CSC161 2011S *Imperative Problem Solving*

# Assignment 6: An RPN Calculator

Assigned: Friday, 4 March 2011
Due: 11:00 p.m., Wednesday, 9 March 2011

**Summary:** In this assignment, you will extend K&R's RPN calculator.

**Purposes:** To help you understand the calculator in more depth. To give you experience with separate compilation. To help you think more about error checking and recovery.

**Expected Time:** Three to four hours.

**Collaboration:** I encourage you to work in groups of two or three students. However, you may work on your own or in groups of up to size four. You may discuss the assignment with anyone you wish, provided you clearly document such discussions.

**Submitting:** Submit tarballs of your solutions on Pioneerweb. Further details at the end of the assignment.

**Warning:** So that this assignment is a learning experience for everyone, I may spend class time publicly critiquing your work.

# Preparation

I've built a variant of the K&R Reverse-Polish-Notation calculator that will be the starting point of your assingment. To prepare for this assignment, you should make a copy of that variant.

- Create a directory for this assignment. I'd recommend `csc161/hw6`, but you can choose whatever you'd like.
- In that directory, save a copy of the code for this assignment.
- Unpack the tarball with `tar xvf filename`.
- Look at the code so that you understand the parts.
- Type `make rpn` to create the executable.
- Test the executable to make sure that it works as expected.

Here's a sample session with the calculator. Note that `p` means "print".

```
$ ./rpn
1 7 / p
        0.14285714
7 * p
        1
.32 + p
        1.32
.1 * p .1 * p .1 * p
        0.132
        0.0132
        0.00132
```

```
42*p
        0.05544
1 2 3 4 + + + p
        10
```

# Assignment

As you may have noted, this calculator is relatively simplistic. There are therefore a number of extensions that we should make.

1. The calculator does not accept negative numbers as input. Revise it so that it can accept negative numbers. (You will have to change `get_token`.)

2. As you no doubt have noticed, we tend to use the exponentiation operator a lot. Add one to the calculator. Use `^` as the exponentiation sign. For example,

```
$ ./rpn
2 3 ^
        8
3 2 ^
        9
*
        72
```

3. While this calculator supports subtraction and negative numbers, it's also use to support negation. Use `n` to represent the negation operation. For example,

```
$ ./rpn
3 4 + n p
        -7
```

4. The program leaves leftover results on the stack. At times, we want to be able to reset the stack. Add a `c` operation to clear the stack. You will probably want to change not only the main body of rpn, but also add a stack function for clearing the stack.

5. It is possible to put too much on the stack (overflow) or to read too much from the stack (underflow). For example, although we're supposed to have two operands on the stack in order to use `+` or `-`, we can easily call `+` without having entered any numers. Similarly, although the stack can officially hold only eight numbers, nothing stops you from adding more. (It may be worth seeing what happens in either of these cases.)

To fix this problem, you will need to:

- Add routines (`ds_is_full` and `ds_is_empty`) to `dstack` so that the client can check whether or not it is safe to push or pop a value.
- Update your main so that it checks the appropriate precondition before pushing and popping, issuing an error message to the user if the desired operation is not possible.

6. While we make it the client's responsibility to check preconditions, as Kernighan and Ritchie note, it is also useful for a procedure to also check error conditions and send a special "error value" to the client if the function cannot complete properly.

- Update `dpush` so that it returns an integer indicating success (1) or failure (0).
- Update `dpop` so that it returns a special value (`1.0/0.0`) if there is nothing to pop.
- Update your client code so that it checks the return value and gives an error message if the operation failed.

7. Most calculators these days have memory in which you can store values and later retrieve them. We'll use two-character strings to represent these actions. The first character will be either `s` (for save) or `l` (for load). The second character, which will a lowercase letter, will represent the name of the memory location in which you are storing the character. For example,

```
$ ./rpn
3 4 + sa
p
        7
5 p
        5
la p
        7
+ + p
        19
la +
        26
```

8. Note that we've started to create a simple programming language. (It is not as powerful as C, but it can do a lot of computation.) One thing that we find useful in programming languages is comments. Let's mark comments with a pound sign (#). Update your program so that anything between a pound sign and the end of the line is ignored.

```
$ ./rpn
3
happy
error: unknown command h
error: unknown command a
        3
        3
error: unknown command y
# happy
4 +
7
```

# Submitting Your Homework

Using `script`, build a log of sample runs of your program. Your script should be stored in the file `typescript`.

Edit `README` to insert any information you'd like to provide to me or the grader.

Edit the Makefile to substitute your user names for username in the line that reads

```
USERNAMES=username-username-username
```

Type `make tarball`. This should create a tar file.

Submit the tarball on Pioneerweb under the Assignment 6 link.

---