The Shell Algorithms & Documentation

UC Riverside
Computer Science

I created a bash shell that accepts Linux/Unix commands and syntax in C/C++. This shell supports operators &&, ||, and ;. It also supports test and []. Finally, this shell supports redirection and piping.

My implementation consists of trees which were formed using nodes. I followed composite pattern in order to not worry about whether a node was an executable or a composite. The source code for this shell, along with the current downloadable version is available at https://github.com/cs100/cs100-assignment-2-the-meaty-crocodiles (the repository is currently private).

Finally, this shell is under the GNU license, meaning that it is open-source. Anyone may download it and tweak it to their needs.

To understand how it operates, I have created the following documentation where I review the process by which the shell interprets and creates a tree for your command-line input in order to execute it.

I start by laying out the classes created. Note that in this tree, executables are commands that you wish to be executed (such as "echo a"), while composites link two executables together ("echo a && echo b" Here, the "&&" is a composite). For a visual representation of this, see below where I animate the trees.

The Classes:

Base: Command - base class which all other classes will inherit from.

Leaf: Executable - consists of a string of args that will later be tokenized. Execute function uses execvp and the whole function returns a bool to determine success of execution.

The execute() method also contains an if-else statement. Execute can either be a normal execute from assn2, or it can be a "test" from assn3. If it is a "test", we simply execute in a different way using stat().

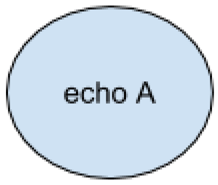
Composites: All have a left and right child ptr of type Command.

- Semi - a semicolon. The execute function first executes left child, then right child and it returns the right child's success as a bool.
- And - an &&. It returns true if left AND right child succeed, otherwise return false.
- Or - returns true if left doesn't succeed but right does, OR if left does succeed.

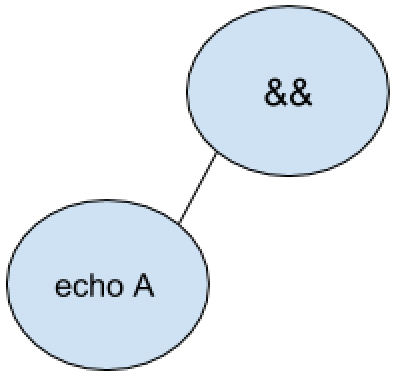
The Tree:

Suppose we have “echo A && echo B || echo C”. The goal is to build a tree where “echo A” is a leaf node, “&&” is a composite node, “echo B” is a leaf, “||” is a composite… and so on.

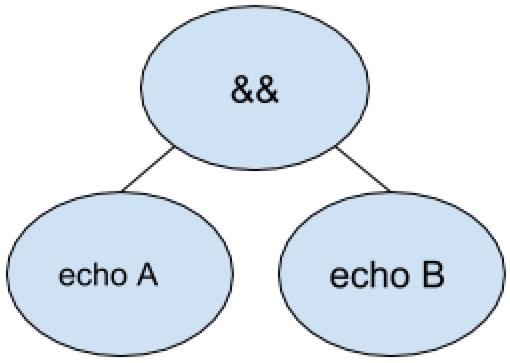
Going from left to right, start with making “echo A” an executable and so our tree is simply:



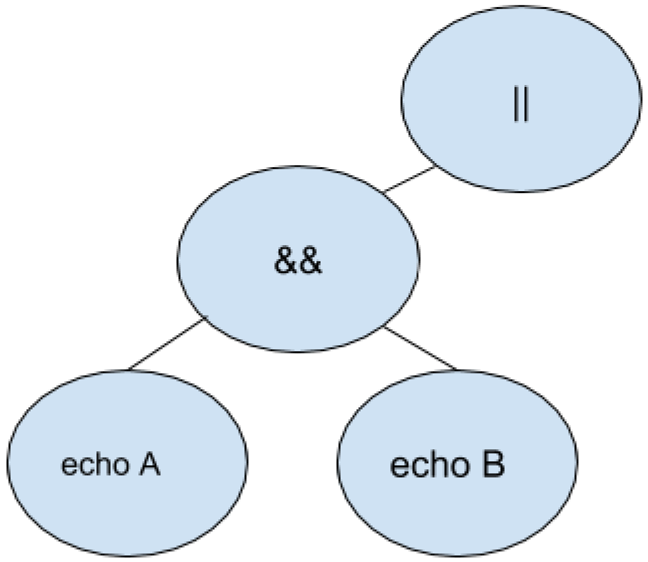
Next, we create an And object and set its left child to “echo a” and the right child ptr to 0 for now.



