Design Document

January 29, 2018; Winter Quarter
Last Updated: March 9, 2018

Jonathan Oaks joaks001 SID:861303298

Introduction

This design is for a command shell program “rshell” written in C++. The rshell program will present a command prompt ($) which will read commands in the form:

$ [command] [arguments] [connector]

where [command] is any bash command located in /bin, [arguments] are any arguments those bash commands may take, and [connector] is one of {;, &&, ||}.

Connections allow for multiple commands to be run from one line based on the condition of the connection provided: ; will execute the next command, && will execute the next command only if the first command was successful, and || will execute the next command only if the previous command was unsuccessful.

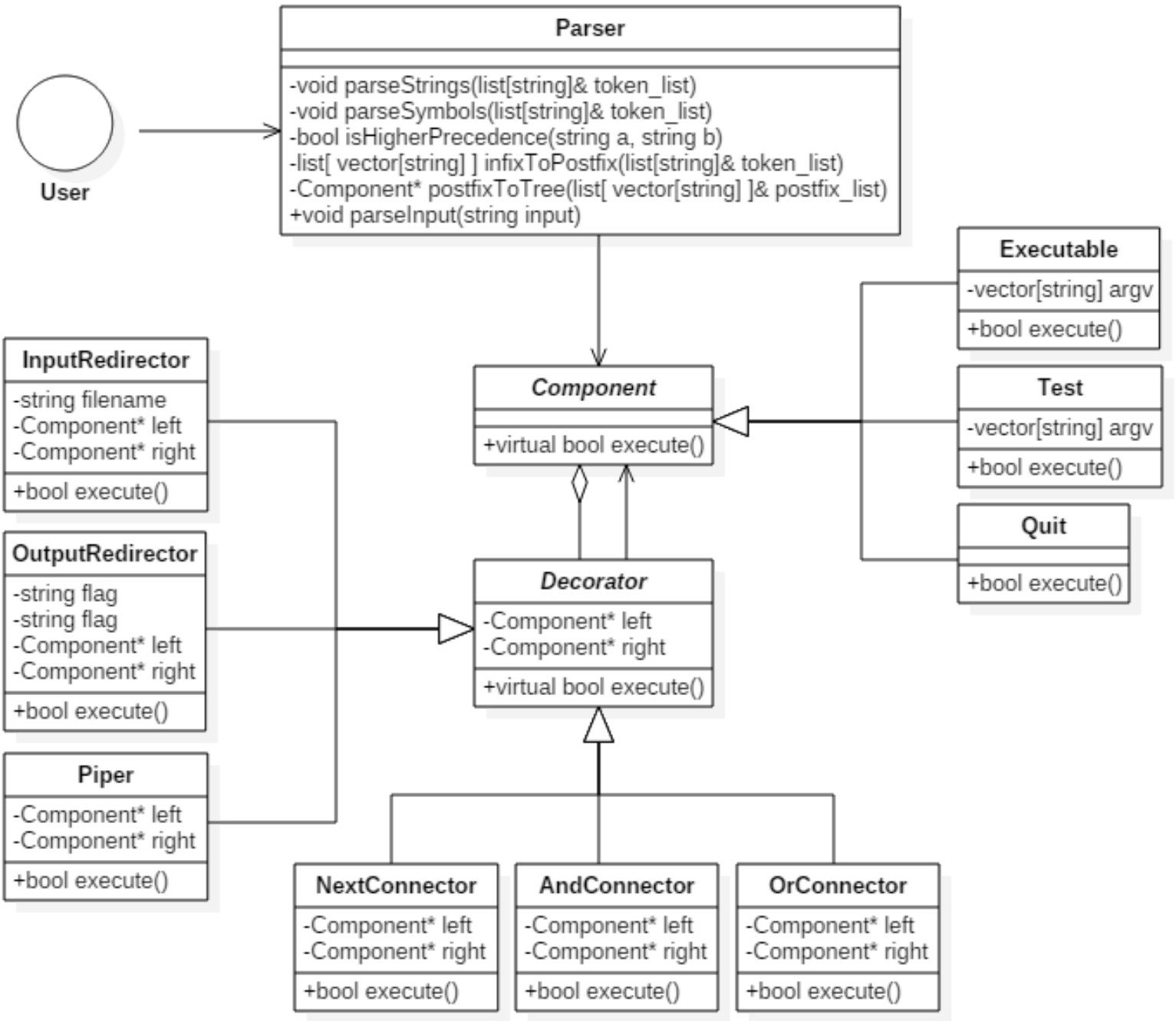
There is no limit to the amount of commands that can be chained together with connections, and there can be any combination of of connections in a single line. The program will run until the user exits rshell using a custom command.

