Introduction

For our final project in ISSE, we will bring together the concepts we have been learning about this semester. Introducing **Plaid Shell** ("plaidsh"): A fully featured shell that runs on Linux, written by you.

Unlike in previous assignments, I am not providing data structures or code scaffolding. I will provide only my test script, which steps plaidsh through its paces. You will need to use the various techniques and approaches we have covered in the class to meet the requirements outlined below.

Here is a sample interactive session with plaidsh. As before, user input is **highlighted**:

```
$ ./plaidsh
Welcome to Plaid Shell!
#? pwd
/home/howdy/Teaching/CMU-ISSE/Assignments/12
#? ls --color
'CMU plaid.pxd'             pipeline.c   setup_playground.sh
'ISSE Assignment 12.docx'   pipeline.h   test.psh
 Makefile                   pipeline.o   token.h
 clist.c                    plaidsh      tokenize.c
 clist.h                    plaidsh.c    tokenize.h
 clist.o                    plaidsh.o    tokenize.o
 parse.c                    ps_test     '~$SE Assignment 12.docx'
 parse.h                    ps_test.c
 parse.o                    ps_test.o
#? ./setup_playground.sh
Plaid Shell playground created...
#? cd Plaid\ Shell\ Playground
#? ls
 README  'best sitcoms.txt'  'seven dwarfs.txt'   shells.txt
#? ls *.txt
'best sitcoms.txt'  'seven dwarfs.txt'    shells.txt
#? echo $PATH
$PATH
#? author
Howdy Pierce
#? author | sed -e "s/^/Written by /"
Written by Howdy Pierce
#? grep Happy *.txt
best sitcoms.txt:Happy Days
seven dwarfs.txt:Happy
#? cat "best sitcoms.txt" | grep Seinfeld
Seinfeld
#? sed -ne "s/The Simpsons/I Love Lucy/p" < best\ sitcoms.txt > output
#? ls -l
total 20
-rw-rw---- 1 howdy howdy  64 Nov 15 01:48  README
-rw-rw---- 1 howdy howdy 142 Nov 15 01:48 'best sitcoms.txt'
-rw-rw---- 1 howdy howdy  12 Nov 15 01:48  output
```

```
-rw-rw---- 1 howdy howdy  45 Nov 15 01:48 'seven dwarfs.txt'
-rw-rw---- 1 howdy howdy  25 Nov 15 01:48  shells.txt
#? cat output
I Love Lucy
#? this is not a command
this: Command not found
Child 46881 exited with status 2
#? echo Hello > /usr/bin/cant_write
/usr/bin/cant_write: Permission denied
#? cd
#? pwd
/home/howdy
#? cd ~
#? pwd
/home/howdy
#? exit
```

Some features to note about plaidsh:

- There are five tokens:

TOK_WORD	A word is the most basic token. Generally, words are delineated by whitespace or one of the redirection tokens, but these characters can be included in a word by escaping them as described below.
TOK_QUOTED_WORD	A quoted word is like a word, but it is surrounded by double quotes. Within quoted words, whitespace and redirection tokens do not delineate word boundaries, and globbing is not performed.
TOK_LESSTHAN	A literal **<** character indicates a redirection of standard input. The following token must be of type TOK_WORD, and will indicate the filename to redirect input from.
TOK_GREATERTHAN	A literal **>** character indicates a redirection of standard output. The following token must be of type TOK_WORD, and will indicate the filename to redirect output into.
TOK_PIPE	A literal **|** character separates commands within a pipeline. The standard output of the command on the left of the pipe character is redirected into the standard input of the command on the right.

- After tokenization, words undergo *globbing*: the patterns *, ?, [], and ~ are expanded. We will use the system function **glob** to perform globbing[1]. Note that this process can change the number of words: For instance, the command `ls *.c` will be turned into `ls clist.c pipeline.c plaidsh.c` if those were the files that matched the glob pattern "*.c".

- It is not possible for a child process to affect most aspects of its parent process, and as a result certain shell functionality cannot be implemented by calling a child process. Included in this category are commands that cause the shell to terminate and commands that change the current working directory. As a result, plaidsh has the following built-in commands:

`exit`	Terminates plaidsh
`quit`	Terminates plaidsh

[1] See man glob for details. We will use the flag GLOB_TILDE_CHECK.

`author`	Prints the name of the author of this plaidsh—that is, YOUR name—to stdout. Note that the output of this command can be redirected.
`cd [directory]`	Changes the current working directory to the named directory, which must exist. If directory is omitted, changes the current working directory to $HOME[2].
`pwd`	Prints the current working directory[3]. Note that the output of this command can be redirected.

- If the parsed argv[0] does not match any of the built-in functions, plaidsh will create a child process using the fork() system call, and then will use the execvp() system call to execute the specified command line. The parent process will wait for the child to terminate using either the waitpid() or the wait() system call.