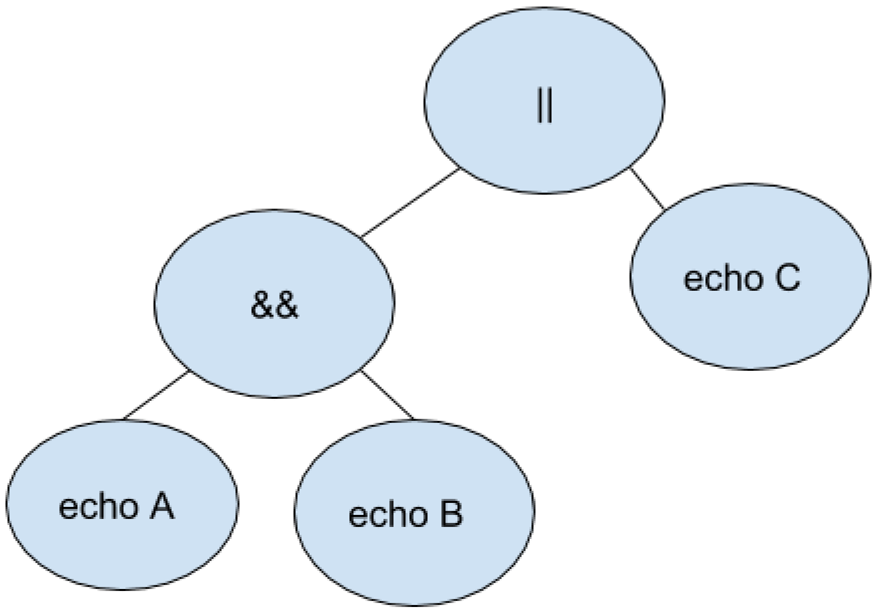
Now we create another leaf object for “echo B” and set it to the right of “&&”.



Next we create a composite object for “||” and set its left child to “&&”.

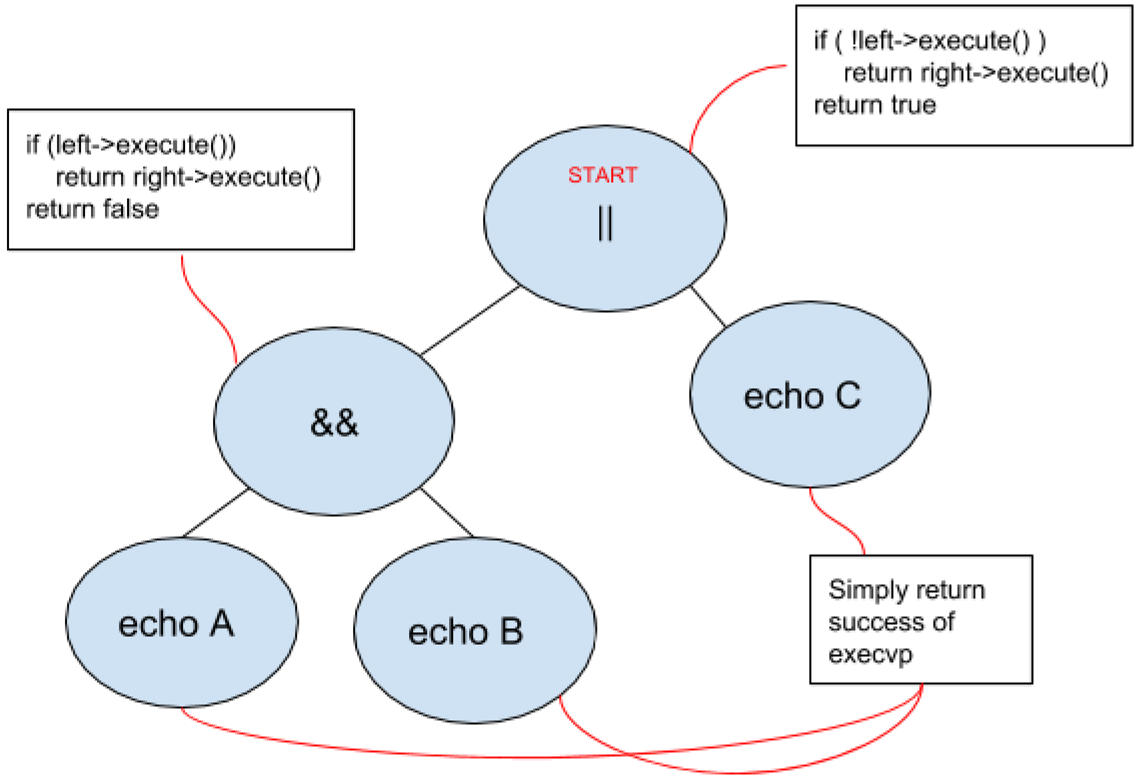


Finally, we create a leaf object for “echo C” and set Or’s right child to this newly created leaf node.

