MP12: Postfix Notation Evaluator

Due Date: Wednesday Dec. 10. 2014 at 10 pm

Overview

Your task this week is to create a program capable of evaluating an expression in postfix notation (reverse polish notation) using trees. Postfix notation is a notation where every operator follows all its operands. So for example “3 - 2” would be represented as “3 2 -”. If there are multiple operations, the operator is given immediately after its second operand; so the expression written "3 − 4 + 5" would be written "3 4 − 5 +".

This assignment must be completed on your own.

The Pieces

`main.c` - The `main` function for this program. This gets the user input and calls the functions you must write. DO NOT MODIFY.
`mp12.c` - This file will have all your code to create the expression tree and evaluate it. It also contains code for a stack structure you can use.
`mp12.h` - All function definitions and structures are defined here. Look at it for more info on the structures and functions.
`Makefile` - This contains instructions to compile this code. If you decide to add more files change this file so your code compiles. To compile your code we will simply use your `Makefile` and the `make` command.

Details

There are four functions you will need to implement for this program.

```
void postfix(char * exp_str);
node * create_postfix_tree (char * exp_str);
int evaluate_postfix (node * curNode);
void delete_tree (node * curNode);
```

`postfix` is the top level function that when given an expression string will create the expression tree, evaluate the expression tree, **print the solution**, and then delete the tree. For simplicity the expression string will only have single digit values as its operands (0 - 9). The expression string will have spaces between each item. *Hint: This will allow you to read a character at a time from the input string and do something depending on the character value.*

Here are a couple examples:

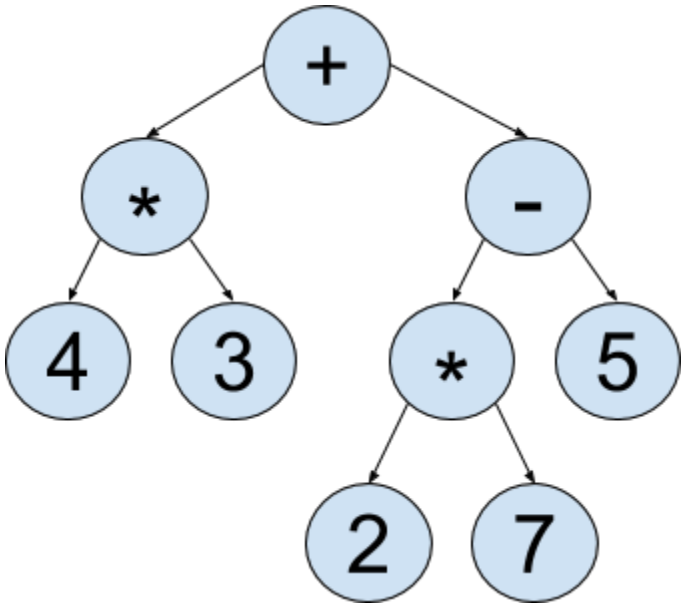
Input (postfix notation)	Infix notation (conventional)	Output
3 4 -	3 - 4	-1
2 5 * 4 + 3 2 * 1 + /	(2 * 5 + 4) / (3 * 2 + 1)	2

You can assume all input strings will be valid postfix expressions. (Nothing will be incorrectly formatted)

`create_postfix_tree` will, given an expression string, create an expression tree. We have provided you with the structure for a node in the tree. Each node must have memory allocated for it (malloc). The algorithm for creating this tree will be discussed in lab.

```
struct node {
    char item;
    node * left;
    node * right;
};
```

`item` will hold the item for this node (0 - 9) or the possible operators (+ - * /)
`left` points to the left node
`right` points to the right node
So for example given the expression " 4 3 * 2 7 * 5 - +" you should produce the tree below:



`evaluate_postfix` should, given a pointer to the root of a tree, traverse the tree and return the evaluated expression. So for the previous example `evaluate_postfix` should return 21. Remember each node contains a character so if the character is a number you must convert it to a numeric value.

`delete_tree` should, given a pointer to a tree, traverse the tree and delete it freeing any allocated memory.

We provide you with a stack structure to use in any of your code. The stack structure contains an array of up to `MAXSIZE` node pointers and an integer `top` which specifies the top index of the stack. There are three functions you can use with a stack structure.

```
void stackInit(stack * myStack);
void stackPush(stack * mystack, node * element);
node * stackPop(stack * myStack);
```

`stackInit` will initialize a given stack (sets top to -1).
`stackPush` will push an element onto the given stack.
`stackPop` will pop the top element off the stack and return it.

Only use the provided libraries.

Challenges

(20 points) For this challenge your program must be functional for infix notation expressions. To run your infix code run the program with 1 as the argument (./mp12 1). To get points for this challenge your code MUST take an infix expression and convert it to a postfix expression then use the code already completed for this MP to evaluate the expression. This conversion from infix to postfix can be accomplished by following the Shunting-yard algorithm. Please note infix expressions can contain parentheses and order of operation matters. If you create any additional files make sure you provide a functional Makefile to compile your code. Otherwise we won't be able to properly compile and grade your code. Again you can assume all inputs will be valid infix expressions.

Building and Testing

To build and execute the program run the following commands:

```
make
./mp12 0
```

The command line argument indicates that the program should run for postfix notation expressions. Set the argument to 1 (./mp12 1) to run the program for the infix notation.

gdb can be used to help track down bugs.

Grading Rubric

Functionality (70%)

- (5%) - `postfix`
- (30%) - `create_postfix_tree`
- (25%) - `evaluate_postfix`
- (10%) - `delete_tree`

Style (15%)

- (5%) - Compilation generates no warnings.
- (10%) - Indentation and variables names are appropriate and meaningful

Comments, Clarity and Writeup (15%)

- (5%) - Introductory paragraph explaining what you did. Even if it is required work.
- (10%) - Code is clear and well commented.

Note that some point categories in the rubric may depend on other categories. If your code does not compile, you may receive a score close to 0 points.