CSC345: Operating Systems
Program 1
Due: March 31

Instructions

Please write this program in either C or C++. The executable should be called `techShell` and it should compile with a `Makefile` that YOU supply. **(Specific submission instructions will be provided by Dr. Box later. My guess is that you will submit a tar-ball via email to him.)**

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. You should also include the following:

1. A README file containing:
   (a) Your name.
   (b) If you have not fully implemented all functionality then list the parts that work (and how to test them if it is not obvious) so that you can be sure to receive credit for the parts you do have working.
2. All the source files needed to compile, run and test your code (Makefile, .c or c++ files, optional test scripts). Do not submit object or executable files.
3. Output from your testing of your shell program. Make sure to demonstrate:
   (a) That simple Unix commands still work.
   (b) That piping still works.
   (c) I/O redirection
   (d) Error conditions

1 Tech Shell (Previous Version)

In this exercise, you are going to extend a previous techShell written by students from the CSC222 class. (Don't worry, one working code set is provided). Here is the description of the previous version. The extensions (and your assignment) follow this section.

Execution TechShell is able to handle statements which consist of a sequence of zero or more commands separated by pipes. And it handles 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 statement (d and e) and then (f and g) - blank statements are acceptable and then the final statement (h). The shell executes the piped commands in parallel (except for the built-ins which are done immediately) and waits until all commands for that statement have completed before going to the next statement. Thus, statements are done sequentially.

To execute commands the program creates a pipe (if needed), forks a process, duplicates the pipe to stdout and stdin (if needed), and runs 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. `"#"`.

Built-ins TechShell supports five simple built-in commands which are case insensitive. So, that SET and set and SeT all mean the same thing.

- `set [var] [value]`: Sets a variable and value. The first argument 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.
- `list`: Lists all the variables currently stored in the shell.
- `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`.

Substitution TechShell supports variable substitution as follows. 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 TechShell 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:

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

$$ exit
```

Interactive versus non-interactive modes If the techShell is run with no arguments, it runs in interactive mode and provides the following prompt `"$$ "` which is output to standard output (stdout). Input comes from standard input (stdin), unless redirects exist! If the techShell 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) and regular input also comes from standard input (stdin). In both cases, exit status and any other error messages generated by the shell are output to standard error (stderr) with the following precursor `">> "`.

Input and Output

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

2 Tech Shell (Revised Version)

Your assignment is to extend this shell to support some additional features. In particular, the new shell version should be able to support input and output redirects to files as well as a new builtin shell command `pwd`.

PWD This shell command should not run the executable `/bin/pwd`, instead it should be its own built-in command. It should print out the current working directory of the shell. In addition, if the directory is a subdirectory of the `HOME` directory, it should be truncated with `~`. For example, if the current home directory (use `getenv`) is `/home/foo` and the current working directory is `/home/foo/bar/one` the output of `pwd` should be `~/bar/one`.

Redirects In any command, the input, output, and error streams can be redirected to or from a file. These redirects override the piping redirects (in the case of a pipe). Here are some examples that should clarify:

```
$$ echo "Hi there" > fileA
   # The output of the echo command goes to the file fileA

$$ find . -name '.c' >& /tmp/fileErr > /home/foo/file.out
   # The output is sent to the file /home/foo/file.out
   # Any error output is sent to the file /tmp/fileErr
   # Behaviour is undefined if the two files are the same!
   #   (But the shell should NOT CRASH!)
   #   Note this is different than bash where >& sends both error AND standard output

$$ calc < /home/input > /tmp/output
   # The input to the calc program is read from /home/input
   # The output is sent to /tmp/output

$$ cat < foo | grep 'c' | tr [a-z] [A-Z] > FOO
   # Gets the input from the file foo (via redirect), sends it through the
   # pipe sequence and ultimately redirects the output to the file FOO.

$$ set OUTPUT foo
$$ echo Hello > $OUTPUT$
   # The output of echo (Hello) is redirected to the file foo.
```