ECE 264: Advanced C Programming

Programming Assignment 8 - Stacks, Due 7/27/19

In this assignment you will build a calculator program that reads in operations from a file and outputs the result of the computation. This assignment will use a stack to implement its calculator.

1 Learning Goals

In this assignment, you will learn

1. Reverse Polish Notation (RPN)
2. Stacks
3. Using a stack to evaluate RPN expressions

2 Background

2.1 Reverse Polish Notation

In grade school, you learned about *order of operations*, the idea that different operations have higher or lower priority. That means that in an arithmetic expression like `2 + 3 * 7`, the multiplication has higher priority than the addition, so you perform it first, even though the addition shows up earlier in the expression (You may have learned the mnemonic 'PEMDAS" — Parentheses, Exponentiation, Multiplication, Division, Addition, Subtraction). The result of the expression is `23`, not `35`.

Reverse Polish Notation, or RPN for short, is an alternate notation for writing expressions that avoids worrying about order of operations, because the notation makes explicit the order that operations must be evaluated.

In RPN, rather than operators (like addition) being between their operands (the numbers being added), operators follow their operands. Instead of `3 + 4`, RPN would say `3 4 +`. If you want to write a more complex expression, just note that each operation consumes two operands and leaves a resulting operand in its place. So to write `(3 + 4) * 7`, RPN would say `3 4 + 7 *`, which gets computed by evaluating the leftmost operator in the expression and replacing the operator and its operands with the result, and continuing until there is only one value left:

`3 4 + 7 *` evaluates `3 4 +`, replacing that with `7`:

`7 7 *` which evaluates `7 7 *`, replacing that with `49`:

`49` at which point we are done.

Notice that RPN does not need parentheses to encode order of operations. You can always evaluate operators left to right. `3 + 4 * 7` (which would be `31`) is written in RPN as:

`3 4 7 * +`, which evaluates `4 7 *`, replacing that with `28`:

`3 28 +`, which evaluates `3 28 +`:

`31`

Where in normal notation we would need parentheses to distinguish `(3 + 4) * 7` from `3 + 4 * 7`, RPN captures the order of operations by changing the order of the operators.

RPN is also called 'postfix notation", because the operators follow their operands. The 'normal" arithmetic notation you are used to is 'infix notation."

2.2 Stacks

A stack is a container (a data structure that holds items) that implements a *last in first out* (LIFO) policy: you can only retrieve items from the data structure in the reverse order that you put them in. Stacks can be easily implemented by linked lists by restricting how you add and remove elements:

- Adding an element always places it at the front of the linked list (i.e., the head pointer will point to the new element). This operation is called a **push**.
- Removing an element always removes the first element in the linked list (i.e., whatever head points to). This operation is called a **pop**

Note that we've seen these terms, since a *program stack* is exactly this type of stack. When you call functions you *push* new frames on to the stack, and when you return from functions, you *pop* those frames off the stack, in the reverse order that you called the functions.

2.3 Using stacks to evaluate RPN

The classic algorithm to evaluate RPN expressions (which you can also read about on the Wikipedia page) uses a stack to hold the *operands* (the data being operated on). The algorithm proceeds by scanning the expression from left to right, and starting with an empty stack.

- If you see an operand (a number), *push* it onto the stack.
- If you see an operator (addition, subtraction, etc.), *pop* two operands off the stack, combine them using the operator, then *push* the result onto the stack. (If there aren't two operands on the stack, the expression is malformed.)

Continue until you have processed all of the expression. At the end of the process, there should be one operand left on the stack: the result of the expression. (If there is more than one operand on the stack, the expression is malformed.)

Let us see how this algorithm lets us evaluate `3 4 7 * + 2 /`. The stack starts out empty: `[ ]`.

1. Read a `3` and push it onto the stack, which now looks like `[3]`.
2. Read a `4` and push it onto the stack: `[4 3]`.
3. Read a `7` and push it onto the stack: `[7 4 3]`.
4. Read a `*`. This is an operator, so pop two elements off the stack (`7`and `4`) and multiply them (`28`), pushing the result onto the stack: `[28 3]`.
5. Read a `+`. Pop two elements off the stack (`28` and `3`) and add them (`31`), pushing the result onto the stack: `[31]`.
6. Read a `2` and push it onto the stack: `[2 31]`.
7. Read a `/`. Pop two elements off the stack (`2` and `31`) and divide them (`15.5`), pushing the result onto the stack: `[15.5]`. (Pay careful attention to which operand was the divisor!)

Now we're done with the expression and there is only one operand left on the stack: the answer (`15.5`).

3 What you need to do

You need to implement an RPN calculator that will read in an RPN expression from a file (specified as a command line argument) and print out (to `stdout`) the result of performing the computation. No other output is required.

If the input file is malformed such as having too few numbers for the operands (3 4 + -), or too many numbers for the operations (3 4 2 *), then your program should return EXIT_FAILURE.

In this assignment, we are giving you very little starter code: just a pa08.c file that shows you how we expect your program to be invoked. It is your job to implement a stack data structure (feel free to modify the linked list data structure from PA07), the code to read in the expression from the file, and the calculator itself. Note that the calculator should operate on floating point values.

3.1 Development strategy

We recommend you adopt an incremental development strategy here. One possible development plan:

1. Implement and test your stack data structure.
2. Implement and test code to read in an RPN expression. You may find it easiest to read in one 'word" at a time, determine whether that word is an operand or an operator, and act accordingly. You may also find the stdlib.h function atof helpful. Keep in mind atof returns a double (and for this lab we want to use floats).
3. Implement and test your calculator. We have included a couple of RPN expressions in the inputs directory. Feel free to create and share additional test expressions on Piazza.

3.2 What to turn in

1. All source files you wrote, including pa08.c
2. A Makefile that specifies a target called all that builds your calculator, producing an executable that *must* be called pa08. Your Makefile should also contain a target called clean that removes all intermediate files.