ECE 264: Advanced C Programming

Programming Assignment 9 - Expression Trees, Due 7/31/19

This assignment is a bonus assignment. This is extra credit (it has the same weight as all other assignments)

This assignment is the second in a two-part series on building a calculator. It follows on from PA08, but does not directly build on it (you do not need a working PA08 to successfully complete this assignment). In PA08, you had to build an RPN calculator that took a postfix expression and computed the result. In this assignment, you will build a tool that converts an *infix* expression (i.e., an expression that looks 'normal") into a postfix output that the PA08 calculator can operate on.

This assignment essentially asks you to write a very basic compiler. We provide some additional background about compilers in note boxes (like this one) throughout the background section, but you can ignore them if you want. If you're interested in learning more about how compilers work, take ECE 468.

1 Learning Goals

In this assignment, you will learn

1. Basic expression parsing
2. Building a tree bottom-up
3. Performing a post-order traversal of a tree

2 Background

2.1 Fully parenthesized infix expressions

Infix expressions are the 'normal" way you're used to seeing arithmetic expressions written out (e.g., `2 + 3 * 7`), with all the complications of order-of-operations, etc. We will use a simpler version of infix expressions that we will call 'fully parenthesized": every binary operation is enclosed in parentheses. So `2 + 3 * 7` would be written `(2 + (3 * 7))`. Note that because we put parentheses everywhere, we can ignore order of operations.

To precisely capture what a fully parenthesized expression is, we can define a fully parenthesized infix expression *recursively.* A fully parenthesized expression is:

1. A (floating point) number
2. An open parenthesis '(' followed by a *fully parenthesized expression* (which also follows these rules) followed by an operator (`+`, `-`, `*`, or) followed by another *fully parenthesized expression* followed by a closed parenthesis ')'.

Defining expressions and other things recursively is very common in computer science/engineering, so people have come up with notation to directly capture the recursive rules laid out above. We would express these rules as:

1. `E -> lit`
2. `E -> (E op E)`

2.2 Basic expression parsing

The process of determining whether a string is a valid fully-parenthesized expression and breaking that expression into its components (‘this is a number”, ‘this is a parenthesized sub-expression”, etc.) is called *parsing*. The way to do this is to look at the rules that define a valid fully-parenthesized expression and decide whether a given string obeys those rules (i.e., you could use those rules to generate the string).

There is a whole sub-field of computer science devoted to determining how to parse strings (in fact, a parser is one of the major components of a compiler: it determines what parts of your code are if statements, what parts are variable declarations, etc.). But our rules for fully-parenthesized expressions are simple enough that we can write a simple recursive function to decide whether an expression is valid.

2.3 Tokenization

The first step in parsing is to *tokenize* a string: to turn a string of characters into a string of "words" that represent the individual pieces of an expression. We call these words *tokens*. In other words, a tokenizer's job is to take a string like:

`(3 + (7 * 8))`

And tell you the individual words (tokens) in the string:

`LPAREN VAL ADD LPAREN VAL MUL VAL RPAREN RPAREN`

In this case, `LPAREN` means the character `(`, `VAL` means any number, `RPAREN` means the character `)`, and `ADD` and `MUL` are (hopefully) self-explanatory.

It might seem like adding an extra step to turn a fairly simple expression into a bunch of tokens, but this is useful because that way later stages of parsing don't have to worry about things like skipping over white space, or deciding whether `8))` is one word or three (in this case, it's three!).

The tool that tokenizes a string is called a *tokenizer* or a *scanner*. We have written a scanner for you (in `scanner.c`) that reads in a file, breaks it up into tokens, and lets you find the next token in the string by calling `nextToken`. Pay close attention to the `Token` data type defined in `scanner.h`!

2.4 Recursive parsing

Given a tokenizer, we can now write a simple recursive function to tell whether a string is a valid, fully-parenthesized expression. Here is the algorithm, which you can wrap in a function that returns a `bool`.

1. Get the next token
2. If the next token is a `VAL` (matches rule 1), this is a valid fully-parenthesized expression. Return `true`.
3. If the next token is an `LPAREN` we now try to match all of rule 2:
    (a) We have already seen the `LPAREN`, so the next thing we expect to see is a fully-parenthesized expression. We can just call *this same function* recursively to do that! If the recursive call returns `true`, it means we have found a fully-parenthesized expression

    (b) The next part of rule 2 is to match an operation, so we grab the next token and see if it is an `ADD`, `SUB`, `MUL`, or `DIV`. If it is, we continue.

    (c) Then we call this same function recursively again to find another fully-parenthesized expression.

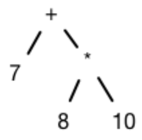
    (d) Finally, we grab the next token to see if it is an `RPAREN`. If it is, we have matched rule 2, and this is a fully-parenthesized expression, so we return `true`.

If any of those steps fail, that means we don't see what we expect, and the string is not a fully-parenthesized expression, so we should return false.

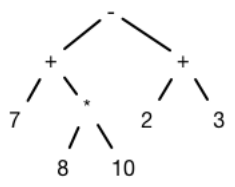
The recursive function we wrote is a simplified version of a *recursive descent parser*. These types of parsers are not quite powerful enough to decide, for example, if a given program is a valid C program (the rules for valid C programs are too complicated). But they're still pretty powerful.

