Lab3
Mini Shell

Introduction:

The main goal is to build a mini shell and execute some simple shell commands, some sources is being provided so you don't need to start from scratch.

First Part: Understand the source code provided

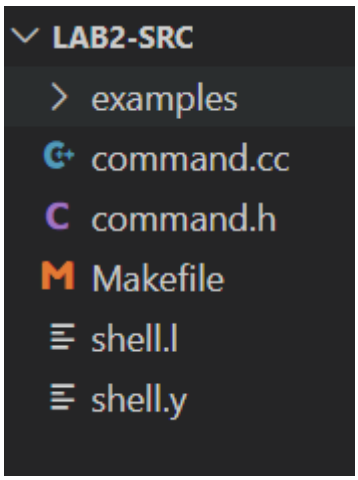
Mainly we use Lex and Yacc to build the scanner and parser for your shell so you do not have to implement a parser yourself to parse the user's input. Please take some time to learn Lex and Yacc before starting

Here are some tutorials on Lex and Yacc.
Also, watch this video!

Download the lab3-src-tar.gz file that contains all what you need in your machine and do the following command that will unpack the files for you

```
tar xvfz lab3-src.tar.gz
```

Let's give you overview about the project arcticture before doing your first run.



- **examples** folder which have some code snippit that may help you while development
- **Commands.cc** and **command.h** is where you will put most of the C code
- **Makefile** no need to do any changes there
- **shell.l** and **shell.y** is lex and yacc configuration files

Let's start to run:

Build the shell program by typing :

```
make
```

To run it type:

```
./shell
```

Then type commands like

```
ls -al

ls -al aaa bbb > out
```

Check the output printed

Try to understand how the program works. First read the **Makefile** to learn how the program is built. The file **command.h** implements the data structure that represents a shell command. The struct ***SimpleCommand*** implements the list of arguments of a simple command. Usually a shell command can be represented by only one ***SimpleCommand***. However, when pipes are used, a command will consist of more than one ***SimpleCommand***. The struct *Command* represents a list of *SimpleCommand* structs. Other fields that the *Command* struct has are ***_outFile***, ***_inputFile***, and ***_errFile*** that represent input, output, and error redirection.

Currently the shell program implements a very simple grammar:

```
cmd [arg]* [> filename]
```

You will have to modify shell.y to implement a more complex grammar

```
cmd  [arg]*   [ | cmd [arg]* ]*    [ [> filename]
                                     [< filename]
                                     [>> filename] ]* [&]
```

Insert the necessary actions in shell.y to fill in the *Command* struct. Make sure that the *Command* struct is printed correctly.

Run your program against the following commands:

```
ls
ls -al
ls -al aaa bbb cc
ls -al aaa bbb cc > outfile
ls -al aaa bbb cc >> outfile
cat file | grep text
ls | cat | grep > out < inp
ls aaaa | grep cccc | grep jjjj ssss dfdffd
httpd &
```

Second part: Process Creation, Execution, File Redirection, Pipes, and Background

Starting from the command table produced in Part 1, in this part you will execute the simple commands, do the file redirection, piping and if necessary wait for the commands to end.

1. For every simple command create a new process using *fork()* and call *execvp()* to execute the corresponding executable. If the _bakground flag in the Command struct is not set then your shell has to wait for the last simple command to finish using *waitpid()*. Check the manual pages of *fork()*, *execvp()*, and *waitpid()*. Also there is an example file that executes processes and does redirection in cat_grep.cc. After this part is done you have to be able to execute commands like:

```
ls -al

ls -al /etc &
```

2. Now do the file redirection. If any of the input/output/error is different than 0 in the Command struct, then create the files, and use *dup2()* to redirect file descriptors 0, 1, or 2 to the new files. See the example ls_output.cc to see how to do redirection. After this part you have to be able to execute commands like:

```
ls -al > out
ls -al >> out

cat out

cat < out

cat < out > out2

cat out2
ls /tt >>& out2
```

3. Now do the pipes. Use the call *pipe()* to create pipes that will interconnect the output of one simple command to the input of the next simple command. use *dup2()* to do the redirection. See the example cat_grep.cc to see how to construct pipes and do redirection. After this part you have to be able to execute commands like:

```
ls -al | grep command

ls -al | grep command | grep command.o

ls -al | grep command

ls -al | grep command | grep command.o > out

cat out
```

Third part: Control-C ,Exit, Change Directory, Process creation log file

1. Your shell has to ignore ctrl-c. When ctrl-c is typed, a signal SIGINT is generated that kills the program.
2. You will also have to implement also an internal command called *exit* that will exit the shell when you type it. Remember that the *exit* command has to be executed by the shell itself without forking another process.

```
myshell> exit

         Good bye!!

csh>
```

3. Implement the *cd [dir]* command. This command changes the current directory to *dir*. When *dir* is not specified, the current directory is changed to the home directory. Check "man 2 chdir".
4. Extend lex to support any character in the arguments that is not a special character such as "&", ">", "<", "|" etc. Also, your shell should allow no spaces between "|", ">" etc. For example, "ls|grep a" without spaces after "ls" and before "grep" should work.
5. It's required to create a log file that contains Logs when every child is terminated you can use SIGCHLD signal to do so.

Bouns part:

1. do the wildcarding. The wildcarding will work in the same way that it works in shells like csh. The "*" character matches 0 or more nonspace characters. The "?" character matches one nonspace character. The shell will expand the wildcards to the file names that match the wildcard where each matched file name will be an argument.

```
echo *          // Prints all the files in the current directory

echo *.cc       // Prints all the files in the current
                // directory that end with cc

echo c*.cc

echo M*f*

echo /tmp/*     // Prints all the files in the tmp directory

echo /*t*/*

echo /dev/*
```