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, $1 + 2 * 2 - 3$ is equivalent to $1 + (2 * 2) - 3$, because multiplication has higher precedence than addition. The PEDMAS mnemonic is a good way to remember the order of precedence: *parentheses, exponentiation, division, multiplication, addition, subtraction.*

This equation could be represented in reverse polish notation as well.

$$1 \quad 2 \quad + \quad 7 \quad 3 \quad - \quad * \quad 2 \quad 6 \quad * \quad / \tag{2}$$

To the uninitiated, this might seem strange, because the properties of this are not yet clear. When evaluating this, it goes strictly from left to right. Numbers are treated as data and placed on an imagined stack. Operators like + (addition) consume data (numbers) from the stack, and place their results back on it. Thus, evaluating a sequence like 1 2 + would go like such.

The last result that is left on the stack is the result of the calculation, in this case 1.

Operation	Stack	Comment
1	1	Place 1 on stack.
2	1, 2	Place 2 on stack.
+	3	Add last two numbers on stack.
7	3, 7	
3	3, 7, 3	
-	3, 4	
	12	
2	12, 2	
5	12, 2, 6	
	12, 12	
/	1	

Another way to think about reverse polish notation is that equations are basically tree structures, and reverse polish notation is just another way to represent the tree.

```
              div
             /   \
          mul     mul
         /  \     / \
      add   sub  2   5
      /\    /\
     1  2  2  3
```
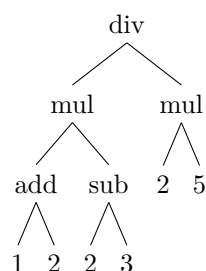
2 Evaluating

With the knowledge of how reverse polish notation looks like, it's fairly trivial to parse and interpret it. Given a function that splits a string into individual tokens, and some classes to represent the operations, parsing RPN is as simple as a few if statements.

```cpp
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));
```

To represent the stack, the C++ `std::deque<double>` is a good candidate, because it acts like a stack (meaning that 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 data onto the stack.

```cpp
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.

```cpp
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 example code in the `calc` folder in this repository with working code that acts as a little reverse polish notation parser. Compiling that and playing with it might help.

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

```
$ make calc
$ .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

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

3   Implementations

3.1   DC

DC might just be the oldest and yet most universal reverse polish notation interpreter. Its name is an acronym for *desktop calculator*, and that is exactly what it does.

You can pop open the manual page for `dc` to get a bit of information about it.

```
$ man dc
```

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.

3.2   PostScript

PostScript is one of those things that people normally don't ever run into, but it's also quite fascinating. 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 particularly 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.

4   Generating

Reverse polish notation is
Reverse polish notation in dc