Here is something that may be helpful for your part3 of the Lab3. I skip part2 since almost all the contents for part2 are in the class notes.

Part3 contains many parts. I will follow the order they come out in the test-shell, so that you can test them one by one using the test cases. Also we will use the test-shell to grade your code, except that we may make some small changes.

After you finish part2, you will be able to pass the first 9 test cases.

So I start from test10. When reading each parts, please keep two points in mind for your implementation:

(1) Where to modify your existing code?

(2) How?

Test10: Zombie processes

In order to handle Zombie process, you need to add a signal handler for the SIGCHLD signal, which is sent from the child process to the parent process telling that is has finished its job and askes the parent process to remove its entry from the process table.

As illustrated in the class notes in page 209, you need to use the sigaction() function call as follows

(sigset() will give the parser a "-1", which will make the whole program quit)

```
struct sigaction signalAction;
signalAction.sa_handler = killzombie;
sigemptyset(&signalAction.sa_mask);
signalAction.sa_flags = SA_RESTART;

int error = sigaction(SIGCHLD, &signalAction, NULL );
if ( error )
{
        perror( "sigaction" );
        exit( -1 );
}
```

This piece of code needs to be added in the main function before yyparse() is called.

In your killzombie function, you need to wait until the child process terminate. In the handout and FAQ there is one way of implementation that you use the wait3() call by

wait3(0,0,NULL);

However, some students report that they always have 1 zombie process left. So their test10 results looks like

Test10 failed. Zombie processes still around (0, 1).

This is mainly because the last child process called wait3 but still had not returned from the waiting.

Here is another way of waiting the child process to quit.

while(waitpid(-1, NULL, WNOHANG) > 0); // if wait3 doesn't work, try this one

This way uses busy waiting until the child process's quit.

Test11: Enviroment: Set variable

You need to implement the builtin command *setenv A B.* This command sets the environment variable *A* to be *B.* Check man pages for function putenv().

For where to implement the setenv, it needs to be added before you call the fork(). So before you call the fork process, you need such a check of the commands.

```
if ( !strcmp( _simpleCommands[i]->_arguments[0], "setenv" ) )

{

// add your code to set the environment variable

}
```

Make sure that "fork" of child process will not run after you run this command, because you don't want to fork another process and set the environment.

Test12: Enviroment: Redirect printenv

See the class notes of page 207, remember to declare the extern global variable char ** environ by

extern char ** environ;

Notice, here, the environ char ** in fact is a char* array. Each element in the array is a char * points to some string. Check the man pages for environ.

You need to implement this “printenv” in the child process for the reason that we will redirect the output of printenv to another file.

Test13: Enviroment: replace variable

Just the same as set variable. If you did the test11 correctly, you would have no trouble to pass this test.

Test14: Enviroment: delete variable

In order to delete the variable, you need to implement the builtin command *unsetenv A*. This command removes environment variable *A* from the environment.

The place to implement the unsetenv is the same as the setenv. But the way is different. There will not be a convenient function call like putenv() for you. This time you need to go through the environ array, find the variable you want to delete (strstr() is suggested) and then remove it from the variable array.

How to remove it from the array?

The environment variable array looks like

environ[0] environ[1] environ[2] environ[3] environ[4] environ[5]

| | | | | |

“bbb=222” “ccc=333” “ddd=444” “eee=555” “fff=666” “ggg=777”

After you insert “aaa=111”, it will become

environ[0] environ[1] environ[2] environ[3] environ[4] environ[5] environ[6]

| | | | | | |

“aaa=111” “bbb=222” “ccc=333” “ddd=444” “eee=555” “fff=666” “ggg=777”

The new variable is inserted at the beginning of the array.

So in order to delete “aaa”, you need to make it change back to what it looks like before.

Test15 and 16: Enviroment: Variable expansion

You will implement environment variable expansion. When a string of the form *${var}* appears in an argument, it will be expanded to the value that corresponds to the variable *var* in the environment table.

Where to implement?

I suggest you to implement the variable expansion in the SimpleCommand::insertArgument(char * argument) function.

How the implement?

When you meet an argument, before you inserting it into the “_argument” array, you need to check whether there is a *${var}* inside it. The checking method involves the regular expression, like something in the regular.cc.