- In the event a child terminates with a non-zero exit status, the message "Child <pid> exited with status <status>" is printed to stderr. When a child exits with a zero status, no message is printed.

- In general, child processes inherit the parent's stdin, stdout, and stderr. As with other Linux shells, however, it is possible to redirect the child's stdin and stdout (but not stderr) using the <, >, or | tokens.

- A word can be any length; there can be any number of arguments to a command; and there can be any number of commands within a pipeline. Each pipeline can have at most one input redirection and at most one output redirection.

- Plaidsh is built upon readline, just like ExprWhizz was. You should have interactive editing with the left and right arrow keys; the ability to scroll through history using the up and down error keys; and basic tab completion of filenames[4].

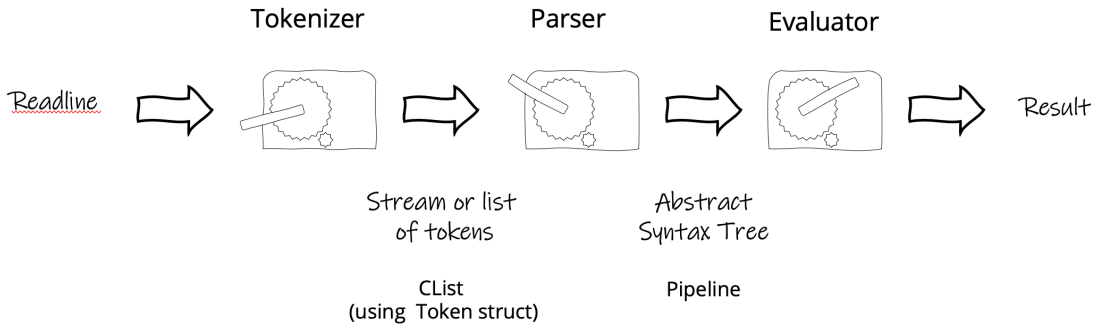
[2] You can change the current working directory using the chdir() call, and you can access an environment variable using the getenv() call.

[3] You can get the current working directory using the getcwd() call. Note it would not be strictly necessary to implement pwd as a built-in; there is an executable `/usr/bin/pwd`. However, it's easy to do, and most shells have pwd as a built-in.

[4] Do not worry about making tab completion work correctly if there are spaces in a filename.

High Level Design

The design of plaidsh is similar to that of ExprWhizz, as shown in the following diagram:



Rather than use ExprTree, we will need to create a new abstract syntax tree, which I called a "pipeline". This AST is simpler than ExprTree. Likewise, the plaidsh parser is simpler, and does not need to use recursion.

The Evaluator module is slightly more involved than what we did for ExprWhizz. You will need to generalize your fork/exec code from Assignment 7.

Prompt

The plaidsh prompt is "`#?` " – that is, a hash sign, a question mark, and a space. The test script relies on finding this pattern and may not work correctly without it.

My own code uses terminal escape sequences to put the prompt in a bold color[5]. This is not a requirement; you are free to use other colors, or none at all.

Tokenization Details

A WORD ends with any of the following characters:

- An unescaped **<**, **>**, or **|**; each of these is a token on its own
- An unescaped double quote; this starts a QUOTED_WORD
- Any unescaped whitespace
- The end of the input line

QUOTED_WORDs are simpler: A quoted word begins with a double quote, and continues until the next unescaped double quote.

[5] Specifically, my code uses the string "\e[01;31m#?\e[00;39m " for the prompt. See https://en.wikipedia.org/wiki/ANSI_escape_code for details as to how I constructed this string.

Within WORDs and QUOTED_WORDs, the following escape sequences are recognized:

Escape sequence	**Converted to**
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\"`	" (double quote)
`\\`	`\`
`\␣` (space)	␣ (space)
`|`	`|`
`\<`	`<`
`\>`	`>`

The following examples explain the tokenization rules in greater detail:

User types	Resulting tokens
`echo a b`	WORD “echo”; WORD “a”; WORD “b”
`echo a\ b`	WORD “echo”; WORD “a b”
`echo "a b"`	WORD “echo”; QUOTED_WORD “a b”
`echo a\\ b`	WORD “echo”; WORD “a\”; WORD “b”
`echo hello|grep "ell"`	WORD “echo”; WORD “hello”; PIPE; WORD “grep”; QUOTED_WORD “ell”
`echo hello\\\|grep "ell"`	WORD “echo”; WORD “hello|grep”; QUOTED_WORD “ell”
`echo boo >out_file`	WORD “echo”; WORD “boo”; GREATERTHAN; WORD “out_file”
`echo"boo">out_file`	WORD “echo”; QUOTED_WORD “boo”; GREATERTHAN; WORD “out_file”
`(no input)`	(no tokens)
`echo "hello | grep"`	WORD “echo”; QUOTED_WORD “hello | grep”
`echo a"b c"`	WORD “echo”; WORD “a”; QUOTED_WORD “b c”