Test commands, and their symbolic equivalent [ ] can test whether a file or directory exists and return true or false. They can also accept the flags -e, -f, -d to specify if the file/directory exists, if the file exists, and if the directory exists respectively.

Precedence operators ( ) can change the precedence of the returns of commands. There can be nested parentheses. If there are too many open parenthesis, the program will return an error message. If there are extra closed parenthesis, the program will ignore them.

Input redirection operators <, >, >>, | can override the default input/output streams. Cmd < filepath will provide the command’s input with the contents of the file. Cmd > or >> filepath will send the output of the command (or set of commands) to the file. > will create a new file at the path while >> will append to the file at the path (or create it if it doesn’t exist). Cmd | Cmd will pipe the output of the left command tree to the input of the right command.

Diagram


Parser
-void parseStrings(list[string]& token_list)
-void parseSymbols(list[string]& token_list)
-bool isHigherPrecedence(string a, string b)
-list[vector[string]] infixToPostfix(list[string]& token_list)
-Component* postfixToTree(list[vector[string]]& postfix_list)
+void parseInput(string input)
User
Executable
-vector[string] argv
+bool execute()
InputRedirector
-string filename
-Component* left
-Component* right
+bool execute()
Component
+virtual bool execute()
Test
-vector[string] argv
+bool execute()
Quit
+bool execute()
OutputRedirector
-string flag
-string flag
-Component* left
-Component* right
+bool execute()
Decorator
-Component* left
-Component* right
+virtual bool execute()
Piper
-Component* left
-Component* right
+bool execute()
NextConnector
-Component* left
-Component* right
+bool execute()
AndConnector
-Component* left
-Component* right
+bool execute()
OrConnector
-Component* left
-Component* right
+bool execute()

Classes/Class Groups

class Component

The base class for what will become the executable commands and the decorators to those commands. Has one virtual function execute() that returns a bool indicating the success of the commands its children will run.

class Executable : public Component

On construction, assigns a vector of strings to an argument vector. When execute() is called, the process is split with fork(). The child process runs the command with arguments contained in argv. The parent process waits for the child process to exit and returns the success status of the child process. Executable also has a function to return the string at an index of its argument vector.

class Test : public Component

On construction, assigns a vector of strings to an argument vector. When execute() is called, the vector is trimmed of its closing bracket if it is the last string is], as well as the first string which will be either [or “test”. The new front of the vector is saved as a flag and trimmed if it matches to one of the three available flags. The stat() function is run on the next string which is the file/directory to search for. The st_mode return buffer is examined and the result is returned depending on its relation to the flags.

class Quit : public Component

When its execute() is run, calls exit(0) from the main process and exits the program without running any other commands or returning to the main loop.

class Decorator : public Component

The base class for the three types of connectors. Contains two pointers to Components for a left node and a right node. The decorator structure indicates that the Components will be organized as an expression tree, with the Decorators acting as the branches and the Executables as the leaves. The execute() function will, in general, return the success of the child nodes.

class NextConnector : public Decorator

Representing the semicolon. When its execute() function is called, it calls its left node’s execute(), then returns the value of its right node’s execute().

class AndConnector: public Decorator

Representing the logical && operator. When its execute() function is called, it calls its left node's execute(), then if it was successful, it will run its right node's execute(). If both nodes return a success, then AndConnector will return a true, otherwise it will return a false.

class OrConnector : public Decorator

Representing the logical || operator. When its execute() function is called, it calls its left node's execute(), then if it was not successful, it calls its right node's execute(). If either node returned true, then OrConnector will return true, otherwise, it will return false.

class InputRedirector : public Decorator

Representing the < operator. When its execute() function is called, it forks the process. The child process replaces the standard input file descriptor with the file at the provided filename. It then executes its command, which will receive its input from the file. The child process exits with the appropriate status if the file doesn't exist or the command doesn't execute properly. The parent process, which has maintained its standard output file descriptor, waits for the return status of the child process and returns the boolean of its success.

class OutputRedirector : public Decorator

Representing the > and >> operators. When its execute() function is called, it forks the process. The child process replaces the standard output file descriptor with the file at the provided filename. The file is either appended to or truncated based on the flag set by > or >>. It then executes its command, which will send its output to the file. The child process exits with the appropriate status if the file doesn't exist or the command doesn't execute properly. The parent process, which has maintained its standard output file descriptor, waits for the return status of the child process and returns the boolean of its success.

