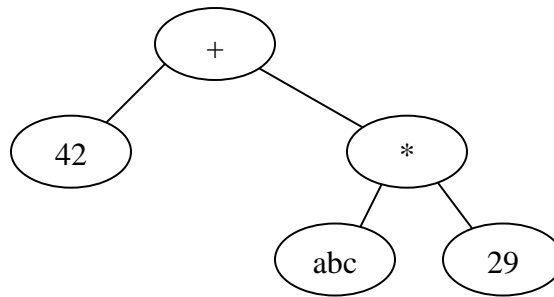


CMSC 212
Project #4
Due 11/6/2005 11:59 PM

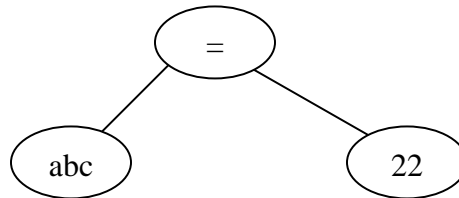
Background

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



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

The AST may include symbolic variables. The values of symbolic variables are defined via assignment nodes, as below, and used in variable nodes. You will store values via an associative memory table created with the routines in project P2.



Evaluating this tree should return the result of the expression on the right hand side. However, evaluation of an assignment node also has the side effect of updating the associative memory to store the value of the right-hand expression into the location of the variable name. During evaluation of ASTs, the value of a variable node is the value of the last expression assigned to it.

What to do:

There are two parts to this assignment. First you will implement the API described below in order to build these trees. The purpose of this part of the 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:

- i) 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-arcs -ftest-coverage`. These should be added to both the CFLAGS and LDFLAGS macros in the project Makefile. Details on viewing the resulting output will be discussed in lecture/lab-sections.
- ii) 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:

Operator	Definition
<code>plusOperator</code>	Add the two sub-trees
<code>minusOperator</code>	Subtract right sub-tree from left
<code>multOperator</code>	Multiply the two sub-trees
<code>divOperator</code>	Divide left tree by right.
<code>assignOperator</code>	Perform assignment, value is right-hand side

`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, always using parenthesis and spaces to demarcate expressions. 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. Your solution should only print parentheses when required by operator precedence. For the first example above, 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 complete (i.e. there are no undefined variables in the tree), return a value of 0 and store the result of evaluating the expression in result. If the expression can not be evaluated, then return -1.

int simplifyNode(ASTnode *node, int subtree)

If ‘subtree’ is zero, attempt to simplify the tree rooted at ‘node’ by replacing it with a constant node, if and only if the entire tree rooted at ‘node’ is constant-valued.

If ‘subtree’ is non-zero, simplify any sub-trees that can be simplified even if the entire tree can not be simplified.

The return value should be 0 if the tree was simplified and -1 if no simplification was possible.

Notes:

- 1) Nodes won't be used more than once. E.g., we'll never call “createOperatorNode(plusOperator, c1, c1)”, where “c1” is a pointer to a constant node.
- 2) Use a single associative table for the entire run of a program. This means that an assignment done while evaluating one tree may be used in another.
- 3) Our test cases *will not* include any assignments to variables that are also referenced in the same sub-tree.
- 4) Just as a reminder, you test coverage by:
 - a) compile AND link w/ -fprofile-arcs -ftest-coverage
 - b) “tap lcov” to get lcov in your path

- c) `lcov -c -d . -o coverage.out`
- d) `genhtml coverage.out`
- e) `lynx index.html`

5) Coverage is only 8 pts, so heroic measures are not necessary.

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).

NOTE: Though your AST's implementation of symbolic variables will be checked, the "tests.c" you turn in should *not* include any assignment to symbolic variables.

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`. All this "calculator" does is accept an RPN expression as defined above (i.e. "(4 3 -)"), and print it in infix notation "4 + 5". Expressions can not be nested (i.e. ((4 3 -) 2 *) is not valid).

Files

Place the source for your implementation in `ast.c`, and the source for your tests in `tests.c`. You also need a copy of your `assocMemory.c`, which you will be calling from `ast.c`. We provide several buggy AST implementations to run your test suite against, as well as the binary object of an implementation of P2.