A *shell* is a command-interpreter used widely on Unix-based systems. Most shells receive character-based input, typically read from a keyboard or a textfile, parse their input into a token sequence forming the syntax recognised by the shell (command names, command arguments, and control-flow delimiters), and then attempt to execute the token sequence. The shell used by most CSSE students under macOS and Linux is named *bash* (the Bourne Again Shell); a former CSSE graduate is the author of a very popular book describing the Bash shell.

The goal of this project is to implement a program named *myshell* supporting a *small subset* of the features of a standard Unix-based shell. Successful completion of the project will develop your understanding of some advanced features of the C99 programming language, and your understanding of the role of a number of system-calls responsible for process invocation and monitoring, and input and output management using file and pipe descriptors.

Suggested steps

A number of C99 source code files are provided as a starting skeleton for the project. The most challenging-to-write part of the project, the command parser, has been written for you, and should be used without modification (and need not be understood). The parser returns a pointer to a binary tree of shell commands which your code will traverse and execute.

In summary, the function *parse_shellcmd(FILE *fp)* accepts a *FILE* pointer as its only input, and attempts to read and parse its input. If successful, function *parse_shellcmd()* returns a pointer to a user-defined datatype named *SHELLCMD*, a self-referential structure holding all information necessary to execute the requested command-sequence. Your role is to execute the commands in the structure by (implementing and) calling your own function named *execute_shellcmd()*.

The project only requires implementation of a *small subset* of a traditional Unix shell. However, that subset will be quite faithful to standard shells. For this reason, this project description does not specify all details in great detail. If unsure about the role and action of shell features listed in these steps, you should experiment with the same feature in your standard shell, *bash*.

The approach taken to develop the *sample solution* has been broken into 9 independent steps, presented below. There's no requirement for you to follow these steps, but they are described here to provide an obvious pathway to a solution, and to explain the project's required features.

Step 0. Develop a *Makefile*

Develop a *Makefile*, employing variable definitions and automatic variables, to compile and link the project's files.

Step 1. Execute simple *external* commands

1. execute commands without arguments, such as */bin/ls*
2. execute commands with arguments, such as */usr/bin/cal -y*

Helpful C99 and POSIX functions: *fork(), execv(), exit(), wait()*.
DO NOT use the functions: *system()*, *popen()*, *execlp()*, or *execvp()* anywhere in your project.

Step 2. Search path

The global variable *PATH* points to a character string which is interpreted as a colon separated list of directory names.

If the user enters a command name which does not contain a '/' character, then the members of the search path list are considered as directories from which to attempt to execute the required command. Once the required command is found and executed, the search terminates.

Helpful C99 and POSIX functions: *strchr()*.

Step 3. Execute *internal* commands

Internal commands are performed by *myshell*, itself, rather than by external commands. *myshell* does not create a new process, with *fork()*, before running an internal command. Three internal commands are to be supported:

1. *exit* - terminate *myshell* by calling the function *exit()*. When an additional argument is provided, it should be interpreted as the numeric exit-status. When requested without any arguments, the exit-status of the recently executed command should be used.
2. *cd* - change from the current working (default) directory to the directory specified as the first argument to the *cd* command. If the command *cd* is given without arguments, then the variable *HOME* should be used as the new directory.
   As with the role of *PATH* for command execution, the variable *CDPATH* points to a character string which is interpreted as a colon separated list of directory names. If the user enters a directory-name which does not

contain a '/' character, then the members of *CDPATH* are considered as directories to locate the required directory. (Note that due to the use of a simple parser, the command sequence <i>cd&nbsp;somewhere \&nbsp;echo&nbsp;*</i> will list incorrect filenames - can you think why? You do not need to address this problem.) <p></p>

3. *time* - the 'following' command's execution time should be reported in milliseconds (e.g. 84msec) to *myshell*'s *stderr* stream.

Helpful C99 and POSIX functions: *exit(), chdir(), gettimeofday()*.

Step 4. Sequential execution

There are three forms of sequential execution, and thus three subparts to this step. The three forms differ in the action they take when a command *fails*. By convention, a command which fails will return a non-zero exit status.

1. A sequence of commands separated by the ";" token requires *myshell* to execute each command sequentially, waiting until each command has finished before beginning the next. In this form, *myshell* should continue to the next command regardless of the result of earlier commands. The sequence *ls ; date monday ; ps* would execute *ls*, then on its completion execute *date monday* (which will fail), then execute *ps* (even though the previous command failed). The exit value of sequential execution is the exit status of the command sequence to the right of the ";" token.
2. If the commands are separated by the token "&&" then *myshell* should continue with further commands only if the command sequence to the left of the token returns an exit value indicating success. For the command sequence *a && b*, *myshell* should wait for command *a* to terminate, test its return value and execute *b* if and

only if *a*'s return value indicates success. The exit value of an "&&" command sequence is the exit status of the last command executed.