class Piper : public Decorator

Representing the | operator. When its execute() function is called, it creates a pipe from the system call and forks the process, which also forks the input and output streams to the system's pipe. The child process replaces its standard output with the write-pipe and runs its left command, then exits with the appropriate status. The parent process waits for the child process to end, then creates another fork for the right command, except replacing the standard input with the read-pipe. Once the second

child process ends, the parent process, which has maintained its standard input/output file descriptors, returns with the appropriate boolean value.

class Parser

Converts the user's input string to a full fledged expression tree. It first splits the string into tokens by " and ' characters. Then it reconstructs compounded strings into a single token so that strings can have quotes within them (e.g. " a 'bc' d" or 'a "bc" d'). It then splits the tokens into more tokens separated by a space, a semicolon, or parenthesis, keeping the semicolon and parentheses tokens as individual tokens. During this step, if the comment character (#) is found at the start of a token, a comment has started and the remaining tokens are discarded as comments. The remaining tokens are converted from an infix notation (ls && ls) to a postfix notation (ls ls &&). The postfix is converted to a Component expression tree, declaring the necessary classes for each token. Finally, the root of the expression tree is executed.

Coding Strategy

The main steps of this design are:

1) Tokenizing user input to commands to be run
2) Chaining commands together
3) Executing /bin programs in parallel to the rshell

Implementing from the inside out seems to be the most prudent path.

1) Verify the syntax used by the system calls to run /bin programs, updating the specifics in the design document as necessary
2) Implement chaining using premade Commands to test functionality and edge cases.
3) Implement string tokenization and Command construction.

By implementing the Executable first, we ensure that if a Command is formatted correctly, testing it will produce the correct outcome. This reduces the possible errors encountered when implementing Command chaining; rather than wonder if the programs are executing properly or if the chains are running properly, we can be sure the error lies with the chains. Once Command chaining is implemented, the use of premade Command chains can highlight errors with certain combinations of Commands or Connectors. After both Executions and Commands are verified to work correctly, tokenizing the input string properly will have the minimal room for external errors.

If these tasks were to be completed independent of each other, the outer layers should be tested with placeholder functions that print the expected state of the program following their completion, e.g: Completing the Parser independently can be rudimentarily tested by printing out the final lists of tokens and Components. Completing the Command task independently should print the expected programs to run and arguments to pass in. Test cases could be made to demonstrate the validity of the Next, And, and Or Connectors by assigning test values to the boolean variables. Executing the /bin programs is the most inward step of this process, so it can be completed as normal. Integrating these tasks together should be seamless, but each case should still be tested for possible errors introduced during integration.

The precedence operators were already implemented during Assignment 2 as part of building the expression tree, so all that needs to be tested is the case where parenthesis are mismatched.

The test command, while have the appearance of a precedence operator, is really just another command to run. Whether it starts with square brackets or “test”, its basic functionality is the same. The return values from the stat() method can be tested individually before putting in the logic for flag parsing.

The Input/Output redirection has a higher precedence than the logic operators, so proper precedence logic has to be introduced to the parser. Once the precedence is rewritten, the Input, Output, and Pipe classes can be implemented independently.

Roadblocks

The Parser class currently holds a lot of functionality inside its only function. It’s possible each step in the parsing process could be its own private method or class that interfaces with the Parser, and it would probably be cleaner on memory that way as well. However, as it already functions as expected, I wouldn’t want to split the functionality at this point. If in the future, more functionality is required of the parser, then I would focus on decoupling the functionality at that point.

The Container class currently is not doing much. In the original design, I had it set up so that many expression trees would be pushed into the Container, delimited by the semicolon, until I realized that the semicolon could just be its own sort of connector within the expression tree. The Parser could just as easily run the final tree when it's done creating it, but to avoid giving even more functionality to the Parser and as it's still possible that future assignments could have a use for a Container type class, I'll leave it in.

The functionality of the parser, while still being in one class, has at least been moved to different functions which should help keep memory clean of temporary or unused variables as the input string transforms into a list of a vector of strings.

The Container continues to not have any practical use, however there also continues to be no reason to remove it, so it stays.

The global variable present during the Assignment 2 submission was successfully removed with the simple implementation of the Quit command.

Because () precedence has unique logic to operator precedence, I hadn't created a proper function for discerning precedence. This will have to finally be implemented.

The Container class, having been useless the entire time, can be taken out for cleanliness.