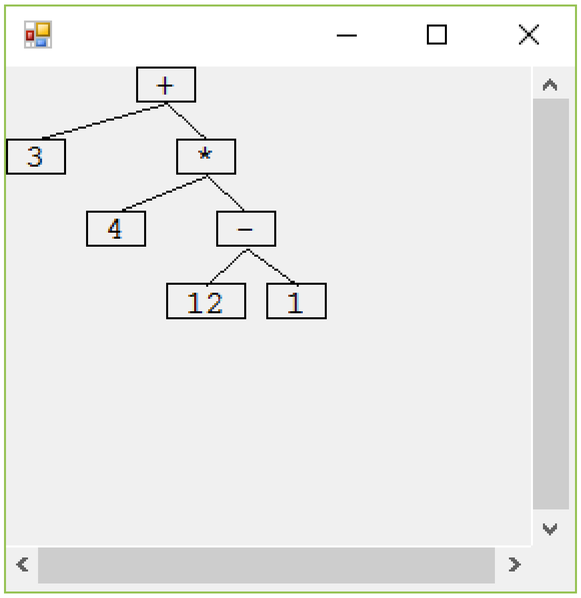
Expression Trees

An expression tree is a kind of binary tree used for representing arithmetic expressions. For example, the infix expression:

*3 + 4 * (12 - 1)*

Has the corresponding expression tree:



Note that leaves are always operands (numbers, in this case), and non-leaf nodes are always operators (+, -, *, /). The child trees of any operator are the two expressions that the operator is operating on. For example, in the "*" node above, it is operating on 4 (the left subtree) and 12 - 1 (the expression resulting from the right subtree).

We can also write arithmetic expressions in prefix or postfix notation. In prefix notation, operators go BEFORE what they are operating on. In postfix notation, operators go AFTER what they are operating on.

The infix expression 3 + 4 * (12 - 1) can be written as:

Prefix: + 3 * 4 – 12 1

Postfix: 3 4 12 1 - * +

Once we have an expression tree, we can write it in either infix, postfix, or prefix notation by doing an inorder, postorder, or preorder traversal on the tree.

A postorder traversal is done by:

```
If the current tree isn't empty
    Recursively traverse to the left subtree
    Recursively traverse to the right subtree
    Visit/print the data for the current tree
```

We can similar to an inorder traversal by putting the "visit/print the data for the current tree" in between the two recursive calls. A preorder traversal can be done by putting the "visit/print the data for the current tree" before the two recursive calls.

Starting the assignment

Create a GitHub repository using this URL and clone it to your machine. This repository contains:

- *types.h*
- *tree.h and tree.c*
- *stack.h and stack.c*
- *proj4.c*
- *.gitignore*

You will need to make changes to the *tree.c*, *stack.c*, and *proj4.c* files as part of this project. **You will also need to write a Makefile (following the template here) to contain build and clean instructions for your project.**

Assignment description

You are to write a program in C that gets a postfix expression from the user along with the user's choice on whether to convert to prefix or infix. Your program should then build an expression tree representing the expression and use that expression tree to complete the desired conversion. It should also print the evaluation of the original expression (by using the expression tree).

Here is a sample run of the program: (user input is in red):

```
Enter postfix expression: 2 3 +
Enter 1 for infix, 2 for prefix: 1
```

```
(2 + 3)
Answer: 5
```

Here is another sample run of the program: (user input is in red):

```
Enter postfix expression: 20 17 4 * + 5 -
Enter 1 for infix, 2 for prefix: 2

- + 20 * 17 4 5
Answer: 83
```

Running your program

You will need to write a *Makefile* with build instructions for your program. It must compile by command-line with the statement:

```
make
```

This should produce the executable *proj4*.

Your program should then run with the statement:

```
./proj4
```

Structuring your program

The included header files (*types.h*, *tree.h*, and *stack.h*) are finished. You should not make any changes to them other than to add documentation to the top of each file. You will need to finish the *tree.c*, *stack.c*, and *proj4.*c files.

Finishing *tree.c*

You will need to finish the *create_singleton*, *build_tree*, *inorder*, *preorder*, *evaluate*, and *free_tree* functions. Follow the documentation in *tree.h* to see what each function is supposed to do.

Finishing *stack.c*

You will need to finish the *init_stack*, *push*, *pop*, and *free_stack* functions. Follow the documentation in *stack.h* to see what each function is supposed to do.

Finishing *proj4.c*

Follow the instructions in *proj4.c* to complete the main function for the project.

You are welcome to add additional functions to any .c file.

Other requirements/assumptions

- You may assume the user will enter a valid postfix expression that consists of nonnegative integers (which might have multiple digits) and operators (+, -, *, or /). Assume that all numbers and operators are separated by spaces (you will want to use the *strtok* function with " " as the delimeter to break apart the pieces of the input string).
- You may assume that the user will enter either "1" (for inorder traversal) or "2" (for preorder traversal) – assume they won't enter any other number or a non-integer.
- You MUST store the inorder and preorder functions in an array of function pointers in the main function. (The array should be type *trav_fn*.) Use the user's numerical input (1 or 2) and subtract one to get the correct index in that array and call that traversal function. You should NOT use an if-statement on the input to determine which traversal to call.

Constructing an expression tree from a postfix expression

We can use the following algorithm to construct an expression tree from a postfix expression:

```
Create a stack of expression trees
For each piece in our postfix expression
   If the current piece is a number
      Create a singleton tree with that number and push it on the stack
   If the current piece is an operator
      Pop the stack twice (call the first tree1 and the second tree2).
      Create new expression tree:
         The current operator is the root,
         tree1 is the right subtree, and tree2 is the left subtree
      Push that new tree onto the stack
Afterwards, the only item on the stack is the overall expression tree
```

Other test cases

Here are two other test cases for your project (user input is in red):

```
Enter postfix expression: 17 4 – 2 – 10 5 / +
Enter 1 for infix, 2 for prefix: 1

(((17 – 4) – 2) + (10 / 5))
Answer: 13
```

```
Enter postfix expression: 151 20 + 17 – 4 11 * -
Enter 1 for infix, 2 for prefix: 2

- – + 151 20 17 * 4 11
Answer: 110
```

Documentation

Your program must include comment blocks at the top of each file and at the top of each function prototype. The function comments should include a brief description of what the function does and an explanation of any function arguments and return values.

All the provided function prototypes already have documentation – you can keep those as-is unless you want to provide more details. You do need to add documentation to the top of each file and to the top of any new function prototypes.

Use the comment block below as a template for the top of each file:

```
/*
 * Name: (YOUR NAME)
 * Lab: (YOUR LAB SECTION – Tuesday or Thursday)
 * Assignment: Project 4 – Expression trees and traversals
 *
 * (WRITE A DESCRIPTION OF THIS FILE)
 */
```

Use this comment block as a template for function prototypes:

```
/*
 * (WRITE A DESCRIPTION OF THE FUNCTION)
 *
 * (PARAMETER NAME): description (repeat for each parameter)
 * (If return is non-void, describe what is being returned)
 */
```

Submission

Add all your edited files and then commit and push your changes to GitHub. Submit the entire URL of the commit that you want graded (it should look like ***.../ksu-cis308-fall2022/project-4-(GitHub username)/tree/(hex value)***.

Grading

Projects that do not compile will receive a grade of 0. It is your responsibility to make sure your program compiles before submitting it. Projects that do compile will be graded according to the rubric for this assignment.