3 If the token "||" is used to separate commands then *myshell* should continue to the next command only if the first command sequence returns an exit value indicating failure. This is the converse of part 2. For the command sequence *a* || *b*, *myshell* should wait for command *a* to terminate, test its return value and execute *b* if and only if *a*'s return value indicates failure. The exit value of a "||" command sequence is the exit status of the last command executed.

Step 5. Subshell execution

The sequence *(commands)* should cause *commands* to be executed in a subshell. A subshell is created by forking a copy of *myshell* to execute the commands. The exit value for a subshell is the exit value of the command sequence inside the parentheses.

Step 6. stdin and stdout file redirection

Commands may redirect their standard input and standard output from and to files, respectively, before the command is executed.

1 The sequence *command* < *infile* requests that the command use *infile* as its standard input.

2 The sequence *command* > *outfile* requests that the command use *outfile* as its standard output. If the file *outfile* does not exist then it is created. If it does exist it is truncated to zero then rewritten.

3 The sequence *command* >> *outfile* requests that the command appends its standard output to the file *outfile*. If *outfile* does not exist it is created.

NOTE: in all of the above cases *command* could be a subshell. It is valid to have both input and output redirection for the same command. Thus the sequence

(sort ; ps) < junk1 >> junk2 would take input from the file *junk1* and append output to the file *junk2*.

Helpful C99 and POSIX functions: *open(), close(), dup2()*.

Step 7. Pipelines

The sequence *command1 | command2* requests that the *stdout* of *command1* be presented as *stdin* to *command2*. By default, the *stderr* output of *command1* is not redirected and appears at its default location (typically the terminal). With reference to this example, the *myshell* parser will not permit the *stdout* of *command1* to be redirected to a file (with >), nor the *stdin* of *command2* to be received from a file (with <). Note that the sequence *command1 | command2 | command3* requests that a pipeline of two different pipes be established.

Helpful C99 and POSIX functions: *pipe(), dup2()*.

Step 8. Shell scripts

Once an executable file name has been found (possibly using the search path) the standard function *execv()* can be called to try to execute it. If *execv()* fails, you should assume that the file is a *shell script* (a text file) containing more *myshell* commands. In this case you should execute another copy of *myshell* to read its input from the shell script. No specific filename extension is required.

Helpful C99 and POSIX functions: *access(), fopen(), fclose()*.

Step 9. Background execution

The token "&" causes the preceding command to be executed without *myshell* waiting for it to finish (asynchronously). Thus the sequence *ls ; ps & date* should start the command sequence *ls*, once this has completed start *ps* and immediately proceed to the command *date*. The exit value of background execution is success unless the fork call fails, in which case the exit value should indicate failure.

Commands placed into the background may take a long time to execute and, thus, we wish to know when they have completed. The parent is informed of the termination of child processes using *asynchronous signal-passing*. When so informed, *myshell* should report which background process has terminated.

Similarly, when *myshell* exits, it should first terminate all of its still-running background processes, and wait for them to finish.

Helpful C99 and POSIX functions: *signal()* and *kill()*.

Starting files

The amount of code to be written for this project is similar to that of the 1st project, although less needs to be designed "from scratch" because you're extending an (incomplete) code skeleton.

An **executable sample solution** will be available soon.

Start your project by downloading, reading, and understanding the files in the archive myshell.zip.

- myshell.h - provides the definition of *myshell*'s user-defined datatypes and the declaration of global variables.
- myshell.c - provides the *main()* function, and calls the *parse_shellcmd()* function.
- globals.c - defines global variables, and the helpful *print_shellcmd0()* function.

execute.c - where you define your *execute_shellcmd()* function.

parser.c - defines the *parse_shellcmd()* and *free_shellcmd()* functions, which should be used without modification (and need not be understood).

You may modify any file or add any additional files to your project.

Program requirements

1. Your project, and its executable program, **must** be named *myshell*.
2. Your project **must** be developed using multiple C99 source files and **must** employ a *Makefile*, employing variable definitions and automatic variables, to compile and link the project's files.
3. If any error is detected during its execution, your project **must** use *fprintf(stderr, ...)* or *perror()* (as appropriate) to print an error message.
4. Your project **must** employ sound programming practices, including the use of meaningful comments, well chosen identifier names, appropriate choice of basic data-structures and data-types, and appropriate choice of control-flow constructs.