So this time you need to check whether there is a “${var}” sub-string in the argument you insert. If yes, then change that “${var}” to the value of the “var”, which is got by getenv() function call. The regular expression used to recognize the sub-string will be something like

"^.*${[^}][^}]*}.*$". // start + some_string + “$”character + “{”character + any_string_except_the_“}”_character + “}”character + some_string + end

So you codes may be something like

// insert_string is the argument you want to insert

char * buffer = "^.*${[^}][^}]*}.*$";

char * expbuf = compile(buffer, 0, 0);

```
while(advance(inserted_string, expbuf))
{
        // find the place of “var”, get the string “var”
        // find the value of “var” and then change the inserted_string
}
```

Test17 & 18: Parsing: words and special characters

For this part you need to modify the shell.l to let the lexer realize that

“str1 str2 str3” is only one argument instead of 3 argument.

So if “str1 str2 str3” met, only 1 WORD should be returned.

How to do this?

For example, the quotes:

\"[^\n\"]*\" can be used in shell.l to express the regular expression of “string_without_newline”. Remember to remove the quotes.

So there needs to be an item like:

```
\"[^\n\"]*\"      {
                  // do some operation to the yytext and yylval.string_val to remove the quotes
                  return WORD;
        }
```

Same to the escape character. But the regular expression is more complex and the operation to modify the yytext requires more work.

Also modify your regular expression for the most common WORD so that “ls>out” works the same as “ls > out”.

Test 102 to 116

The wildcards and the sub-directory part. See the class notes from page 189 to 205.

A small bug in the class notes:

At page 205, “close(d);” should be placed outside the while loop.

Also the codes in the class notes have some problems in the top directory cases like “/d*/*”. It’s hard to tell where the bug is because there are many ways to fix it in many different places. Just make sure you make your codes pass the test cases.

Notice: This part counts many points and requires much work. Also it is hard to debug. I strongly suggest gdb for debugging.

Use gdb like “gdb shell”.

Then set break point like “break some_file : some_line”.

Run the shell by typing “r”.

Type the commands like in the shell.

Arrive the break point and print the debug information.

Test201: test ctrl-c

Ctrl-C is not hard to deal with. Similarly to handle the SIGCHLD signal for zombie process, it just changes the signal to SIGINT. But, make sure you implement the “exit” command first before you do the ctrl-C; otherwise your shell has no way to exit.

Test202: Robustness

Nothing to say. If you did all the things before correctly, there should be no problem. Good luck.

Test203: test subshell

There is a lot of information in the handout about how to implement this. You need to modify the lexer so that it can recognize the subshell expression and then make your shell fork a child process to run the subshell. After that you will let return the output of the subshell back to the lexer using the yy_unput(int c).

There is more information in the FAQ of Lab3. I just copy it here.

Some additional details on the implementation of the subshell...

1. As I explained in class, do not use temporal files to implement the subshell. The implementation using temporal files was already done last semester and I want it to be different this semester. Use two pipes to communicate the parent and the child. Using a temporal file will not give you any credit.

2. To make sure that the subshell process finishes, make the parent write an extra "exit\n" command to the subshell process.

3. The parent should read character by character from the pipe and fill in the output buffer instead of trying to read the entire buffer length. The read command will return -1 when the eof is reached. The parent should close the side of the pipe it is not using before reading the output.

4. Make sure that you don't overflow the buffer array. Enlarge it as needed.

5. The function in lex that returns characters to the lex input buffer is called:

    int yy_unput(int c)

This function returns one character back to lex. Remember to unput the characters in reverse order. This is not the best solution since the number of characters you may unput has a limit. A better implementation is to redefine the int input(void) macro in shell.l. See the c file generated by lex.

Test204: test tilde expansion

When you insert the argument, after the variable expansion, you need to add the tilde expansion also. Just check whether the first character of your argument is “~” and then change the some part of the arguments.

In order to get the information of other user’s home directory (not your home directory), you need to use the getpwnam(char *) function call. This function will return a struct called passwd.

The pw_dir field of that struct is char * containing the string you want. Man it to get more information.

Also finally, remember to edit the mode, implement the “cd” and the history which is not in the test cases. The handout has much information about it. If you have finished them ALL, there is still an “Extra Credit”….