarrow \quad 1 \quad 2 \quad + \quad 3 \quad *$

The advantage of this is immediately obvious: it becomes possible to execute this without parsing the entire thing. The way this is executed is by using a stack.

When executing this, imagine going through a stream of the tokens $[1, 2, +, 3, *]$. Execution starts at the first token, a 1. This is data, and thus it is placed on the stack. Then the next token is encountered, a 2. This, again, is data and thus placed on the stack. Thus far, the stack contain $[1, 2]$ on it. Next, + is encountered, which is an operation that takes two operands. Thus, the last two items are popped from the stack, added, and the result pushed back on it. Thus, at this point, the stack contains only [3]. Visually, this means that we have simplified the expression tree as such:

Next, we encounter the token 3, which again, is data and is pushed on the stack. The stack now contains $[3, 3]$, where the first three is the result of the addition, and the second three is data. Lastly, a $*$ is encountered, which is a multiplication operation, which takes two items off the stack, multiplies them, and pushes the result, a 9, back onto the stack.

Table 1: Executing a simple RPN

Token	Stack	Comment
1	1	Push 1 onto the stack.
2	1, 2	Push 2 onto the stack.
+	3	Adds last two items on stack.
3	3, 3	Push 3 onto the stack.
*	9	Multiplies last two items on stack.

Table 1 illustrates this process in a more organised way. With this knowledge, Equation 1 can be transformed into reverse polish notation rather easily:

$$1 \quad 2 \quad + \quad 7 \quad 3 \quad - \quad * \quad 2 \quad 6 \quad * \quad / \qquad (2)$$

In a sense, reverse polish notation forms a sort of bridge between the world of mathematics and the world of computer science, because reverse polish notation is both a way to represent equations and a kind of instruction stream that can be processed by a virtual stack machine to evaluate it.

2 Evaluating

As we have seen previously, reverse polish notation is very simple to evaluate. We can examine this by writing a small interpreter for it, that only supports some very basic mathematical operations.

Taking an input stream and splitting it into tokens (separated by spaces) is very simple in C++. Parsing these tokens, as `std::string tok`, can be done with a function as simple as this:

```
if(tok == "+") {
    return new Add();
```

```
} else if(tok == "*") {
    return new Mul();
} else if(tok == "-") {
    return new Sub();
} else if(tok == "/") {
    return new Div();
}

return new Num(std::stod(tok));
```

Every token is either an operator or a number. For the uninitiated, `std::stod()` is a C++ function to parse a (decimal) number from a string, and `Add()`, `Mul()`, `Sub()`, `Div()` all subclasses of an abstract operation class.

To represent the stack, the C++ `std::deque<double>` is a good candidate, because it has efficient pop and push operations, but it also allows index-based access to elements, which is useful for displaying the stack.

With this in mind, the execute functions for the number token can look somewhat like this, simply pushing the number that it was constructed with onto the stack.

```
void Num::execute(std::deque<double> &s) {
    s.push_back(number);
}
```

Similarly, the other operations can be implemented, by popping values off the stack and pushing the result back onto it. Here is the addition, implemented by taking the last two numbers off the stack, and pushing the result of the addition back onto it.

```
void Add::execute(std::deque<double> &s) {
    double rhs = s.back();
    s.pop_back();
    double lhs = s.back();
    s.pop_back();
    s.push_back(lhs + rhs);
}
```

There is working example code in the `rpn-calc` folder in this repository with working code that acts as a little reverse polish notation parser.

After every sequence of tokens, it prints out the entire stack, making examining the state very easy.

```
$ make rpn-calc
$ ./rpn-calc/build/calc-cli
> 1 2 +
3
> 3 4 +
3 7
> +
10
> 5 6 +
10 11
> 7 8 +
10 11 15
> +
10 26
> +
36
```

This particular interpreter only supports the four basic math operators, but it is very simple to extend the code to make it support more.

In fact, there are existing reverse polish notation interpreters and compilers that support a lot more. There's even one on your printer right now – we'll get to that one in a bit.

Exercises

1. Compile the `rpn-calc` tool from the code in this repository. If you don't have the code, you can extract it from this PDF file as documented in the introduction. Run the `calc-cli` tool and try out some expressions like `1 2 +`.