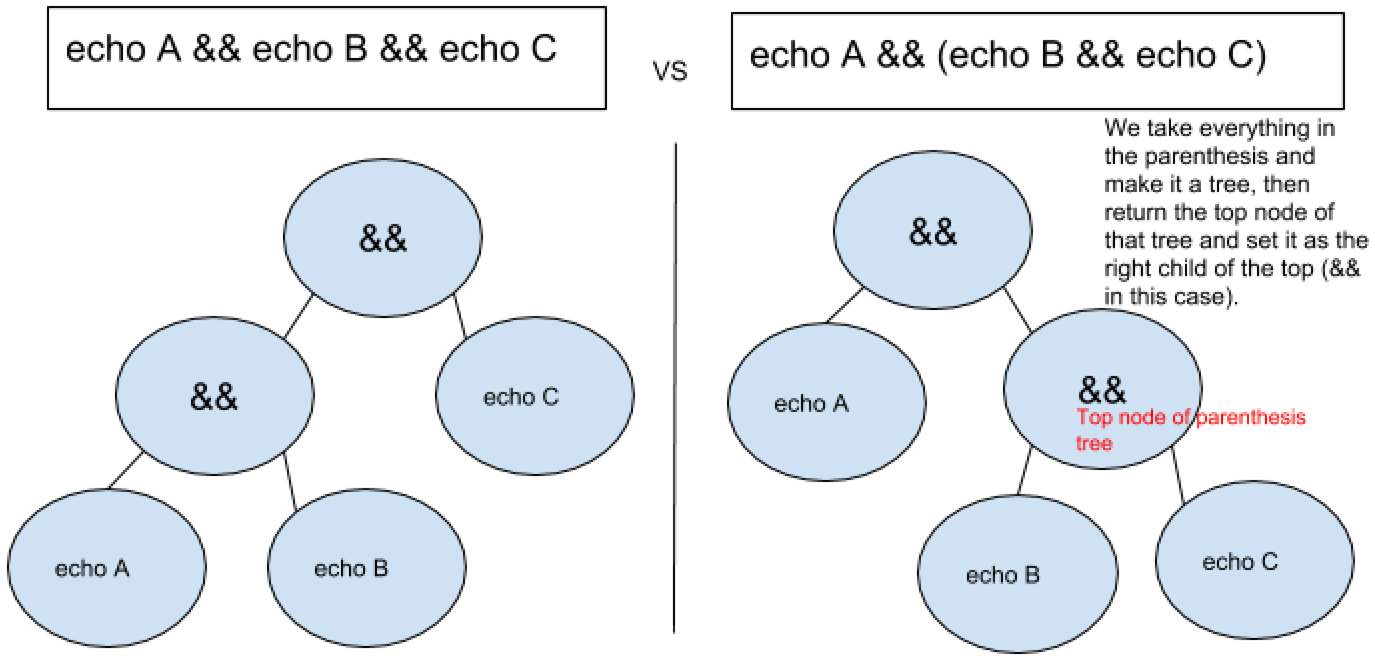


How A Tree Executes - Not Including Test or Redirection

In our above example of a tree, we have "echo A && echo B || echo C" in tree form. The program is designed so that the client (in this case, the main function) does not need to distinguish a leaf from a composite. In the main function, we will simply get the user's input and pass that string as an argument to a function called "make_tree". This make_tree function will return a pointer to the top node of the tree. After the top node is returned, the main function simply calls execute on this node, and the executes will trickle down.



Note that all a parenthesis will do is create a sub_tree. Thus the implementation is a recursive call. For example, echo A && (echo B && echo C) will result in:



echo A && echo B && echo C
vs
echo A && (echo B && echo C)
&&
&&
echo C
echo A
echo B
We take everything in the parenthesis and make it a tree, then return the top node of that tree and set it as the right child of the top (&& in this case).
&&
echo A
&&
Top node of parenthesis tree
echo B
echo C

main.cpp and the make_tree Algorithm

In main.cpp, there are 2 functions.