Abstract Syntax Tree

For plaidsh, I created an abstract syntax tree I called a *pipeline.* It has the following features:

- The caller can set or get an input file for the whole pipeline; it is set to stdin by default
- The caller can set or get an output file for the whole pipeline; it is set to stdin by default
- The caller can add one or more *commands*, which are understood to be piped together (that is, the output of the first command is piped to the input of the second command, and so on)
- Each command can have an arbitrary number of arguments, which can be added one-by-one

Implementation Notes

I recommend that you approach this project as follows:

1. Create a file plaidsh.c, which contains main(); within main, implement a while(1) loop using readline; to start, simply print out what the user types. Get history via the up and down arrows working.
2. Create a Makefile; be certain to set the following flags to gcc:
   ```
   -Wall -Werror -g -fsanitize=address
   ```
   You should have at least a make all and make clean rule.
3. Work on basic tokenization: Create the files token.h, tokenize.c and tokenize.h, and some automated tests for tokenization. Get to where you can pass all the tokenization tests. You should use at least the examples above. Then tie your tokenization into the main loop, so that you can type interactively into plaidsh and examine the resulting stream of tokens.
4. Create pipeline.h and pipeline.c files, and create automated tests for these.
5. Create a parser that accepts the list of tokens as an input, and outputs an AST (the Pipeline data structure). Write some automated tests for your parser and ensure that your code passes the tests. Your parser will be simpler than the recursive descent parser we implemented for ExprWhizz; mine is a simple loop with no recursion.
6. Create a module that knows how to take a Pipeline structure and do the fork/exec step for children. Get this working with external commands first, and then add the builtin internal comments (**quit**/**exit**, **author**, **cd**, and **pwd**).
7. At this point you should have an interactive version of plaidsh that you can type into. Congratulations!
8. Figure out how to add support for globbing. This should come during the parsing step, while the Pipeline structure is being built up.

Future Directions

The version of plaidsh described here does most of the functions that shells like bash or tcsh shell do. If you are interested, the are some obvious next steps you could take to extend plaidsh to be even more useful:

- There is no support for reading either environment variables or shell variables, and you cannot set either kind of variable, either. It would be straightforward to create a new **$** token, which introduces variable substitution of the form $varname. You could also create a new **setenv** builtin function, which would allow users to set environment variables. Even better would be to allow the setting of both shell variables and environment variables using varname=value syntax, though this would require recognizing **=** as a token.

- It would be really cool to support $(( )) style math – you already have the code to do this in ExpressionWhizz!

- A proper shell should have a builtin **history** command, and "!!" and "!-2" style abbreviations.
- By default, readline provides a form of filename completion via the <tab> key, although it does not perform proper escaping if the file names contain spaces. See https://thoughtbot.com/blog/tab-completion-in-gnu-readline for pointers on how to improve the default tab completion.

Testing

Hopefully by this point in the class, you have a good sense for how to write your own tests, and an appreciation of their importance.

I am not providing any test scaffolding for you this time, but that doesn't mean you shouldn't develop it! I expect to see test code that tests your CList and Token structures; your TOK_next_consume functions; your TOK_tokenize_input function; your Pipeline data structure; and your parsing code. For plaidsh, my own test framework runs to 560 lines of C code, and you should aim for a similar amount.

For integration tests, I have provided the script `plaidsh_test.py`, which depends internally on the helper script `setup_playground.sh`. In addition, my `plaidsh` executable is installed on ScottyOne at `/var/local/isse-12`, so if in doubt you can run test commands against my executable.

Scottycheck is not available for this assignment.

Grading and Submission

To submit your code, you should create a private GitHub repo that contains your code and a README.md file; the README should (at minimum) document how to build your code. Prior to the submission deadline, you should then submit the URL of the GitHub repo on Canvas. We will compute a grade based on the state of your repo as of the submission deadline, so feel free to make changes to it up to that point.

Points will be awarded as follows:

Points	Item
5	Compilation: Does your code compile without warnings? Do you have a Makefile? Does it include at least "make all" and "make clean" rules? Do your CFLAGS include the options specified earlier in this document?
10	Code legibility: Is your code indented reasonably and are variable names well selected? Are there appropriate comments? Does the code use #defines appropriately? Subjectively, does this look like good, professional code?

Points	Item
15	Your tests: Do you have a substantial automated test suite, written in C, that tests the internals of your tokenization, your parsing, and so forth?
65	Correctness: As reported by plaidsh_test.py (which includes tests for memory leaks)
5	Bonus points: 5 extra points if you fully pass all automated tests.

Good luck and happy coding!