2.5 Expression trees

A useful way to represent a fully-parenthesized expression is using an *expression tree*. This is a binary tree where each *leaf node* (node with no children) is a number, and each *interior node* (a node with two children) is an arithmetic operation. The structure of the expression tree matches the parentheses: each fully-parenthesized (sub) expression becomes its own (sub) tree. So, for example, (7 + (8 * 10)) has an expression tree that looks like:



While a more complicated expression like: ((7 + (8 * 10)) - (2 + 3)) has an expression tree that looks like:



(Note that the expression (7 + (8 * 10)) appears as a subexpression here, so its expression tree appears as the left subtree in the larger tree.)

2.6 Building expression trees

Interestingly, you can *build* an expression tree from a fully-parenthesized expression during the process of parsing it (using the recursive approach from above). The key is the concept of building a tree 'bottom up".

In class, we have seen how to build trees 'top down" (think binary search trees): we start by creating the root, then we add new nodes by adding children to nodes at the 'bottom" of the tree. The tree 'grows" from the top down: we create a node before we create any of its descendants.

The idea behind *bottom up* tree construction is that we create a tree by *first* creating the subtrees, then making those subtrees the left and right children of a newly-created root. In other words, we create a node by first creating its descendants.

We can modify our recursive parser from above to build trees instead. The idea is that invoking the parser function returns an expression tree that represents the (sub)expression we just matched. Rather than returning a bool, we can have it return a TreeNode *.

- In step 1, where we used to return true because we found a value, we can instead return a pointer to a newly-allocated TreeNode containing that value. (This TreeNode doesn't need left or right children – recall that in values are the leaves of an expression tree.).
- In step 2, each of the recursive calls to the parse function will return TreeNode * pointers to trees that represent the sub-expression we matched. These can become the left and right children of a new node, whose value is the *operator* we match in step 2. This new node can be returned as the root of a tree representing the expression we just matched.

2.7 Example

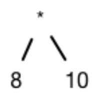
So let's see how our parse function would work for (7 + (8 * 10)):

- We start by seeing an LPAREN, so we're trying to match rule 2.
- We invoke the recursive method again.
  - We see a VAL, so we build a node containing that value, and return it:

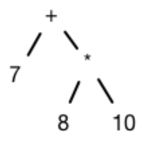
    7
- We see an ADD, which we remember, but don't do anything with
- We invoke the recursive method again.
  - We see an LPAREN, so we're trying to match rule 2.
  - We invoke the recursive method again.
    * We see a VAL, so we build a node containing that value and return it:

      8
  - We see a MUL, which we remember, but don't do anything with
  - We invoke the recursive method again.
    * We see a VAL, so we build a node containing that value and return it:

      10
  - We create a new TreeNode storing the MUL, and make its left and right children the TreeNodes we returned from the two recursive invocations:



  - We see an RPAREN, so we return the new tree.
- We create a new TreeNode storing the ADD, and make its left and right children the TreeNodes we returned from the two recursive invocations:



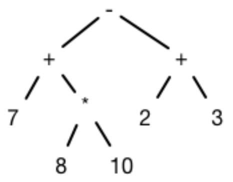
- We see an RPAREN, so we return the new tree

2.8 Postfix traversal

There are three 'standard" ways to traverse a binary tree, each of which describes a way of recursively visiting each node of the tree.

1. *inorder* recursively visits the left child, then processes the current node, then recursively visits the right child.
2. *preorder* processes the current node, then recursively visits the left child, then recursively visits the right child.
3. *postorder* recursively visits the right child, recursively visits the left child, then processes the current node.

If 'processing" a node prints out its value, note that doing an *inorder* traversal of the expression tree recovers the original infix expression (though without the parentheses, so the order of operations might be screwy). If we print this expression tree inorder:



We get 7 + 8 * 10 - 2 + 3, which looks like its source expression if you remove all the parentheses.

Printing the tree *preorder* instead gives us: - + 7 * 8 10 + 2 3.

Finally, printing the tree *postorder* give us: 7 8 10 * + 2 3 + -.

Note that *this is exactly what the postfix representation of the source expression would be.* If we create an expression tree from a fully-parenthesized expression, *we can turn it into postfix by performing a postorder print!*

3 What you have to do

Your job is to:

1. Write a recursive function to parse a fully-parenthesized infix expression and produce an expression tree
2. Write a function to traverse the expression tree in postorder and print the postfix representation of the input expression to a file.

We have given you many starter files:

1. tree.h and tree.c: data structure definitions for TreeNodes, as well as basic allocation and deallocation routines. You may find the printTree routine useful for debugging.
2. scanner.h and scanner.c: a scanner that reads in a fully-parenthesized expression from a file and creates a stream of tokens. You can invoke nextToken to get the next token from the input expression (the token definitions are in scanner.h). You may find the printToken routine useful for debugging.
3. expressionTree.h: a header file containing two function declarations: one to build an expression tree, the other to print it out in postorder to a file.
4. pa09.c: a driver file for the assignment, containing the main function.

Your job is to write expressionTree.c, which contains two functions:

1. buildExpressionTree: this is the recursive parsing function, which returns the root of an expression tree built from the input expression. The function takes a Scanner as an argument, which can be used to step through the tokens in the input expression.
2. makePostfix: this function performs a post-order traversal of the expression tree (whose root is passed in as t) and writes it out to the file (whose FILE pointer is passed in as fptr). The output file should be an RPN expression that you can pass in to your calculator from PA08.

Do not modify any files other than expressionTree.c!

4 What you have to turn in

Turn in only expressionTree.c. If you submit any other files, we will not use them: we will overwrite them with our own versions of those files during the grading process. Any helper methods you need should just be declared and defined directly within expressionTree.c (not in expressionTree.h).