Homework 7: Reverse Polish Notation Calculator

**Assigned**: April 6thth, 2018 **Due**: April 21th, 2018

Make a subdirectory "hw7" of your cs240 folder for this assignment. Copy the starter files from /courses/cs240/s18/kamaral/GROUP/hw7 to your hw7 subdirectory.

The io.o module is already compiled for you, but incase you want to modify it, the source is also available as io.c. You shouldn’t need to modify this file.

The following functions are implemented for you already in the io module.

* int parseoperator(char\*);
	+ Extracts an operator from input and passes it back through call-by-reference.
	+ If it fails, returns false. Otherwise returns true.
* int parseoperand(int\*);
	+ Extracts an operand from input and passes it back through call-by-reference.
	+ If it fails, returns false. Otherwise returns true.
* int endofstream(void);
	+ Checks if the end of input has been reached.

To extract the next token using these functions, the functions should be used in conjunction.

* Attempt to parseoperand
	+ If it succeeds, store the operand with the stack module.
	+ If it fails,
		- Check EOF with endofstream
		- If not EOF, try parsing operators instead
* Attempt to parseoperator,
	+ If it succeeds, process the operator with the calc module.
	+ If it fails,
		- Check EOF with endofstream
		- If not EOF, that means the next token exists AND it’s not an operator or an operand. This is an error.

endofstream will only return true (true means we have reached end of stream) if an attempt to read the standard input has failed as a result of reaching end of stream. This value is meaningful only after calls to parseoperator and parseoperand. It will not pre-emptively detect end-of-file before calling those functions, so it should not be used to assert the success of the parse functions before they’re called.

**Goal: Reverse Polish Notation Calculator**

The overall goal of this program is to create a program that calculates reverse polish expressions for any integers with the following operators:

* Addition (+)
* Subtraction (-)
* Multiplication (\*)
* Division (/)
* Remainder (%)
* XOR (^)
* Bitwise AND (&)
* Bitwise OR (|)
* Unary Not (~)

Example Reverse Polish expressions:

* 123 21 + 567 432 - \*
	+ Algebraic/In-fix Notation: (123 + 21) \* (567 - 432)
	+ Answer: 19440
* 987 10 / 10 %
	+ Algebraic/In-fix Notation: (987 / 10) % 10
	+ Answer: 8
* 127 4 ^ ~
	+ Algebraic/In-fix Notation: ~(127 ^ 4)
	+ Answer: -124
* 9 6 & 2 127 & | 127 –
	+ Algebraic/In-fix Notation: ((9 & 6) | (2 & 127)) - 127
	+ Answer: -125

The program takes as a single line of input a reverse polish expression. After reading the entire line of input and processing it, the program outputs the result as a single integer value.

If there are any errors, the program outputs an error message and should then terminate using exit(EXIT\_FAILURE) from stdlib.h.

Example:

Enter Expression (Reverse Polish): 9 6 & 2 127 & | 127 –

Result: -125

**Part 1: Stack Module**

In the stack module, you’ll be implementing the stack functionality, as we discussed in class. The slides on stack include prototypes which your function definitions should match.

Make sure that this file has the following functionality:

* push
* pop
* empty

As you see fit, feel free to include one or more of the following additional functions if you need to use them:

* peek
* size

The stack module has two important features that should be hidden from other modules to avoid breaking the data structure. Make sure that the underlying storage and global variables for the stack are only visible within the same file using the static keyword.

Make sure that the functions are visible because that is how other modules will interact with this module.

**Part 2: Calc Module**

The calc module is responsible for doing the arithmetic associated with the reverse polish notation calculator.

It only needs to have one function implemented,

* void eval(char);
	+ The argument to this function is going to be an operator extracted from input.
	+ This function should take the operator, the top two operands on the stack, and evaluate the three of them together, similar to how Homework 6 was done.
		- In the case of the unary not operator (~), instead of two operands, the eval function should evaluate it with only the top operand.
	+ The result of the evaluation should be pushed onto the stack and not returned.

This module does need to make use of the functions in the stack module, so it should declare those functions as external using the extern keyword.

If the necessary stack functions fail (return false) the calc module should print an appropriate error message and exit(EXIT\_FAILURE) from stdlib.h.

**Part 3: Main Module**

This is the primary piece of the program responsible for the overall operation. This is where the high-level logic for the Reverse Polish Notation program goes.

The main function should be the only function in this file.

This function should iterate over every token from the input using the io module.

Once all input has been read, it should check for the result in the stack module.

If the stack module contains more than one result, or less than one result, the program should print an error message and exit without printing an erroneous result.

If everything is hunky-dory, it should print the result as mentioned earlier.

Since this module needs to access functions from every other module, make sure to extern them.

**Part 4: Makefile**

Create a makefile for the program. The first rule of the makefile is the default rule and should be named after the program it’s going to create. Let’s call this program polish.

There should be the following build rules:

* polish
* calc.o
* stack.o
* io.o
* main.o

Remember, every makefile rule has 3 parts:

* rule name (usually matches the name of the file it’s creating)
* dependency list
* build commands

Be sure to provide accurate rules for all 5 parts of the program.

**Final Remarks**

For this homework, in the makefile do now show manual compilation of the program. I don’t want to see any gcc commands. Instead, use the makefile command make. If you’ve built your makefile correctly, running the make command just does everything for you. This makes your job in making the typescript that much easier. All other typescript expectations remain the same.

Also, when you turn in your typescript, please include runs of your program on at least the example expressions listed above.