Program 4
Due: Friday, Aprill 3
CSC399: Independent Study (Systems Programming)

Objectives

At the end of this assignment, the student will have:

- Examined a moderately complex C program - composed of multiple files.
- Modified (enhanced) a moderately complex C program.
- Used C's system library to perform operations such as fork and dup.

Instructions

Please write this program in the C programming language. The executable should be called `quShell` and it should compile with a `Makefile` that YOU supply. As always, zip all files into one file called `prog4.zip` and upload to Blackboard.

Remember, you ***must include*** a Makefile (a sample one is provided) that compiles the assignment with the default target `all`. The program ***must*** compile to the executable name given.

Note: It is very important to correctly follow the submission procedure, the naming conventions, and all other instructions. This makes it far easier for us to grade (we can use an automated script).

In particular, the zip file must be called (exactly) `prog4.zip`. In addition, unzipping this file should create all files in the current directory. That is, the zip file should only contain regular files and not folders or directories. Since you are submitting a Makefile, the condition is that running the command `make` should create an executable called `quShell`.

1 QU Shell (Name: quShell)

In this exercise, we are going to extend a provided program to create a basic (qu)shell.

Execution. Your code should be able to handle statements which consist of a sequence of zero or more commands separated by pipes. And it should handle lines with zero or more statements. An example would be:

```
a | b | c ; d | e ; ; f | g ; h
```

This executes the first statement by running a, piping the ouput to b, piping that output to c and printing that output to standard out. It then proceeds with the next statemt (d and e) and then (f and g) - blank statements are acceptable and then the final statement (h). The shell should execute the piped commands in parallel (except for the built-ins which can be done immediately) and must wait until all commands for that statement have completed before going to the next statement. Thus, statements are done sequentially.

You are ***not allowed*** to use the functions: `system` and `popen` or any other function that indirectly invokes the shell `/bin/sh` command. This defeats the entire purpose of our own shell (by using another!) To execute commands you will need to create a pipe (if needed), fork a process, duplicate the pipe to stdout and stdin (if needed), and run exec on the command.

Comments. A comment starts with a `#` symbol and continues to the end of the line. It is not counted when inside a quoted string: e.g. `"#"`.

Variable Assignment. Your shell will also support variables in two parts. One part supports assignment of variables, using the `set` command. The other supports variable substitution with variables identified between two `$` symbols. This built-in command is considered case insensitive (so `set` or `Set` all mean the same thing). The command `SET [var] [value]` takes two arguments. The first is the name of the variable to assign a value. The second is the value itself. Any other arguments are simply ignored. If there are no arguments, the statement does nothing. If the value argument is left off, the default value is the empty string `""`. If the variable already exists, it gets the new value given. If the variable does not exist, it is created and stored with the given value. Variable names are case sensitive.

The command `LIST` (which is also case insensitive) takes no arguments, but any arguments given are simply ignored. It lists all the variables currently stored in the shell. For example,

```
set CC /usr/bin/cc
set PATH /usr/bin:/usr/local/bin:/usr/sbin
set LIB /usr/lib
set cc /usr/bin/g++
set CC /usr/local/bin/gcc
list
cc: /usr/bin/g++
LIB: /usr/lib
PATH: /usr/bin:/usr/local/bin:/usr/sbin
CC: /usr/local/bin/gcc
```

Note that the order of the variables printed is not essential.

The code to perform variable assignment is already provided. It is listed here for completeness.

Variable Substitution. Any token that is not part of a single quoted string can have variable substitution. The variable name is tagged between two `$` symbols and is replaced by the value associated with the name. If the name does not exist, it simply uses the empty string. The token remains a single token (so spaces don't cause new tokens). In addition, the quShell can do recursive substitution. In this case, if after substituting all variables another substitution is possible, then another level is done. The substitutions stop after 10 levels. An example is the following short script:

```
$$ let a "A"
$$ let b "B$a$"
$$ let c 'C$b$'
$$ list
c: C$b$
b: BA
a: A
$$ echo $c$
CBA
$$ let b '$a$B'
$$ list
c: C$b$
b: $a$B
a: A
$$ echo $c$
CAB
$$ echo $dog$
```

```
$$ exit
```

Built-ins. Your code should support three more simple built-in commands.

- ~~`exit`: Causes the shell to exit.~~
- `status`: Toggles on/off reporting of the exit status of any statement. The exit status of a statement is defined to be the exit status of the last command executed in its piped sequence. For simplicity, the exit status of a built-in is always 0 (success).
- ~~`cd`: Change the working directory. If an argument is given, change the working directory to that argument. (Report an error to stderr if not found.) If no argument is given, go to the home directory of the user, accessed by the environment variable HOME. If the HOME environment does not exist, it should go to the root /. For this you will need to look at the functions `chdir` and `getenv`.~~

Interactive versus non-interactive modes. If the quShell is run with no arguments, it should run in interactive mode and provide the following prompt `"$$ "` which is output to standard output (stdout). Input should then come from standard input (stdin). If the quShell is run with an argument (or more - others ignored) then the first argument is the name of a script file to open and run line by line. In this case, no prompt is output though regular output would still go to standard output (stdout). In both cases, exit status and any other error messages generated by the shell should be output to standard error (stderr) with the following precursor `">> "`.

Improvements to Bash

I also want you to submit (in the same zip file) a file called README.TXT which contains your (group's) answer to the following question:

> If given the time and chance, what improvement(s) would you make to bash? Why? And (most importantly) how would you go about handling the change?

I just want a short, thought-out answer that is not more than 100 words.

Scoring

The following is the general scoring breakdown for this particular assignment.

- **(10 points)**: Executes single command statements properly (and waits for command to finish).
- **(10 points)**: Executes piped command properly - redirecting properly (and waits for commands to finish).
- **(15 points)**: Executes lines containing multiple statements with multiple pipes (and waits for commands to finish before going to successive statements).
- **(5 points)**: Recognizes and ignores comments.
- **(5 points)**: Simple variable substitution.
- **(5 points)**: Recursive variable substitution.
- **(5 points)**: Supports built-in exit command.
- **(5 points)**: Supports status toggling on/off (and actually prints out exit status information).
- **(5 points)**: Supports changing directory with argument.

- **(5 points)**: Supports changing directory to home directory (using HOME environment variable).
- **(10 points)**: Supports interactive and non-interactive modes *with proper prompt!*
- **(10 points)**: README.TXT file contains answer to improvement question.
- **(10 points)**: Commenting and style.
- **(-15 points)**: If uses `system`, `popen`, or any function that calls a shell command like `/bin/sh`.
- **(-30 points)**: If it does not compile with 'make' command or if instructions are not followed.

Input and Output

To help in both debugging and understanding, I am providing several "scripts" written for `quShell` along with the output from them. They each test some of the basics mentioned above but not EVERY detail.

My starting code is provided for you if you wish to use it. It might be useful to run the program through a debugger so you can follow the logic - skipping over functions that you know or can reason about and diving into functions that you don't quite see.