Background

In this assignment you will build a set of functions to operate on nodes of an expression tree (also called an abstract syntax tree). The trees in this assignment can be used to represent expressions in a simple calculator. Here is a sample tree:

```
+  
42 *  
    abc 29
```

In this example, there are two binary operator nodes (+ and *), two integer nodes (42 and 29), and one variable node (abc).

There are two parts to this assignment. First you will implement the API described to build these trees. The purpose of this assignment is to provide practice writing pointer-based data structures in C. The second part of the assignment is to write test cases for this API. Your test cases will be graded based on two criteria.

The first grading criterion is the degree of code coverage provided by your tests. Code coverage is a metric of how many of the lines in your API implementation are executed by the test cases you write. The more lines of code your test cases are able to cover, the higher your score on this part of the assignment. To record code coverage, you will need to invoke the gcc compiler with the following additional options: -fprofile-arc -ftest-coverage. These should be added to your CFLAGS macro in the project Makefile. You can use the lcov visual code coverage browser to see what statements were covered or not.

The second criterion in grading your test suite will be to use it to find bugs in a set of buggy implementations of the API that we will provide you. We will supply 3 tests cases in binary form (which will be the public tests for this assignment) and you will link your test suite to the buggy versions.

API

```
ASTnode *createOperatorNode(opType operator, ASTnode *left, ASTNode *right)
```

Create a new node with the passed children. Here are the operators and their definitions:
<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>plusOperator</td>
<td>Add the two sub-tress</td>
</tr>
<tr>
<td>minusOperator</td>
<td>Subtract left sub-tree from right</td>
</tr>
<tr>
<td>multOperator</td>
<td>Multiply the two sub-trees</td>
</tr>
<tr>
<td>divOperator</td>
<td>Perform integer division of the left / right sub-tree</td>
</tr>
</tbody>
</table>

ASTnode *createConstantNode(int constant)
Create a leaf node of type constant.

ASTnode *createVariableNode(char *variableName)
Create a symbolic variable expression. You should make a copy of the variableName character array.

ASTnode *copyTree(ASTnode *node)
Create a copy of the passed tree. The copy should share no data with the item it is being copied from (i.e. copy all sub-trees and other dynamically allocated space).

void freeTree(ASTnode *node)
Free the tree node rooted. This should free all children nodes (and any auxiliary memory such as variable name strings).

void printRPN(ASTnode *node)
Print out the tree rooted at node using reverse polish notation (for example, the tree given in the above example would be printed as:
(42 (abc 29 *) +)

void printInfix(ASTnode *node)
Print out the tree rooted at node using infix notation. (for example, the tree given in the above example would be printed as: 42 + abc * 29. Your solution should only print parentheses when required by operator precedence. For example, you should print 42 + abc * 29 rather than 42 + (abc * 29).

int evaluateNode(ASTnode *node, int *result)
Evaluate the tree rooted at node. If the evaluation is an integer value (i.e. there are no variable expressions in the tree), return a value of 0 and store the result of evaluating the expression in result. If the expression is contains one or more variable terms, then return -1.

int simplify(ASTnode *node, int subTree)
Replace all sub-trees that are constant expressions by a constant node with the results of the evaluation. If the value of subTree is non-zero, you should simplify any sub-trees that evaluate to constant expressions even if the overall tree rooted at node can’t be simplified. The return value should be 0 if the tree was simplified and –1 if no simplification was possible.
Test Suite

You should write a test suite that provides as complete of testing of your API implementation as possible. You should write your test programs as a series of functions in the file tests.c. The function testAPI should invoke each of your tests as appropriate. The return value from testAPI should be 0 if the implementation being tested passed all tests, and non-zero if any test failed. Your tests may produce output if you wish, but the only thing that will determine your grades for detecting faulty API implementations is that return value from testAPI.

We will also test your implementation of the AST API using a series of private tests. Private tests are run after your assignment has been submitted, but you will not see the results until we return your project to you (i.e. you can’t run private tests multiple times like release tests).

RPN Calculator

To illustrate a use of your API, we have supplied an implementation of an RPN calculator to let you try out the API. The calculator is defined in the file rpn.y, and the supplied makefile will compile it (along with your API) into the program rpn. In an RPN calculator, numbers are entered in form (<operand1> <operand2> <operator>). The calculator will read the input and print it back out in in-fix notation (<operand1> <operator> <operand2>). All expressions must be included in parenthesis. Here are some samples of valid input:

(3 4 +)
((a 5 *) (6 7 +) (/)

Challenge problem (for fun, no points)

Combine your api implementation with your previous project on hash tables to build a calculator that supports symbolic variable names. You can store in the hash table an item of the form:

typedef struct {
    char *name;
    float value;
} variableValue;

You will need to add one more method to your AST API:

ASTnode *createAssignmentNode(ASTnode *left, ASTNode *right)

This node will represent an assignment of a variable to an expression. For example:

```
= abc 22
```
If you call evaluate on this node, it should return the result of the expression on the right hand side, but also as a side effect, update the hash table to store the value of the right-hand expression into the location of the variable name.

When you call evaluate, any expression that contains a variable expression that has a value in the associative array should return that value.