1) int main()
2) Command* make_tree(string, size_t)

Note that the make_tree() function in main.cpp is a recursive function called at first by main(). In main(), we get the user's input, clean the string so that it only contains an even number of parenthesis & brackets and does not include anything after a #. For example, if the user inputs:

“(echo hi && echo lo) && (echo lo”

then the main function will catch the uneven number of brackets error and ask for another input.

Another example user input,

“(echo hi && echo lo) # asdfajsdfh”

will be cleaned to “(echo hi && echo lo)”

After the string has been cleaned, main() will create a Command*, which is a pointer that will hold the top node of the entire tree. Command* will be set to the return value of a call to make_tree(), which takes in arguments (string, size_t) and returns Command*.

Moving on to what happens when you make a call to make_tree(string user_input, size_t start).
The first parameter, string user_input, is the user input string passed in by main. The second parameter is passed by reference and it is the position in user_input where we want to start our work. Obviously, when main calls make_tree, it will pass in 0 for the starting position.
Therefore, main's call to make_tree looks like:

```
size_t  start = 0;
Command* base = make_tree(user_input, start);
```

Once we're inside make_tree, the real work of creating a tree of commands begins. To help, I will first give a few concrete examples of how the algorithm would create a tree out of a specific user command, and then I will give the full implementation of the algorithm for the general case.

Example 1: How make_tree would create a tree without parenthesis

Let’s say that main() reads the user’s input as “echo 1 && echo 2 || echo 3; echo 4”

Let’s set up some variables to keep track of where we are in the string, and where the next connector was found. Let’s call them **start** and **found**. Also, let’s create two base class pointers (Command*) and name them **link** and **ex**. **link** points to the current top of our tree, while **ex** is something we will use as a temp to build executables (the leaves of our tree). Last, we have a string called args which will hold the latest argument and its parameters. For example, if we have “echo A && echo B”, we will first set args to “echo A” to create an executable, then set args to “echo B” and create another executable.

Note: to implement this algorithm, I used a string method called find_first_of which returns the position of the first time it finds one of the many characters you tell it to check for.

Step 1:

How to Handle Redirection and Piping

First, we must understand the behavior of redirect.

The “**>**” and “**>>**” redirection operators:

command > destination_file

- Here, the output of executing the command on the left is then stored into destination_file instead of being printed to your screen.
- If destination_file already exists, the, we will overwrite all of the contents in that file with the output of command.
- If destination_file does NOT exist, we will create a new file with the name destination_file and fill it with the output of command.
- This is different than the **>>** redirection operator, which will only append the output of command to destination_file rather than overwriting destination_file completely (overwriting means deleting everything that was already in the file).
- Note that if the command fails, clear the file and output error to the screen.
- If we’re using **>>** and the command fails, simply output an error to the screen.

The “**<**” redirection operator:

command < input_file

- Here, we will execute command on the contents of input_file.
- If input_file doesn’t already exist, we will instead print an error to the screen and return false.

Next, we see that **piping** has the following format:

output_process | input_process

- The left side’s process will run and have its output fed into the right-hand side’s process.

Notice that for all of these, we need to know what the left and right sides are at the same time. That is, we can’t just take the left side of a pipe and execute it, then use the right side of a pipe and execute it separately. This would probably be possible if we created a new file to store the output of the left side’s process in, but we should assume that no new files should be created.

First approach:

No new classes. Simply treat “echo a > input_file” as an executable.

Then

https://stackoverflow.com/questions/4812891/fork-and-pipes-in-c
http://www.cplusplus.com/forum/general/201187/
http://www.cs.loyola.edu/~jglenn/702/S2005/Examples/dup2.html

ISSUES

// make sure to go to composites and use the files in and out and redirection types right on the right child
// ISSUE ERROR: WHAT IF WE HAVE (echo a && echo b) > yo.txt, not echo a && (echo b && echo c) > yo.txt
// they are different because in the second case, the () are a right subtree, but in the first case, the () are a single tree so
// only setting right to be printed to file would neglect the left side of the tree.
// maybe give that case its own redirection_type number?

// for now, solve case where we have echo a && (echo b && echo c) > yo.txt

Need to make all current | searched be | && str[found+1] == | or else we are trying to pipe
Idea: use redirect_type to tell us whether to put left/right output

// need to add a case where after (), first found is > : (echo hi) > yo.txt