2. Can you translate $1 * 3^{6-1} + \frac{2}{4}$ into reverse polish notation?

3. Can you add `^` as exponentiation operator to the interpreter? Hint: create a class named `Exp()`, add an `Exp::execute(...)` method and add it to the parser.

3 Implementations

While it's always good fun and educational to implement reverse polish notation interpreters ourselves, it is perhaps more interesting to examine existing implementations that are already sitting on your system. We can take a look at three particular implementations — GNU DC, Adobe PostScript and Forth. If none of these ring any bells, don't worry.

3.1 DC

There are some utilities sitting on almost every UNIX-like system[1] that rarely anyone has ever had the pleasure to interact with directly, and yet they have great power.

Out of those obscure utilities, we will take a look at one of them: DC. It might just be the oldest and yet most universal reverse polish notation interpreter — but I don't have references on that. There is only one place where you might have stumbled across it, and that is StackOverflow Code Golfing. It is definitely an esoteric language[2], which, if you are into that kinda stuff, makes it even more interesting.

Its name is an acronym for *desktop calculator*, and that is exactly what it does. You can and should pop open the manual page for `dc` to get a bit of information about it, as well as a command reference.

```
$ man dc
```

I won't really be explaining all of what it can do, but we can take a look at some

It looks really quite cryptic, especially because it is character-based. It can do simple addition like we did before. One difference is that it does not implicitly print the stack – but the `p` command exists to print the stack.

```
$ dc -e "1 2 3 + + p"
6
```

It can do quite a lot more than just adding numbers and printing the result. At this point, we have to leave mathematics a bit behind on this journey, because DC supports some more traditional programming constructs that one would hardly find directly in equations, such as dictionaries and lists.

are leaving mathematics behind a bit and entering into computer science, because

Exercises

1. Write a DC script that will find the sum of numbers from 1 to 100.

2. Write a DC script that will find the first 100 prime numbers.

[1] which is most systems these days, taking macOS, iOS and Android into account

[2] https://esolangs.org/wiki/Dc

3.2 PostScript

PostScript is another one of those things that most people haven't ever heard of or used. But unlike DC, this is something you may very well have used.

Back in the 80ies when Adobe was one of the pioneers of digital printing, they needed some way to send data to printers in an efficient manner.

When you want to print a page, you can render it into a large image and send that off. This is not a particularily great idea, however. If we look at A4, which is the standard paper size you will find in printers, they have a size of 200mm × 283. Printing is often done at a resolution of 300 or 600 DPI. At this resolution, a whole document would take about 4 GB to be represented.

The idea with PostScript is that instead of rendering a document as an image locally and sending the (potentially large) image to the printer, we can generate a PostScript program that will generate it, and this can be run on the printer itself.

Such a program can be considerably smaller, especially for documents that consist of text and vector drawings.

PostScript is implemented as a stack machine. But unlike our small RPN calculator, it is a fairly powerful programming language, supporting functions, variables, arrays and dictionaries.

Exercises

1. Install GhostScript on your local system (if you are on macOS, it should come preinstalled, otherwise check your package manager).

   Run some of the PostScript files present in the `postscript` directory.

2. Can you write a function that will compute the largest common multiple of two numbers?

3.3 Forth

After having taken a look at DC and PostScript, which are both interpreted, higher-level languages, we can now look into a slightly different direction and examine another oddity out of the programming languages world, Forth.

4 Generating

Now that we know what reverse polish notation is (the *postorder* of a tree) and how we can interpret it (by working through a stream of tokens), the last interesting aspect that is left is to find out how we can generate reverse polish notation from regular infix notation.

We have already discussed the parse tree, and this is exactly how we translate from one format into another.

In the `rpn-gen` folder you will find a parser, written in C++, that will emit reverse polish notation (in a format that `rpn-calc` understands).

Exercises

1. Can you implement parsing of exponentiation, such that `1 ^ 2` generates `1 2 ^`?

2. Why does the parser have to parse the entire tree before it can generate RPN?

5 Conclusion

Reverse Polish Notation is an interesting format because it is very accessible — having no complex syntax, parsing or precedence rules. This is what makes it interesting for hackers writing their own little custom interpreters or for people new in the field.

We have also seen that it is not

The disadvantage of RPN is that the code is not as readable.