COSC 3750
Linux Programming
Homework 8, wyshell

# 1 Basic info

This is the beginning of a shell. You will create a program that prints a prompt and parses the command line input from a user. Other than printing output, this program will **not** perform any other actions. We will start with the simple and work up. The idea is that if the assignment is already broken down into small pieces, you will not be tempted to try to implement it all at once.

If this the shell allocates memory that memory MUST be deallocated **before** whichever function the allocation occured in returns.

Your final shell will only provide a small portion of the functionality of "real" shells. Eventually we will provide for

- command execution
- input redirection
- output redirection
- diagnostic output redirection
- pipes

We will not provide any of the fancy command line editing, variables, shell versions of system utilities, or functions.

If you have any questions, contact me in person, in class, or via email.

You **have** to put in some **effort** on this. Each part will build on the previous part. That means you cannot be late, at least not much.

# 2 Syntax

This is the syntax or "language" that this shell will process/recognize. Some of the characters used here are used only to specify the grammar and are not part of the actual language.

Our command line syntax will be fairly simple. In the following, braces ({}) are used for grouping, not as part of the actual syntax. The character ∨ means "or". Square brackets ([]) are used to enclose optional items. An asterisk (*) following an item means "0 or more", a plus (+) means "1 or more". None of these are literally used in the input but are only there as elements of the grammar definition.

The symbols with special meaning to the shell are:

```
| ; > >> 2> 2>> 2>&1 < " ’ &
```

The grammar is:

| | | |
|---|---|---|
| command | → | words [words]* [redir ∨ err_redir]* [&] `(see notes 1 and 5)` |
| redir | → | {input_redir ∨ output_redir ∨ output_append} |
| err_redir | → | {err_redir ∨ err_append ∨ err_to_output} |
| pipe `(note 2)` | → | command pipe_char command [pipe_char command]* |
| input_redir | → | `<` filename `(see note 3)` |
| output_redir | → | `>` filename |
| output_append | → | `>>` filename |
| err_redir | → | `2>` filename |
| err_append | → | `2>>` filename |
| err_to_output | → | `2>&1` |
| commands | → | command [**;** command]* |
| pipe_char | → | `\|` |
| filename | → | words |
| words | → | {quoted_word ∨ word} |
| quoted_word | → | dq {ws* {word ∨ sq}$^+$}* ws* dq |
| quoted_word | → | sq {ws* {word ∨ dq}$^+$}* ws* sq |
| word | → | character$^+$ |
| character[4] | → | [ ^ > < & \| ; " ’ ] |
| dq | → | `"` |
| sq | → | `’` |
| ws | → | `[ \t]` |

Notes:

1. There can be at most one output redirection, one error redirection and one input redirection per command.

2. That means the left side of a pipe must not also redirect output and the right size of a pipe must not also redirect input.

3. The order of redirection is important. The command line

```
command 2>&1 > filename
```

redirects *command*'s standard output to *filename* but its diagnostic output to standard output. On the other hand

```
command > filename 2>&1
```

redirects *command*'s standard output to *filename* and its diagnostic output will also go to *filename*, because that is where the standard output is already redirected.

4. The character class, indicated by characters enclosed in square brackets, is a short hand for the OR of the enclosed characters. That is `[abcd]` means {a $\vee$ b $\vee$ c $\vee$ d}. A dash between characters indicates a *range*. The previous class could have been written `[a-d]`. If the first character in the character class is the caret, ^, then the meaning is inverted, that is everything BUT these characters.

5. The ampersand, &, indicates that a command is to be run in the background. If there are multiple commands separated by pipes, there can only be one ampersand and that must be at the end of the entire command line.

# 3 What you will need

You will need the files *wyscanner.c* and *wyscanner.h*. DO NOT MODIFY THESE FILES. The function that parses the input string is "int parse_line(const char *line)". The first call requires a null-terminated string (*line*) from the user. It will return an integer that indicates what element of the grammar each token represents. Subsequent calls, that process the same string, will take NULL as an argument instead of the string. This is similar to the *strtok(3)* function.

You will need to include "wyscanner.h" in your program and add wyscanner.c to the compilation.

```
gcc -ggdb -Wall wyshell.c wyscanner.c -o wyshell
```

Please look in *wyscanner.h* for the return values from *parse_line()*. You will also see **extern** declarations for the function *parse_line()*, the buffer *lexeme*, and the character *error_char*. *lexeme* will contain a valid string if the return value is WORD, otherwise its content is undefined. You must make a copy of this string because the buffer will be overwritten by the next WORD. If the return value is ERROR_CHAR then *error_char* contains the offending error character. Note that the defined values for the tokens are such that the error values are less that EOL (end of line) and the valid tokens are all greater than EOL.

# 4 What you will create

Your job is to write a program named **wyshell.c** that

1. Prints a prompt, exactly "`$>` ". This will start at left margin and note that there is one space after the `>`.
2. Reads the lines of user input.
3. For **each** token on each line the program prints a line of output
   (a) the first word in each command string is ":`--`: word"
   (b) this means if a pipe or semicolon is encountered, the NEXT word is the first in a command string
   (c) the arguments to the command are " `--`: arg" (not the line starts with a space character NOT a colon)
   (d) replace "word" and "arg" with the strings contained in the variable *lexeme*
   (e) the special tokens are themselves, as in "`2>&1`"
   (f) the end of line is exactly " `--`: EOL"
   (g) errors are "error character: 27", "system error: error message", or "quote error". With system error, just use `perror("system error")` and end the program
   (h) the offending error character is stored in the variable *error_char* and if printed as an integer will give that number which is what I want.
4. After an ERROR_CHAR or QUOTE_ERROR return value, that message is printed and then the entire line of text is abandoned and the next line is processed.
5. After SYSTEM_ERROR, the program exits.
6. When the read from `stdin` returns End-Of-File (or an error), the program will exit.
7. Your program also needs to make sure that the command line does not have any semantic errors. Such as (list is not all-inclusive):
   - Multiple input/output redirection.
   - Redirection with no filename (string).
   - Pipe or redirection before the first or redirection before subsequent commands.

   If a semantic error is encountered, you will print an appropriate error message, abandon that line of input, and begin processing the next line. What are the error messages? Try the strings in whatever "normal" shell you use and use those responses as patterns for your error messages. DO NOT ADD "NOISE". Noise is cute/pretty/verbose extra information.

Program example run on last page.

# 5 Turn in

Upload a **tar archive** (either .tar or .tgz if compressed) that contains *wyshell.c*, *Makefile*, and if you decide that you need extra files for functions, those as well. You do NOT need to have a pseudocode file for this assignment. However, you would be well advised to create a DETAILED pseudocode file **before** you start writing code. Do not try to keep this all in your head.

**Example**

```
$> Some commands are quite long
:--: Some
 --: commands
 --: are
 --: quite
 --: long
 --: EOL
$> redir > error > file
:--: redir
 >
 --: error
Ambiguous output redirection
$> some
:--: some
 --: EOL
$> have pipes | in them | and others; use the semicolon to separate them
:--: have
 --: pipes
 |
:--: in
 --: them
 |
:--: and
 --: others
 ;
:--: use
 --: the
 --: semicolon
 --: to
 --: separate
 --: them
 --: EOL
$> and < f2 > f3
:--: and
 <
 --: f2
```

```
 >
 --: f3
 --: EOL
$>
```